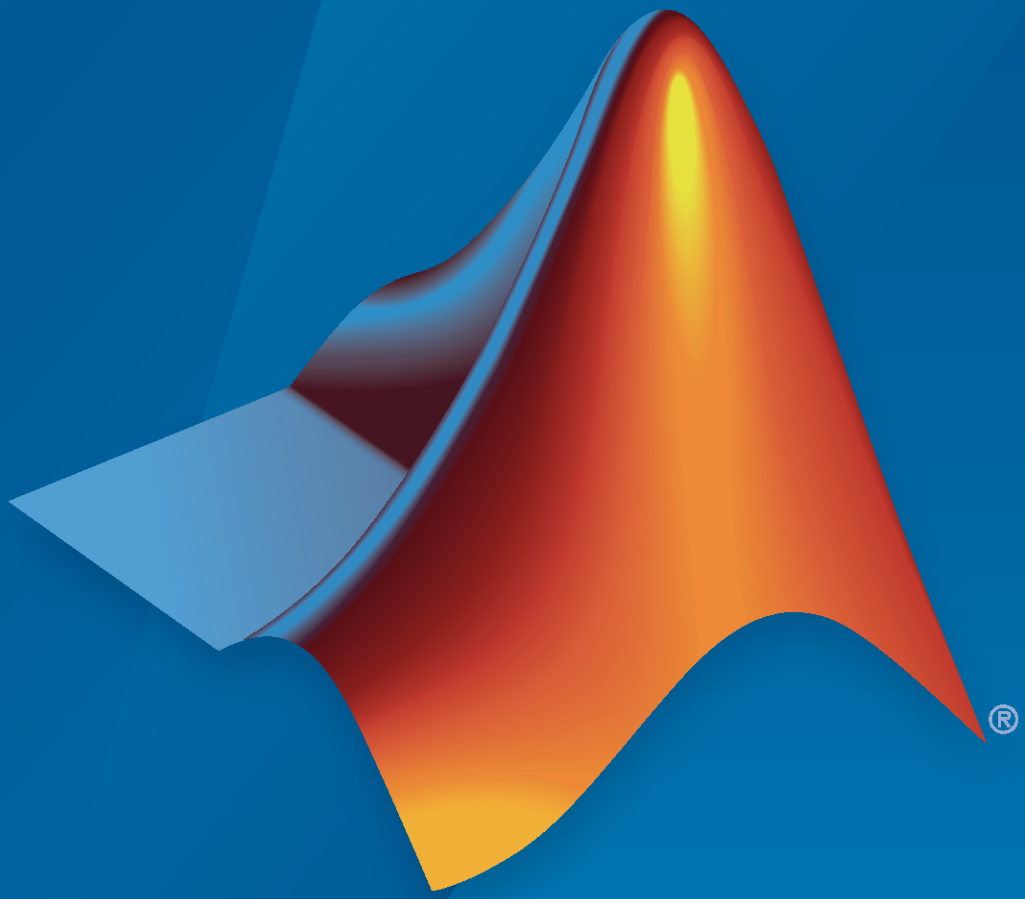
Financial Instruments Toolbox™

User's Guide



MATLAB®

R2022b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Financial Instruments Toolbox™ User's Guide

Revision History

September 2012	Online only	Version 1.0 (Release 2012b)
March 2013	Online only	Version 1.1 (Release 2013a)
September 2013	Online only	Version 1.2 (Release 2013b)
March 2014	Online only	Version 1.3 (Release 2014a)
October 2014	Online only	Version 2.0 (Release 2014b)
March 2015	Online only	Version 2.1 (Release 2015a)
September 2015	Online only	Version 2.2 (Release 2015b)
March 2016	Online only	Version 2.3 (Release 2016a)
September 2016	Online only	Version 2.4 (Release 2016b)
March 2017	Online only	Version 2.5 (Release 2017a)
September 2017	Online only	Version 2.6 (Release 2017b)
March 2018	Online only	Version 2.7 (Release 2018a)
September 2018	Online only	Version 2.8 (Release 2018b)
March 2019	Online only	Version 2.9 (Release 2019a)
September 2019	Online only	Version 2.10 (Release 2019b)
March 2020	Online only	Version 3.0 (Release 2020a)
September 2020	Online only	Version 3.1 (Release 2020b)
March 2021	Online only	Version 3.2 (Release 2021a)
September 2021	Online only	Version 3.3 (Release 2021b)
March 2022	Online only	Version 3.4 (Release 2022a)
September 2022	Online only	Version 3.5 (Release 2022b)

Contents

1 Getting Started

2 Interest-Rate Derivatives

3 Equity Derivatives

4 Hedging Portfolios

5 Mortgage-Backed Securities

6 Debt Instruments

7 Derivative Securities

8 Credit Derivatives

9 Interest-Rate Curve Objects

10 Numerix Workflows

11 Functions

A Derivatives Pricing Options

B Bibliography

Getting Started

Financial Instruments Toolbox Product Description

Design, price, and hedge complex financial instruments

Financial Instruments Toolbox provides functions for pricing, modeling, hedging, and analyzing cash flows, fixed-income securities, and derivative instruments (including equity, interest-rate, credit, and energy instruments). For interest-rate instruments, you can calculate price, yield, spread, and sensitivity values for various instrument types, including convertible bonds, mortgage-backed securities, treasury bills, bonds, swaps, caps, floors, and floating-rate notes. For derivative instruments, you can compute price, implied volatility, and Greeks using binomial trees, trinomial trees, Shifted SABR, Heston, Monte Carlo simulation, and other models. You can also connect to Numerix® CrossAsset Integration Layer for the valuation and risk management of fixed-income securities, OTC derivatives, structured products, and variable annuity products.

Interest-Rate-Based Derivatives

The toolbox provides functionality that supports the creation on page 1-6 and management on page 1-15 of these interest-rate-based instruments:

- Bonds
- Bond options (puts and calls)
- Bond with embedded options
- Caps
- Convertible bonds
- Fixed-rate notes
- Floating-rate notes
- Floors
- Swaps
- Swaption

Additionally, the toolbox provides functions to create *arbitrary cash flow instruments*. The toolbox provides pricing and sensitivity routines for these instruments. For more information, see “Pricing Using Interest-Rate Term Structure” on page 2-61,“Pricing Using Interest-Rate Tree Models” on page 2-81, and“Interest-Rate Derivatives Using Closed-Form Solutions” on page 2-99.

See Also

`instbond` | `instcap` | `instcbond` | `instcf` | `instfixed` | `instfloat` | `instfloor` | `instoptbnd` | `instoptembnd` | `instoptfloat` | `instoptemfloat` | `instrangefloat` | `instswap` | `instswaption`

Related Examples

- “Creating Instruments or Properties” on page 1-16

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34

Equity-Based Derivatives

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

- Asian options
- Barrier options
- Basket options
- Compound options
- Convertible bonds
- Digital options
- Lookback options
- Rainbow options
- Vanilla stock options (put and call options)

The toolbox also provides pricing and sensitivity routines for these instruments. (See “Pricing Equity Derivatives Using Trees” on page 3-64, “Equity Derivatives Using Closed-Form Solutions” on page 3-79, and “Basket Option” on page 3-22.)

See Also

`instasian` | `instbarrier` | `instcbond` | `instcompound` | `instlookback` | `instoptstock`

Related Examples

- “Creating Instruments or Properties” on page 1-16
- “Pricing Equity Derivatives Using Trees” on page 3-64

More About

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Energy Derivative Functions” on page 3-34

Expected Users

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

In designing Financial Instruments Toolbox documentation, we assume that 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 that 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, and so on)
- Responsible for instruments or analyses involving large sums of money
- Perhaps new to MATLAB®

Portfolio Creation Using Functions

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 character vector for `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 functions for 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)
  ```

- Convertible bond instrument

  ```
  InstSet = instcbond(CouponRate,Settle,Maturity,ConvRatio)
  ```

- 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 functions for 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)
```

- Convertible bond instrument

```
InstSet = instcbond(CouponRate,Settle,Maturity,ConvRatio)
```

- Lookback instrument

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

- Stock option instrument

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

See Also

`instadd` | `instaddfield` | `instdelete` | `instdisp` | `instfields` | `instfind` | `instget` | `instgetcell` | `instlength` | `instselect` | `instsetfield` | `insttypes` | `intenvset` | `hedgeopt` | `hedgeslf`

Related Examples

- “Creating Instruments or Properties” on page 1-16
- “Adding Instruments to an Existing Portfolio Using Functions” on page 1-8
- “Instrument Construction and Portfolio Management Using Functions” on page 1-15

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Price an Instrument Portfolio”

Adding Instruments to an Existing Portfolio Using Functions

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
```

See Also

`instadd` | `instaddfield` | `instdelete` | `instdisp` | `instfields` | `instfind` | `instget` | `instgetcell` | `instlength` | `instselect` | `instsetfield` | `insttypes` | `intenvset` | `hedgeopt` | `hedgeslf`

Related Examples

- “Portfolio Creation Using Functions” on page 1-6
- “Instrument Construction and Portfolio Management Using Functions” on page 1-15

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Price an Instrument Portfolio”

Pricing a Portfolio Using the Black-Derman-Toy Model

This example illustrates how the Financial Instruments Toolbox™ is used to create a Black-Derman-Toy (BDT) tree and price a portfolio of instruments using the BDT model.

Create the Interest Rate Term Structure

The structure `RateSpec` is an interest-rate term structure that defines the initial forward-rate specification from which the tree rates are derived. Use the information of annualized zero coupon rates in the table below to populate the `RateSpec` structure.

```
From To Rate

01 Jan 2005 01 Jan 2006 0.0275

01 Jan 2005 01 Jan 2007 0.0312

01 Jan 2005 01 Jan 2008 0.0363

01 Jan 2005 01 Jan 2009 0.0415

01 Jan 2005 01 Jan 2010 0.0458
```

```
StartDates = ['01 Jan 2005'];

EndDates =   ['01 Jan 2006';
              '01 Jan 2007';
              '01 Jan 2008';
              '01 Jan 2009';
              '01 Jan 2010'];

ValuationDate = ['01 Jan 2005'];
Rates = [0.0275; 0.0312; 0.0363; 0.0415; 0.0458];
Compounding = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: [5x1 double]
       StartDates: 732313
    ValuationDate: 732313
            Basis: 0
     EndMonthRule: 1
```

Specify the Volatility Model

Create the structure `VolSpec` that specifies the volatility process with the following data.

```
Volatility = [0.005; 0.0055; 0.006; 0.0065; 0.007];
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Volatility)
```

```
BDTVolSpec = struct with fields:
             FinObj: 'BDTVolSpec'
      ValuationDate: 732313
           VolDates: [5x1 double]
           VolCurve: [5x1 double]
    VolInterpMethod: 'linear'
```

Specify the Time Structure of the Tree

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.

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

```
BDTTimeSpec = struct with fields:
           FinObj: 'BDTTimeSpec'
    ValuationDate: 732313
         Maturity: [5x1 double]
      Compounding: 1
            Basis: 0
     EndMonthRule: 1
```

Create the BDT Tree

Use the previously computed values for `RateSpec`, `VolSpec`, and `TimeSpec` to create the `BDTTree`.

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

```
BDTTree = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 4]
        dObs: [732313 732678 733043 733408 733774]
        TFwd: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [4]}
      CFlowT: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [5]}
     FwdTree: {1x5 cell}
```

Observe the Interest Rate Tree

Visualize the interest-rate evolution along the tree by looking at the output structure `BDTTree`. `BDTTree` returns an inverse discount tree, which you can convert into an interest-rate tree with the `cvtree` function.

```
BDTTreeR = cvtree(BDTTree);
```

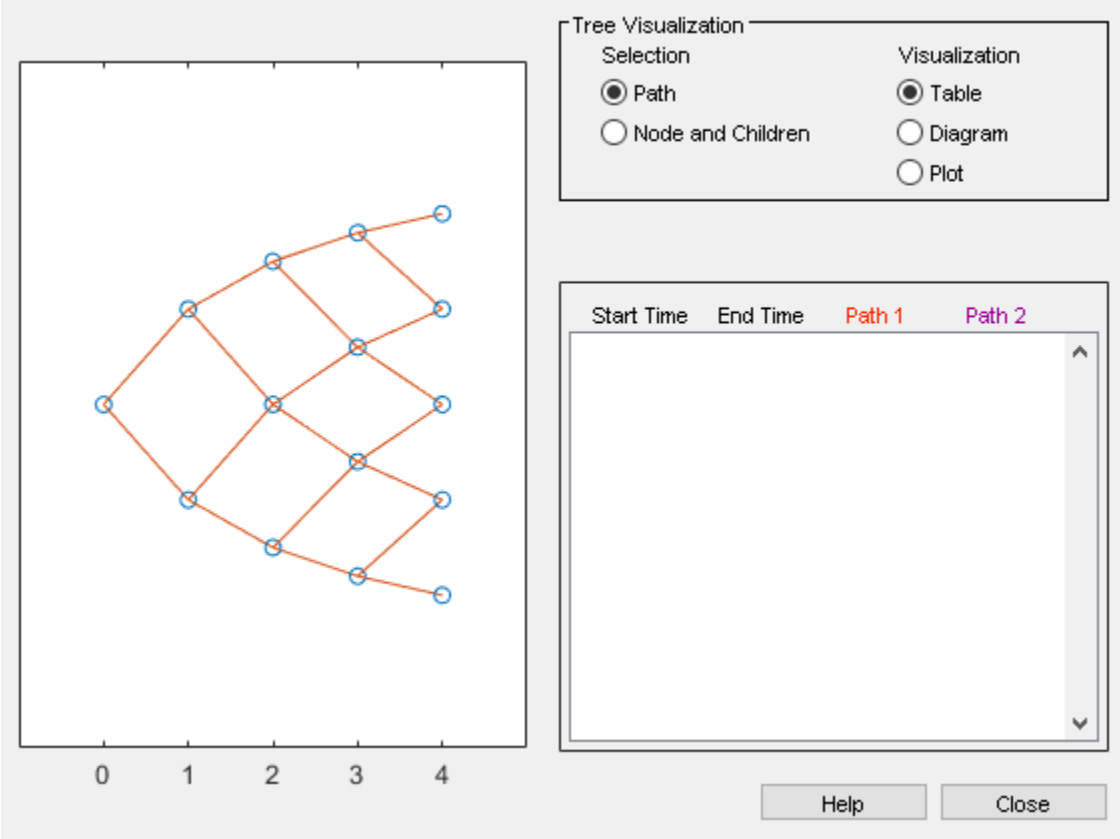
Look at the upper branch and lower branch paths of the tree:

```
%Rate at root node:
RateRoot      = treepath(BDTTreeR.RateTree, [0])
```

```
RateRoot = 0.0275


%Rates along upper branch:
RatePathUp    = treepath(BDTTreeR.RateTree, [1 1 1 1])

RatePathUp = 5×1

    0.0275
    0.0347
    0.0460
    0.0560
    0.0612



%Rates along lower branch:
RatePathDown = treepath(BDTTreeR.RateTree, [2 2 2 2])

RatePathDown = 5×1

    0.0275
    0.0351
    0.0472
    0.0585
    0.0653
```

You can also display a graphical representation of the tree to examine interactively the rates on the nodes of the tree until maturity. The function `treeviewer` displays the structure of the rate tree in the left pane. The tree visualization in the right pane is blank, but by selecting `Diagram` and clicking on the nodes you can examine the rates along the paths.

```
treeviewer(BDTTreeR)
```



Create an Instrument Portfolio

Create a portfolio consisting of two bond instruments and a option on the 5% Bond.

```
% Bonds
CouponRate = [0.04;0.05];
Settle = '01 Jan 2005';
Maturity = ['01 Jan 2009';'01 Jan 2010'];
Period = 1;

% Option
OptSpec = {'call'};
Strike = 98;
ExerciseDates = ['01 Jan 2010'];
AmericanOpt = 1;

InstSet = instadd('Bond',CouponRate, Settle,  Maturity, Period);
InstSet = instadd(InstSet,'OptBond', 2, OptSpec, Strike, ExerciseDates, AmericanOpt);
```

Examine the set of instruments contained in the variable `InstSet`.

```
instdisp(InstSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.04       01-Jan-2005    01-Jan-2009    1      0     1            NaN       NaN
2     Bond 0.05       01-Jan-2005    01-Jan-2010    1      0     1            NaN       NaN

Index Type    UnderInd OptSpec Strike ExerciseDates  AmericanOpt
```

```
3     OptBond 2        call    98     01-Jan-2010    1
```

Price the Portfolio Using a BDT Tree

Calculate the price of each instrument in the instrument set (`InstSet`) using `bdtprice`.

```
Price = bdtprice(BDTTree, InstSet)
```

```
Price = 3×1

   99.6374
  102.2460
    4.2460
```

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.

In the `Price` vector, the first element, `99.6374`, represents the price of the first instrument (4% Bond); the second element, `102.2460`, represents the price of the second instrument (5% Bond), and `4.2460` represents the price of the Option.

See Also

`instadd` | `instaddfield` | `instdelete` | `instdisp` | `instfields` | `instfind` | `instget` | `instgetcell` | `instlength` | `instselect` | `instsetfield` | `insttypes` | `intenvset` | `hedgeopt` | `hedgeslf`

Related Examples

- “Portfolio Creation Using Functions” on page 1-6
- “Instrument Construction and Portfolio Management Using Functions” on page 1-15

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Price an Instrument Portfolio”

Instrument Construction and Portfolio Management Using Functions

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 constructor functions are listed below.

Instrument	Constructor Function
Asian option	`instasian`
Barrier option	`instbarrier`
Bond	`instbond`
Bond option	`instoptbnd`
Arbitrary cash flow	`instcf`
Compound option	`instcompound`
Convertible bond	`instcbond`
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`. Also, 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 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 Instruments 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 Instruments Toolbox provides functions that enable you to:

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

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

instfind

The general syntax for `instfind` is

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

`InstSet` is the instrument set to search. Within `InstSet` instruments categorized by type, each type can have different data fields. The stored data field is a row vector or character vector 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 names specific to the instruments. `DataList` is a cell array or matrix of acceptable values for one or

more parameters specified in `FieldList`. `FieldName` and `Data` (therefore, `FieldList` and `DataList`) parameters must appear together or not at all.

`IndexSet` is a vector of integer indexes 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 character vector or cell array of character vectors 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 EndMonthRule IssueDate FirstCouponDate LastCouponDate StartDate Face Na
1     Bond 0.04       01-Jan-2000    01-Jan-2003    1      NaN   NaN          NaN       NaN             NaN            NaN       NaN  4%

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

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

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

instselect Examples

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

```
load InstSetExamples.mat
instdisp(ExampleInst)
```

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

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

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

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

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

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

instdisp(InstSet)

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

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

See Also

`instadd` | `instaddfield` | `instdelete` | `instdisp` | `instfields` | `instfind` | `instget` | `instgetcell` | `instlength` | `instselect` | `instsetfield` | `insttypes` | `intenvset` | `hedgeopt` | `hedgeslf`

Related Examples

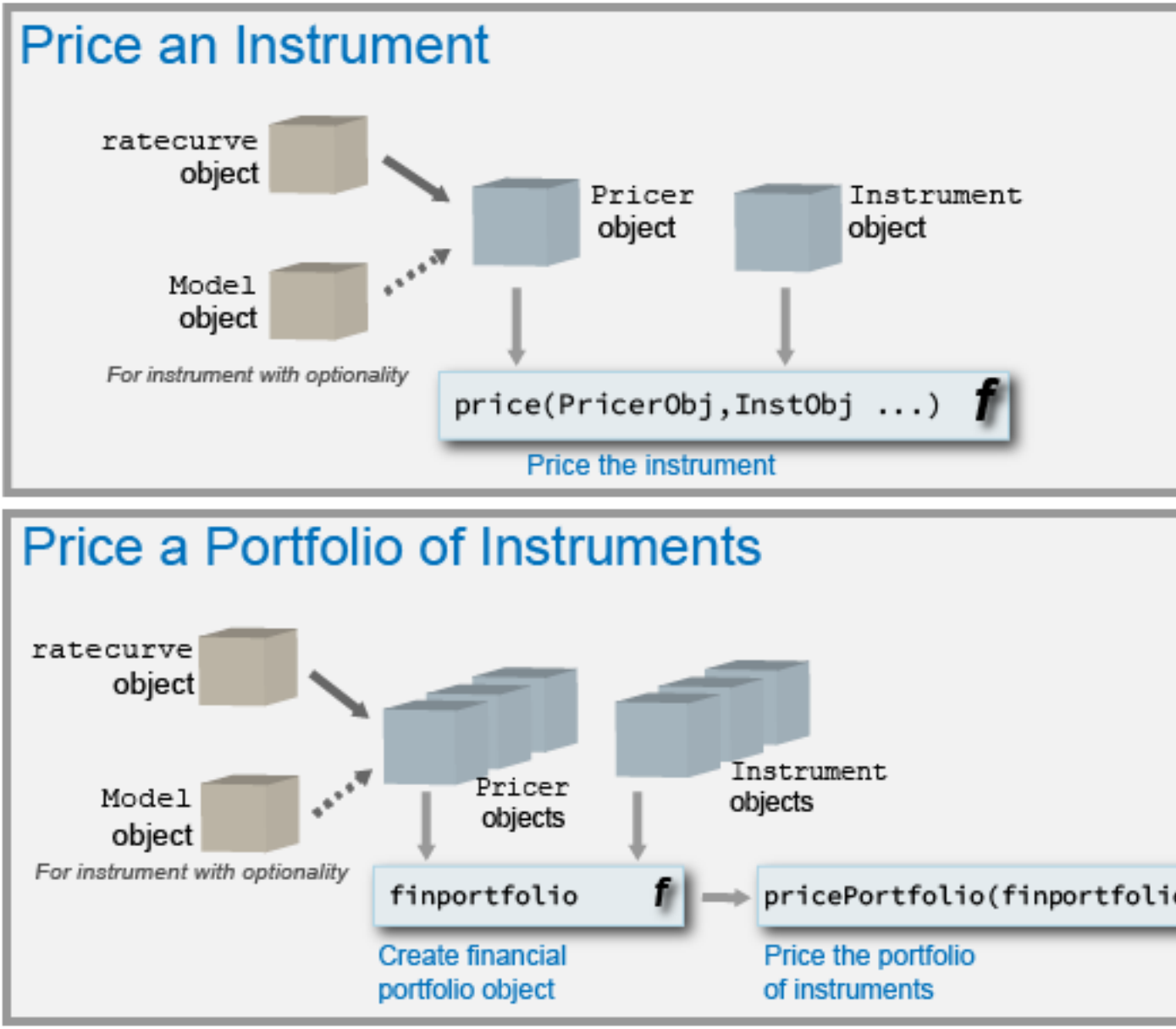
- “Portfolio Creation Using Functions” on page 1-6
- “Hedging Functions” on page 4-3
- “Hedging with hedgeopt” on page 4-4
- “Self-Financing Hedges with hedgeslf” on page 4-9
- “Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
- “Pricing and Hedging a Portfolio Using the Black-Karasinski Model” on page 4-13
- “Specifying Constraints with ConSet” on page 4-24
- “Portfolio Rebalancing” on page 4-26
- “Hedging with Constrained Portfolios” on page 4-28

More About

- “Hedging” on page 4-2
- “Supported Interest-Rate Instrument Functions” on page 2-3

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Choose Instruments, Models, and Pricers” on page 1-53
- “Price an Instrument Portfolio”

Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments



Object-Based Framework Workflow

Financial Instruments Toolbox supports an object framework for pricing financial instruments. There are three types of object constructors in the framework: `fininstrument` to create an instrument object, `finmodel` to create a model object, and `finpricer` to create a pricer object. The canonical workflow to price an instrument is:

1 Create an instrument object using `fininstrument`.

```
myInst = fininstrument(InstType,…)
```

2. Create a model object using `finmodel`.

   ```
   myModel = finmodel(ModelType,…)
   ```

3. Create a `ratecurve` object using `ratecurve`.

   ```
   myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
   ```

4. Create a pricer object using `finpricer`.

   ```
   myPricer = finpricer(PricerType,myModel,myRC,...)
   ```

5. Price the instrument using an associated `price` function.

   ```
   [Price, PriceResult] = price(myPricer, myInst,…)
   ```

For examples showing how to use this workflow to create different types of instruments, see:

- “Workflow to Price an Interest-Rate Instrument” on page 1-25
- “Workflow to Price an Inflation Instrument” on page 1-28
- “Workflow to Price an Equity, Commodity, or FX Instrument” on page 1-37
- “Workflow to Price a Credit Derivative Instrument” on page 1-40
- “Workflow for Creating and Analyzing a ratecurve and parametercurve” on page 1-46

You can also price an entire portfolio. After creating instrument objects and pricer objects, you can add the instrument and pricer objects to a `finportfolio` object and then price the portfolio using this workflow:

1. Create instrument objects using `fininstrument`.

   ```
   myInst1 = fininstrument(InstType,…)
   myInst2 = fininstrument(InstType,…)
   ```

2. Create model objects using `finmodel`.

   ```
   myModel = finmodel(ModelType,…)
   myModel2 = finmodel(ModelType,…)
   ```

3. Create one or more `ratecurve` objects using `ratecurve`.

   ```
   myRC1 = ratecurve('zero',Settle,ZeroDates,ZeroRates)
   myRC2 = ratecurve('zero',Settle,ZeroDates,ZeroRates)
   ```

4. Create a pricer object using `finpricer`.

   ```
   myPricer1 = finpricer(PricerType,myModel,myRC1,...)
   myPricer2 = finpricer(PricerType,myModel,myRC2,...)
   ```

5. Create a portfolio object using `finportfolio`.

   ```
   IP = finportfolio([MyInst1,MyInst2],[MyPricer1,MyPricer2],...)
   ```

6. Price the portfolio using `pricePortfolio`.

   ```
   [portPrice, portSens, instPrice, instSens] = price(IP)
   ```

For an example showing how to use this workflow to create a portfolio, see “Workflow to Create and Price a Portfolio of Instruments” on page 1-42.

See Also

`fininstrument` | `finmodel` | `finpricer`

More About

- “Choose Instruments, Models, and Pricers” on page 1-53
- “Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers” on page 1-70
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Workflow to Price an Interest-Rate Instrument

Price a financial instrument with a zero curve. Such an instrument has no embedded optionality, in other words, the cash flows are deterministic and valuing the instrument is simply a matter of generating the cash flows and then computing the present value of the cash flows by generating corresponding discount factors from the zero curve. For more information on the supported interest-rate instruments, see “Choose Instruments, Models, and Pricers” on page 1-53.

Price Vanilla Fixed Bond Instrument Using ratecurve and Discount Pricer

This example shows the workflow to price a vanilla `FixedBond` instrument when you use a `ratecurve` and a `Discount` pricing method.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",'Maturity',datetime(2022,9,15),'CouponRate',0.021,'Period',2,'Ba
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0210
                     Period: 2
                      Basis: 1
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond_instrument"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                 Type: "zero"
```

```
          Compounding: -1
                Basis: 0
                Dates: [10x1 datetime]
                Rates: [10x1 double]
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `FixedBond` Instrument

Use `price` to compute the price and sensitivities for the `FixedBond` instrument.

```
[Price, outPR] = price(outPricer, FixB,["all"])
```

```
Price = 104.5679

outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []


outPR.Results

ans=1×2 table
    Price       DV01
    ______    ________

    104.57    0.040397
```

See Also

`fininstrument` | `finmodel` | `finpricer`

Related Examples

- “Calibrate Shifted SABR Model Parameters for Swaption Instrument” on page 2-167

More About

- “Choose Instruments, Models, and Pricers” on page 1-53

- “Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers” on page 1-70
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Workflow to Price an Inflation Instrument

When pricing inflation derivatives and building inflation curves, incorporating seasonality can be a critical factor. The zero-coupon inflation swap rates typically have maturities that increase in whole number of years. As a result, the inflation curve is typically built from zero-coupon inflation swap rates on an annual basis. For more information on the supported inflation instruments, see “Choose Instruments, Models, and Pricers” on page 1-53.

Analyze Inflation-Indexed Instruments

This example shows how to analyze inflation-indexed instruments using Financial Toolbox™ and Financial Instruments Toolbox™.

Compute Real Prices and Yields for Inflation-Indexed Bonds

While inflation-indexed bonds have a great deal of variation in the design, for example, the length of the indexation lag, the majority of inflation-indexed bonds now have a three month lag. They are also capital-indexed, that is, the principal of the bond is indexed to inflation. Therefore, the coupon rate of the bond is constant, but the actual coupon payments vary as the principal of the bond is indexed to inflation.

Specifically, the indexation is done with the following ratio:

$$IndexRatio = \frac{CPI_{Ref}}{CPI_{Base}}$$

where CPI_{Base} is the level of the consumer price index (or equivalent price measure) at the time of the bond's issue and CPI_{Ref} is the reference CPI.

Typically, you compute the CPI_{Ref} by interpolating between the index data of a known inflation-index curve. To compute the cash flows for an inflation-indexed bond, you simply compute the appropriate reference CPI and Index Ratio.

The market convention for inflation-indexed bonds is to quote the price and yield using the actual (that is, unadjusted) coupon, which means that your quote is a real price and yield. To get a real price and yield, you can use the Financial Toolbox™ functions `bndprice` and `bndyield`. For example:

```
Price = 124 + 9/32;
Settle = datetime(2009,9,28);
Coupon = .03375;
Maturity = datetime(2032,4,15);

RealYield = bndyield(Price,Coupon,Settle,Maturity);
disp(['Real Yield: ', num2str(RealYield*100) '%'])
```

```
Real Yield: 2.0278%
```

Construct Nominal, Real, and Inflation Curves

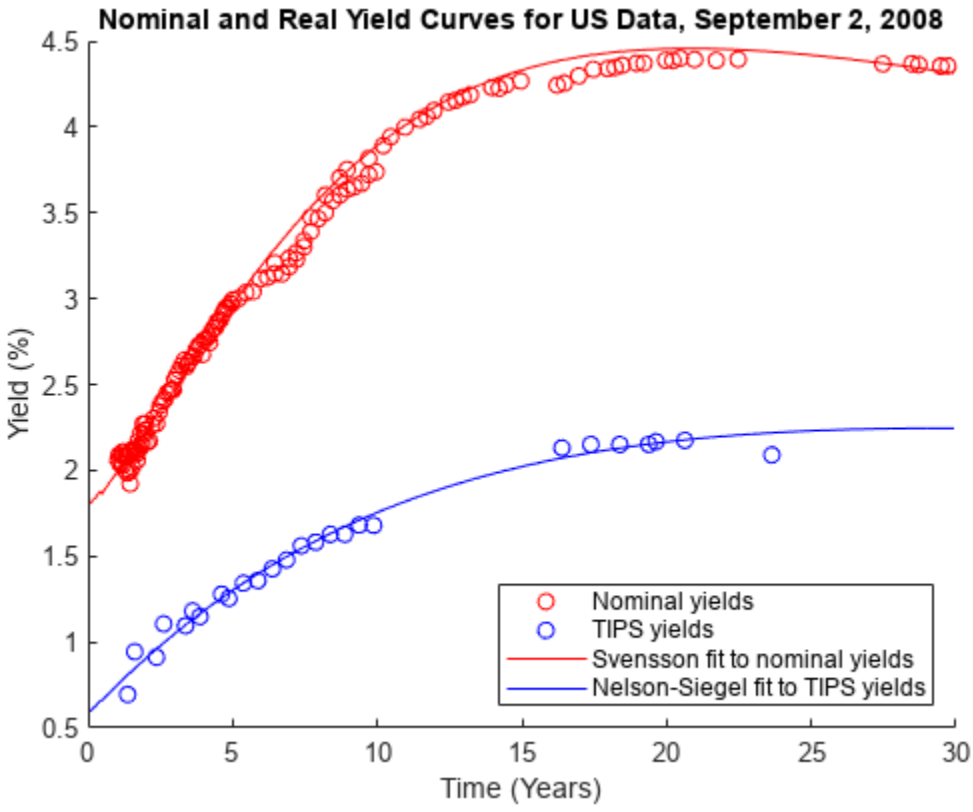
With the advent of the inflation-indexed bond market, real curves can be constructed in a similar fashion to nominal curves. Using the available market data, you can construct the real curve and compare it to the nominal curve.

Note that one issue relates to the indexation lag of the bonds. As stated previously, typically the indexation lag is three months, which means that the inflation compensation is not actually matched up with the maturity or the coupon payments of the bond. While Anderson and Sleath [1] discuss an approach to resolving this discrepancy, for this example, the lag is simply noted.

You can use the `fitNelsonSiegel` and `fitSvensson` functions in the Financial Instruments Toolbox™ to create `parametercurve` objects that fit Nelson-Siegel and Svensson models to real and nominal yield curves in the US. The Nelson-Siegel model typically places restrictions on the model parameters to ensure that the interest rates are always positive. However, real interest rates can be negative, which means that these Nelson-Siegel restrictions are not used in the case below.

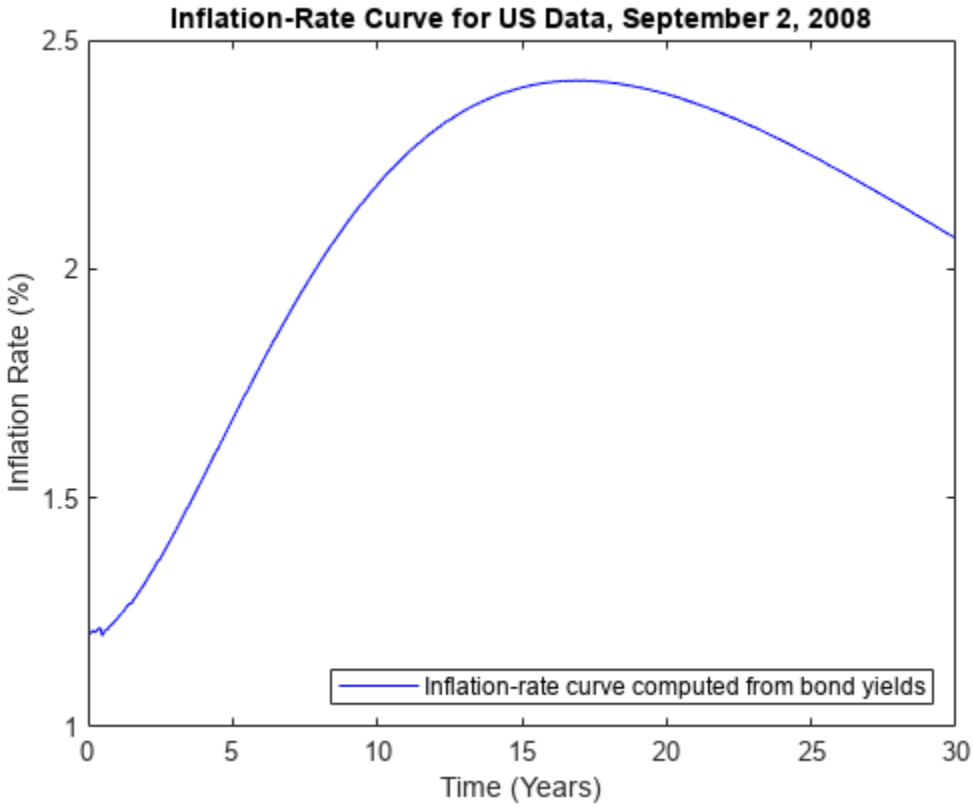
```
% Load the data.
load usbond_02Sep2008
Settle = datetime(2008, 9, 2);
NominalTimeToMaturity = yearfrac(Settle,NominalMaturity);
TIPSTimeToMaturity = yearfrac(Settle,TIPSMaturity);

% Compute the yields.
NominalYield = bndyield(NominalPrice,NominalCoupon,Settle,NominalMaturity);
TIPSYield = bndyield(TIPSPrice,TIPSCoupon,Settle,TIPSMaturity);

% Plot the yields.
scatter(NominalTimeToMaturity,NominalYield*100,'r');
hold on;
scatter(TIPSTimeToMaturity,TIPSYield*100,'b');

% Fit the real yield curve using fitNelsonSiegel.
nInst = numel(TIPSCoupon);
TIPSBonds(nInst,1) = fininstrument.FinInstrument;
for ii=1:nInst
    TIPSBonds(ii) = fininstrument("FixedBond",'Maturity',TIPSMaturity(ii),...
        'CouponRate',TIPSCoupon(ii));
end

TIPSNelsonSiegel = fitNelsonSiegel(Settle,TIPSBonds,TIPSPrice);

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

% Fit the nominal yield curve using fitSvensson.
nInst = numel(NominalCoupon);
NominalBonds(nInst,1) = fininstrument.FinInstrument;
for ii=1:nInst
    NominalBonds(ii) = fininstrument("FixedBond",'Maturity',NominalMaturity(ii),...
        'CouponRate',NominalCoupon(ii));
end

NominalSvensson = fitSvensson(Settle,NominalBonds,NominalPrice);

Solver stopped prematurely.

lsqnonlin stopped because it exceeded the function evaluation limit,
options.MaxFunctionEvaluations = 6.000000e+02.

% Plot the nominal and real yield curves.
PlotDates = (Settle+calmonths(1):calmonths(1):Settle+calyears(30)-1)';
```

```
PlotTimeToMaturity = yearfrac(Settle,PlotDates);

TIPSNelsonSiegelZeroRates = zerorates(TIPSNelsonSiegel,PlotDates);
TIPSNelsonSiegelParYields = zero2pyld(TIPSNelsonSiegelZeroRates,PlotDates,Settle, ...
    'InputCompounding', -1, 'OutputCompounding', 2);

NominalSvenssonZeroRates = zerorates(NominalSvensson,PlotDates);
NominalSvenssonParYields = zero2pyld(NominalSvenssonZeroRates,PlotDates,Settle, ...
    'InputCompounding', -1, 'OutputCompounding', 2);

plot(PlotTimeToMaturity,NominalSvenssonParYields*100,'r')
plot(PlotTimeToMaturity,TIPSNelsonSiegelParYields*100,'b')
hold off;

title('Nominal and Real Yield Curves for US Data, September 2, 2008')
xlabel('Time (Years)')
ylabel('Yield (%)')
legend({'Nominal yields','TIPS yields','Svensson fit to nominal yields',...
    'Nelson-Siegel fit to TIPS yields'},'location','southeast')
```




```
% Create an inflation-rate curve by subtracting the real curve from the
% nominal curve.
InflationRateCurve = ratecurve("zero", Settle, PlotDates, ...
    NominalSvenssonZeroRates - TIPSNelsonSiegelZeroRates);

figure
plot(PlotTimeToMaturity, zero2pyld(...
```

```
    zerorates(InflationRateCurve, PlotDates), PlotDates, Settle, ...
    'InputCompounding', -1, 'OutputCompounding', 2)*100,'b');

title('Inflation-Rate Curve for US Data, September 2, 2008')
xlabel('Time (Years)')
ylabel('Inflation Rate (%)')
legend({'Inflation-rate curve computed from bond yields'},'location','southeast')
```



Constructing Inflation Curves from Zero-Coupon Inflation Swaps

Inflation-linked derivatives have also experienced growth in the market. Some of the most liquidly traded inflation derivatives are zero coupon inflation swaps (`ZeroCouponInflationSwap`) and year-on-year inflation swaps (`YearYearInflationSwap`).

In a zero-coupon inflation swap, the inflation payer agrees to pay the rate of inflation at maturity (lagged by a certain amount) compounded by the number of years. The inflation receiver typically pays a fixed rate, again compounded by the tenor of the instrument. At the inception of the zero-coupon inflation swap, the fixed rate is set to the projected inflation rate for the life of the swap. This rate is called the "breakeven inflation swap rate" and it is quoted in the market [6].

Using the notation from Hurd and Relleen, you compute the rate as:

$$(1 + Rate_{swap})^T = (1 + Inflation_{t-L,\,t+T-L})^T$$

where t is the current time, T is the tenor, and L is the lag. [5]

At maturity, the actual cash flows of the zero-coupon inflation swap are:

$$FixedLeg = N \times [(1 + k)^M - 1]$$

$$InflationLeg = N \times \left[\frac{I(T_M)}{I_0} - 1\right]$$

where

- $N$is the reference notional of the swap.
- k is the fixed inflation rate.
- $M$is the number of years for the life of the swap.
- $I(T_M)$is the inflation index at the maturity date with some lag (for example, three months).
- I_0 is the inflation index at the start date with some lag (for example, three months).

While the fixed-leg cash flow might be different from the actual inflation-leg cash flow at maturity, the fixed breakeven inflation swap rate of the zero-coupon inflation swap represents the projected inflation rate for the tenor of the swap at inception. You can build an inflation curve from a series of breakeven zero-coupon inflation swap rates starting on the same date and maturing on different dates. Here, the dates are already adjusted with the appropriate indexation lag to simplify the notation:

$$I(0, T_{1Y}) = I(T_0)(1 + b(0; T_0, T_{1Y}))^{T_{1Y} - T_0}$$

$$I(0, T_{2Y}) = I(T_0)(1 + b(0; T_0, T_{2Y}))^{T_{2Y} - T_0}$$

$$I(0, T_{3Y}) = I(T_0)(1 + b(0; T_0, T_{3Y}))^{T_{3Y} - T_0}$$

. . .

$$I(0, T_i) = I(T_0)(1 + b(0; T_0, T_i))^{T_i - T_0}$$

where

- $I(0, T_i)$ is the breakeven inflation index reference number for maturity date T_i.
- $I(T_0)$ is the base inflation index value for the starting date T_0.
- $b(0; T_0, T_i)$ is the breakeven inflation rate for the zero-coupon inflation swap maturing on T_i.

You can get your inflation curve this by using the `inflationbuild` function to create an `inflationcurve` object. To build an `inflationcurve` from zero-coupon inflation swap rates, first define the base inflation date and the corresponding base inflation-index value.

```
% Define the base inflation date and index value for the inflation-index
% curve.
BaseDate = datetime(2020,6,1);
BaseIndexValue = 100;
```

Then, define the zero-coupon inflation swap rates and the corresponding maturity dates already adjusted with the appropriate indexation lag.

```
% Define the zero-coupon inflation swap rates and maturity dates.
ZCISTimes = (calyears([1 2 3 4 5 7 10 20 30]))';
ZCISRates = [0.42 0.54 0.76 0.87 0.92 1.39 1.71 2.01 2.46]'./100
```

```
ZCISRates = 9×1

    0.0042
    0.0054
    0.0076
    0.0087
    0.0092
    0.0139
    0.0171
    0.0201
    0.0246
```

```
ZCISDates = BaseDate + ZCISTimes
```

```
ZCISDates = 9x1 datetime
   01-Jun-2021
   01-Jun-2022
   01-Jun-2023
   01-Jun-2024
   01-Jun-2025
   01-Jun-2027
   01-Jun-2030
   01-Jun-2040
   01-Jun-2050
```

In pricing inflation derivatives and building inflation curves, incorporating seasonality can be a critical factor. The zero-coupon inflation swap rates typically have maturities that increase in whole number of years. As a result, the inflation curve is typically built from zero-coupon inflation swap rates on an annual basis. However, when computing inflation-index values for monthly periods that are not whole number of years, you can make seasonal adjustments to reflect the seasonal patterns of inflation within the year. These 12 monthly seasonal rates are annualized and they add up to zero to ensure that the cumulative seasonal adjustments are reset to zero every year. In the `inflationbuild` function and the `inflationcurve` object, you define these seasonal rates using the `'Seasonality'` name-value pair argument and they are internally corrected to ensure that they add to zero.

```
% Define the 12 monthly seasonal rates.
%
% Months:
%    Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
%     1     2     3     4     5     6     7     8     9    10    11    12
% Seasonal Rates (percent):
%   -6.34 -3.00 -1.34  3.34  5.34  3.66  8.66  5.66 -2.34 -2.66 -4.66 -6.32
SeasonalRates = [-6.34 -3.00 -1.34 3.34 5.34 3.66 8.66 5.66 -2.34 -2.66 -4.66 -6.32]./100
```

```
SeasonalRates = 1×12

   -0.0634   -0.0300   -0.0134    0.0334    0.0534    0.0366    0.0866    0.0566   -0.0234   -0.0
```

```
% Build an inflation-index curve from zero-coupon inflation swap rates.
myInflationCurve = inflationbuild(BaseDate, BaseIndexValue, ...
    ZCISDates, ZCISRates, 'Seasonality', SeasonalRates)
```

```
myInflationCurve =
  inflationcurve with properties:
```

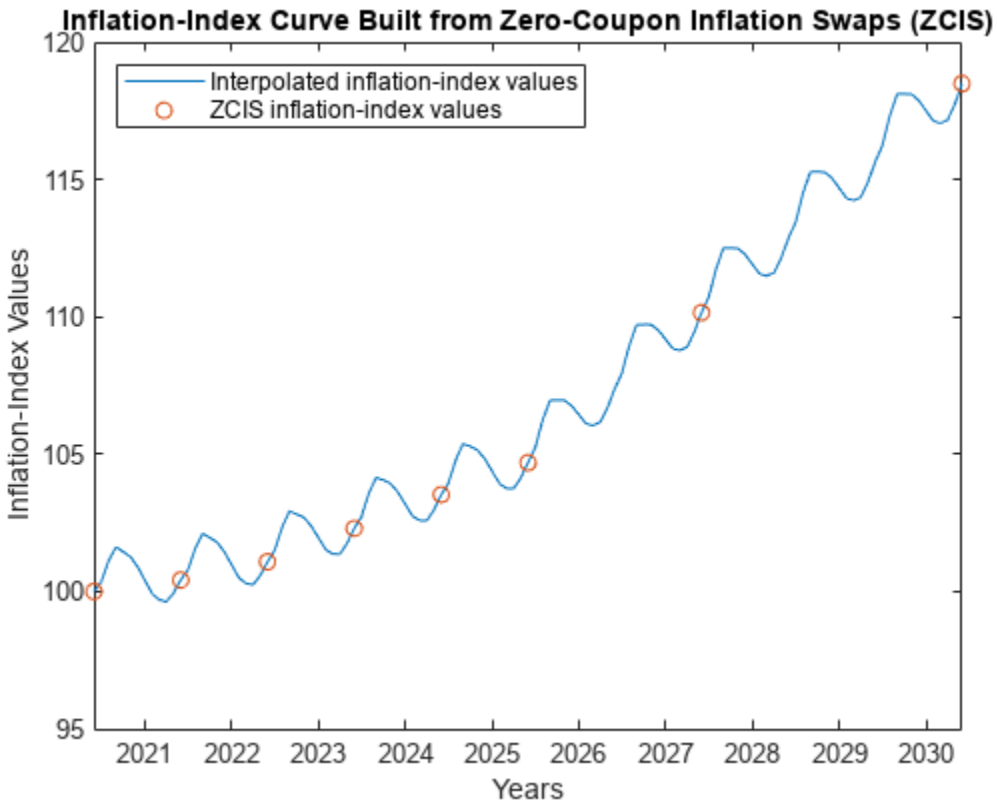
```
                Basis: 0
                Dates: [10x1 datetime]
 InflationIndexValues: [10x1 double]
ForwardInflationRates: [9x1 double]
          Seasonality: [12x1 double]
```

Once you have created the `inflationcurve` object, compute the inflation-index values for each month using `indexvalues`.

```
% Compute the inflation-index values.
IndexPlotDates = (BaseDate:calmonths(1):BaseDate+calyears(10))';
IndexPlotValues = indexvalues(myInflationCurve, IndexPlotDates);
```

To visualize the seasonal patterns of inflation that occur within each year, plot the computed inflation-index values.

```
% Plot the inflation-index curve.
figure; plot(IndexPlotDates, IndexPlotValues)
hold on;
plot(myInflationCurve.Dates(1:8), myInflationCurve.InflationIndexValues(1:8), 'o')
hold off;

title('Inflation-Index Curve Built from Zero-Coupon Inflation Swaps (ZCIS)')
xlabel('Years')
ylabel('Inflation-Index Values')
legend({'Interpolated inflation-index values','ZCIS inflation-index values'},'location','northwes
```



Price Inflation-Indexed Instruments Using an Inflation Curve

With the `inflationcurve` object created, you can price inflation-indexed instruments such as zero-coupon inflation swaps (`ZeroCouponInflationSwap`), year-on-year inflation swaps (`YearYearInflationSwap`), and inflation-indexed bonds (`InflationBond`).

First, create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,25);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0043 0.0051 0.0062 0.0072 0.0096 0.0121 0.0172 0.0241 0.0302 0.0308]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 25-Sep-2020
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Using the `ratecurve` and `inflationcurve` objects as inputs, create an `Inflation` pricer object using `finpricer`.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

    DiscountCurve: [1x1 ratecurve]
   InflationCurve: [1x1 inflationcurve]
```

Create an `InflationBond` instrument using `fininstrument`.

```
IssueDate = datetime(2020,9,20);
Maturity = datetime(2025,9,20);
CouponRate = 0.023;

InflationBond = fininstrument("InflationBond",'IssueDate',IssueDate,'Maturity',Maturity,'CouponRa
```

```
InflationBond =
  InflationBond with properties:

                 CouponRate: 0.0230
                     Period: 2
                      Basis: 0
                  Principal: 100
   DaycountAdjustedCashFlow: 0
                        Lag: 3
```

```
BusinessDayConvention: "actual"
              Holidays: NaT
          EndMonthRule: 1
             IssueDate: 20-Sep-2020
       FirstCouponDate: NaT
        LastCouponDate: NaT
              Maturity: 20-Sep-2025
                  Name: ""
```

Here, the default indexation lag is three months and the bond issue date is `20-Sep-2020`. The first date on the inflation curve of the pricer must be on or before `20-Jun-2020` to price this instrument. In this example, the first date on the inflation curve of the pricer is `01-Jun-2020`.

Price the `InflationBond` instrument by using the `price` function for the `Inflation` pricer.

```
InflationBondPrice = price(outPricer, InflationBond)
```

```
InflationBondPrice = 110.1314
```

References

This example is based on the following papers and journal articles:

[1] Anderson N. and J. Sleath. "New Estimates of the UK Real and Nominal Yield Curves." Bank of England, working paper 126, 2001.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice: With Smile, Inflation and Credit*. Springer, 2006.

[3] Deacon, M., A. Derry, and D. Mirfendereski. *Inflation-Indexed Securities: Bonds, Swaps, and Other Derivatives*. Wiley Finance, 2004.

[4] Gurkaynak, R. S., B.P. Sack, and J.H. Wright. "The TIPS Yield Curve and Inflation Compensation." FEDS Working Paper No. 2008-05, October 2008.

[5] Hurd, M. and J. Relleen. "New Information from Inflation Swaps and Index-linked Bonds." Quarterly Bulletin, Spring 2006.

[6] Kerkhof, J. "Inflation Derivatives Explained." Lehman Brothers, 2005.

See Also

`fininstrument` | `finmodel` | `finpricer`

More About

- “Choose Instruments, Models, and Pricers” on page 1-53
- “Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers” on page 1-70
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Workflow to Price an Equity, Commodity, or FX Instrument

Price a Vanilla option with the Black-Scholes closed form formula. For more information on the supported equity, commodity, or FX instruments, see “Choose Instruments, Models, and Pricers” on page 1-53.

Price Vanilla Instrument Using Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price a `Vanilla` instrument when you use a `BlackScholes` model and a `BlackScholes` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2018,5,1),'Strike',29,'OptionType','
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 01-May-2018
           Strike: 29
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2500
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2019,1,1);
Rate = 0.05;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
```

```
                Basis: 1
                Dates: 01-Jan-2019
                Rates: 0.0500
               Settle: 01-Jan-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackScholes Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',30,'D
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 30
    DividendValue: 0.0450
     DividendType: "continuous"
```

Price Vanilla Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 1.2046

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma      Lambda      Vega       Rho       Theta
    ______    ________    ________    _______    ______    _______    _______

    1.2046    -0.36943    0.086269    -9.3396    6.4702    -4.0959    -2.3107
```

See Also

`fininstrument` | `finmodel` | `finpricer`

Related Examples

- “Price Vanilla Instrument Using Heston Model and Multiple Different Pricers” on page 3-125

More About

- “Choose Instruments, Models, and Pricers” on page 1-53
- “Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers” on page 1-70
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Workflow to Price a Credit Derivative Instrument

Price and analyze a credit default swap. For more information on the supported credit derivative instruments, see “Choose Instruments, Models, and Pricers” on page 1-53.

Price CDS Instrument Using Default Probability Curve and Credit Pricer

This example shows the workflow to price a `CDS` instrument when you use a `defprobcurve` model and a `Credit` pricing method.

Create `CDS` Instrument Object

Use `fininstrument` to create a `CDS` instrument object.

```
CDS = fininstrument("CDS",'Maturity',datetime(2021,9,15),'ContractSpread',15,'Notional',20000,'Pe
```

```
CDS =
  CDS with properties:

           ContractSpread: 15
                 Maturity: 15-Sep-2021
                   Period: 4
                    Basis: 3
             RecoveryRate: 0.4000
    BusinessDayConvention: "follow"
                 Holidays: NaT
        PayAccruedPremium: 1
                 Notional: 20000
                     Name: "CDS_instrument"
```

Create `defprobcurve` Object

Create a `defprobcurve` object using `defprobcurve`.

```
Settle = datetime(2020,9,20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities,'Basis',5)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                  Settle: 20-Sep-2020
                   Basis: 5
                   Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Credit` Pricer Object

Use `finpricer` to create a `Credit` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("credit",'DefaultProbabilityCurve',DefaultProbCurve,'DiscountCurve',myRC)
```

```
outPricer =
  Credit with properties:

             DiscountCurve: [1x1 ratecurve]
                  TimeStep: 10
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price `CDS` Instrument

Use `price` to compute the price for the `CDS` instrument.

```
Price = price(outPricer,CDS)
```

```
Price = 52.7426
```

See Also

`fininstrument` | `finmodel` | `finpricer`

More About

- “Choose Instruments, Models, and Pricers” on page 1-53
- “Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers” on page 1-70
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Workflow to Create and Price a Portfolio of Instruments

Price a portfolio of instruments — each with their own associated pricer. This workflow is common in risk situations where you may have a portfolio of securities that need to be valued.

Create and Price Portfolio of Instruments

Use `finportfolio` and `pricePortfolio` to create and price a portfolio of interest-rate and equity instruments. The portfolio contains a vanilla `FixedBond`, an `OptionEmbeddedFixedBond`, a `Vanilla` European call option, a `Vanilla` American call option, and an `Asian` call option.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates);
```

Create the Instrument Objects

Use `fininstrument` to create the instrument objects.

```
% Vanilla FixedBond
CouponRate = 0.0325;
Maturity = datetime(2038,3,15);
Period = 1;
VanillaBond = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',CouponRate,...
    'Period',Period,'Name',"VanillaBond")
```

```
VanillaBond =
  FixedBond with properties:

                 CouponRate: 0.0325
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Mar-2038
                        Name: "VanillaBond"
```

```
% OptionEmbeddedBond
Maturity = datetime(2024,9,15);
CouponRate = 0.035;
Strike = 100;
```

```
ExerciseDates = datetime(2023,9,15);
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
Period = 1;
CallableBond = fininstrument("OptionEmbeddedFixedBond", "Maturity",Maturity,...
    'CouponRate',CouponRate,'Period',Period, ...
    'CallSchedule',CallSchedule,...
    'Name',"CallableBond");

% Vanilla European call option
ExerciseDate = datetime(2022,1,1);
Strike = 96;
OptionType = 'call';
CallOpt = fininstrument("Vanilla",'ExerciseDate',ExerciseDate,'Strike',Strike,...
    'OptionType',OptionType, 'Name',"EuropeanCallOption")

CallOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 01-Jan-2022
           Strike: 96
             Name: "EuropeanCallOption"



% Vanilla American call option
ExerciseDate = datetime(2023,1,1);
Strike = 97;
OptionType = 'call';
CallOpt_American = fininstrument("Vanilla",'ExerciseDate',ExerciseDate,'Strike',Strike,...
    'OptionType',OptionType, 'ExerciseStyle', "american", ...
    'Name',"AmericanCallOption")

CallOpt_American =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2023
           Strike: 97
             Name: "AmericanCallOption"



% Asian call option
ExerciseDate = datetime(2023,1,1);
Strike = 102;
OptionType = 'call';
CallOpt_Asian = fininstrument("Asian",'ExerciseDate',ExerciseDate,'Strike',Strike,...
    'OptionType',OptionType,'Name',"AsianCall")

CallOpt_Asian =
  Asian with properties:

         OptionType: "call"
             Strike: 102
        AverageType: "arithmetic"
       AveragePrice: 0
```

```
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 01-Jan-2023
                Name: "AsianCall"
```

Create Model Objects

Use `finmodel` to create `HullWhite` and `BlackScholes` model objects.

```
% Create Hull-White model
Vol = 0.01;
Alpha = 0.1;
HWModel = finmodel("hullwhite",'alpha',Alpha,'sigma',Vol);

% Create Black-Scholes model
Vol = .1;
SpotPrice = 95;
BlackScholesModel = finmodel("BlackScholes",'Volatility',Vol);
```

Create Pricer Objects

Use `finpricer` to create `Discount`, `IRTree`, `BlackScholes`, `Levy`, and `BjerksundStensland` pricer objects and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
% Create Discount pricer
DiscPricer = finpricer("Discount","DiscountCurve",ZeroCurve);

% Create Hull-White tree pricer
TreeDates = Settle + calyears(1:30);
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,...
    'TreeDates',TreeDates');

% Create BlackScholes, Levy, and BjerksundStensland pricers
BLSPricer = finpricer("analytic",'DiscountCurve',ZeroCurve,'Model',BlackScholesModel,'SpotPrice'
LevyPricer = finpricer("analytic",'DiscountCurve',ZeroCurve,'Model',BlackScholesModel,...
                       'SpotPrice',SpotPrice,'PricingMethod',"Levy");
BJSpricer = finpricer("analytic",'DiscountCurve',ZeroCurve,'Model',BlackScholesModel,...
                       'SpotPrice',SpotPrice,'PricingMethod',"BjerksundStensland");
```

Create `finportfolio` Object

Create a `finportfolio` object that contains all of the instrument and pricer objects using `finportfolio`.

```
myPort = finportfolio([VanillaBond CallableBond CallOpt CallOpt_American CallOpt_Asian]',...
                           [DiscPricer HWTreePricer BLSPricer BJSpricer LevyPricer]')
```

```
myPort =
  finportfolio with properties:

    Instruments: [5x1 fininstrument.FinInstrument]
        Pricers: [5x1 finpricer.FinPricer]
    PricerIndex: [5x1 double]
       Quantity: [5x1 double]
```

Price Portfolio

Use `pricePortfolio` to compute the price and sensitivities for the portfolio and the instruments in the portfolio.

```
[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(myPort)
```

```
PortPrice = 237.3275

InstPrice = 5×1

  107.4220
  110.8389
    7.5838
    8.8705
    2.6123


PortSens=1×8 table
    Price      Delta     Gamma    Lambda     Vega      Theta      Rho       DV01
    _____    _______    _____    ______    ______    _______    ______    ______

    237.33    -546.39    2840     26.354    124.28    -4.0673    418.68    0.1579


InstSens=5×8 table
                          Price      Delta      Gamma      Lambda     Vega       Theta       Rho
                          _____    _______    ________    ______    _______    ________    ____

    VanillaBond           107.42       NaN         NaN       NaN        NaN         NaN      Na
    CallableBond          110.84    -547.9      2839.9       NaN    -62.532         NaN      Na
    EuropeanCallOption    7.5838    0.57026    0.022762    7.1435     67.763     -1.3962    153.6
    AmericanCallOption    8.8705     0.5845    0.019797    6.2597     76.808     -1.8677    200.6
    AsianCall             2.6123    0.35611    0.032053     12.95     42.238    -0.80342     64.3
```

See Also

`fininstrument` | `finmodel` | `finpricer`

Related Examples

- “Price Portfolio of Bond and Bond Option Instruments” on page 2-172

More About

- “Choose Instruments, Models, and Pricers” on page 1-53
- “Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers” on page 1-70
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Workflow for Creating and Analyzing a ratecurve and parametercurve

Use `ratecurve` or `irbootstrap` to create a `ratecurve` object.

```
% Create a ratecurve
Settle = datetime(2019,9,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =

  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10×1 datetime]
               Rates: [10×1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Use `zerorates`, `forwardrates`, or `discountfactors` with the `ratecurve` object.

```
CurveSettle = datetime(2019,9,15);

% zerorates
outZeroRates = zerorates(myRC,CurveSettle+30:30:CurveSettle+720)

% forwardrates
outForwardRates = forwardrates(myRC,datetime(2019,12,15),datetime(2021,9,15),6,7)

% discountfactors
outDiscountFactors = discountfactors(myRC,CurveSettle+30:30:CurveSettle+720)
```

```
outZeroRates =

  Columns 1 through 14

    0.0052    0.0052    0.0052    0.0052    0.0052    0.0052    0.0052    0.0053    0.0053    0.0

  Columns 15 through 24

    0.0056    0.0057    0.0057    0.0058    0.0058    0.0059    0.0059    0.0060    0.0060    0.0


outForwardRates =

    0.0062


outDiscountFactors =

  Columns 1 through 14
```

```
    0.9996    0.9991    0.9987    0.9983    0.9979    0.9974    0.9970    0.9965    0.9961    0.9

  Columns 15 through 24

    0.9931    0.9926    0.9920    0.9915    0.9910    0.9904    0.9898    0.9893    0.9887    0.9
```

Use `parametercurve`, `fitNelsonSiegel`, or `fitSvensson` to create a `parametercurve` object.

```
% parametercurve
myPC = parametercurve('zero',datetime(2019,9,15),@(t) polyval([-0.0001 0.003 0.02],t),'Parameters
```

```
myPC =

  parametercurve with properties:

             Type: "zero"
           Settle: 15-Sep-2019
      Compounding: -1
            Basis: 0
    FunctionHandle: @(t)polyval([-0.0001,0.003,0.02],t)
       Parameters: [-1.0000e-04 0.0030 0.0200]
```

```
% fitNelsonSiegel
Settle = datetime(2017,9,15);
  Maturity = [datetime(2019,9,15);datetime(2021,9,15);...
      datetime(2023,9,15);datetime(2026,9,7);...
      datetime(2035,9,15);datetime(2047,9,15)];

  CleanPrice = [100.1;100.1;100.8;96.6;103.3;96.3];
  CouponRate = [0.0400;0.0425;0.0450;0.0400;0.0500;0.0425];

  nInst = numel(CouponRate);
Bonds(nInst,1) = fininstrument.FinInstrument;
for ii=1:nInst
    Bonds(ii) = fininstrument("FixedBond","Maturity",Maturity(ii),...
        "CouponRate",CouponRate(ii));
end

NSModel = fitNelsonSiegel(Settle,Bonds,CleanPrice)
```

```
NSModel =

  parametercurve with properties:

             Type: "zero"
           Settle: 15-Sep-2017
      Compounding: -1
            Basis: 0
    FunctionHandle: @(t)fitF(Params,t)
       Parameters: [1.2473e-05 0.0362 0.0903 16.4263]
```

```
% fitSvensson
Settle = datetime(2017,9,15);
  Maturity = [datetime(2019,9,15);datetime(2021,9,15);...
      datetime(2023,9,15);datetime(2026,9,7);...
      datetime(2035,9,15);datetime(2047,9,15)];

  CleanPrice = [100.1;100.1;100.8;96.6;103.3;96.3];
```

```
  CouponRate = [0.0400;0.0425;0.0450;0.0400;0.0500;0.0425];

  nInst = numel(CouponRate);
Bonds(nInst,1) = fininstrument.FinInstrument;
for ii=1:nInst
    Bonds(ii) = fininstrument("FixedBond","Maturity",Maturity(ii),...
        "CouponRate",CouponRate(ii));
end

SvenModel = fitSvensson(Settle,Bonds,CleanPrice)

SvenModel =

  parametercurve with properties:

             Type: "zero"
           Settle: 15-Sep-2017
      Compounding: -1
            Basis: 0
    FunctionHandle: @(t)fitF(Params,t)
        Parameters: [2.2821 -41.8873 41.4090 -5.9589 0.3255 3.2356]
```

Use `zerorates`, `discountfactors`, `forwardrates` with the `myPC` `parametercurve` object.

```
% zerorates
CurveSettle = datetime('15-Sep-2019');
outZeroRates = zerorates(myPC,CurveSettle+30:30:CurveSettle+720)

% discountfactors
CurveSettle = datetime('15-Sep-2019');
outDiscountFactors = discountfactors(myPC,CurveSettle+30:30:CurveSettle+720)

% forwardrates
outForwardRates = forwardrates(myPC,datetime(2019,9,15),datetime(2020,9,15),6,7)

outZeroRates =

  Columns 1 through 14

    0.0202    0.0205    0.0207    0.0210    0.0212    0.0215    0.0217    0.0219    0.0222    0.0

  Columns 15 through 24

    0.0235    0.0238    0.0240    0.0242    0.0244    0.0246    0.0249    0.0251    0.0253    0.0


outDiscountFactors =

  Columns 1 through 14

    0.9983    0.9966    0.9949    0.9931    0.9913    0.9895    0.9876    0.9857    0.9838    0.9

  Columns 15 through 24

    0.9715    0.9693    0.9671    0.9649    0.9627    0.9604    0.9581    0.9558    0.9534    0.9


outForwardRates =
```

```
    0.0229
```

Convert RateSpec to a ratecurve Object

You can create a `RateSpec` using `intenvset` or `toRateSpec` from an `IRDataCurve` object. Then, you can covert this previously created `RateSpec` to a `ratecurve` object.

```
% Assume there is a RateSpec
Settle = datetime('01-Oct-2019');
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Basis = 1;
RateSpec = intenvset('StartDates', Settle, 'EndDates', ZeroDates, ...
    'Rates', ZeroRates, 'Basis', Basis)

RateSpec =

  struct with fields:

          FinObj: 'RateSpec'
     Compounding: 2
            Disc: [10×1 double]
           Rates: [10×1 double]
       EndTimes: [10×1 double]
      StartTimes: [10×1 double]
        EndDates: [10×1 double]
      StartDates: 737699
    ValuationDate: 737699
           Basis: 1
     EndMonthRule: 1

% Convert the RateSpec to a ratecurve
myRC = ratecurve("zero",RateSpec.ValuationDate,RateSpec.EndDates,RateSpec.Rates,...
"Compounding",RateSpec.Compounding,"Basis",RateSpec.Basis)

myRC =

  ratecurve with properties:

                Type: "zero"
         Compounding: 2
               Basis: 1
               Dates: [10×1 datetime]
               Rates: [10×1 double]
              Settle: 01-Oct-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"

% Check the discount factors
OldDF = intenvget(intenvset(RateSpec,'EndDates',datetime('01-Oct-2024')),'Disc')
NewDF = discountfactors(myRC,datetime('01-Oct-2024'))

OldDF =

    0.9424
```

```
NewDF =

    0.9424
```

In this case, the `RateSpec` and `ratecurve` are identical. This may not always be the case because `yearfrac` is used to compute times in `ratecurve` while `date2time` is used in computing a `RateSpec`. For more information, see “Difference Between yearfrac and date2time”.

See Also

`fininstrument` | `finmodel` | `finpricer`

More About

- “Choose Instruments, Models, and Pricers” on page 1-53
- “Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers” on page 1-70
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Workflow for Creating and Analyzing a defprobcurve

You can use `defprobcurve` or `defprobstrip` to create a `defprobcurve` object.

```
% defprobcurve
Settle = datetime(2019,9,20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities)
```

```
DefaultProbCurve =

  defprobcurve with properties:

                 Settle: 20-Sep-2019
                  Basis: 2
                  Dates: [10×1 datetime]
    DefaultProbabilities: [10×1 double]
```

You can then use the `defprobcurve` object with `survprobshazardrates`.

```
% hazardrates
hazardrates(DefaultProbCurve)
```

```
ans =

    0.0099
    0.0039
    0.0030
    0.0050
    0.0111
    0.0142
    0.0194
    0.0057
    0.0064
    0.0060
```

```
% survprobs
Settle = datetime(2019,9,20);
SurvProbTimes = [calmonths([6 12 18])];
SurvProbDates = Settle + SurvProbTimes;
outSurvProb = survprobs(DefaultProbCurve, SurvProbDates)
```

```
outSurvProb =

    0.9950
    0.9930
    0.9915
```

See Also

`fininstrument` | `finmodel` | `finpricer`

More About

- “Choose Instruments, Models, and Pricers” on page 1-53
- “Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers” on page 1-70

- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Choose Instruments, Models, and Pricers

The object-based framework supports a workflow for creating instruments, models, and pricer objects to price financial instruments. Using these objects, you can price interest-rate instruments; inflation instruments; equity, commodity, or FX instruments; or credit derivative instruments.

Interest-Rate Instruments with Associated Models and Pricers

The following table lists the interest-rate instrument objects with models and pricers.

Interest-Rate Instrument Type	Available Models	Available Pricers
`Cap`	• `HullWhite` • `BlackKarasinski` • `BlackDermanToy` • `Black` • `Normal` • `BraceGatarekMusiela` • `SABRBraceGatarekMusiela` • `LinearGaussian2F`	• `HullWhite` for `HullWhite` model • `Black` for `Black` model • `Normal` for `Normal` model • `IRTree` for `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` models • `IRMonteCarlo` for `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models
`Floor`	• `HullWhite` • `BlackKarasinski` • `BlackDermanToy` • `Black` • `Normal` • `BraceGatarekMusiela` • `SABRBraceGatarekMusiela` • `LinearGaussian2F`	• `HullWhite` for `HullWhite` model • `Black` for `Black` model • `Normal` for `Normal` model • `IRTree` for `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` models • `IRMonteCarlo` for `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models

Interest-Rate Instrument Type	Available Models	Available Pricers
`Swaption`	• `HullWhite` • `BlackKarasinski` • `BlackDermanToy` • `Black` • `SABR` • `Normal` • `LinearGaussian2F`	• `HullWhite` for `HullWhite` model • `Black` for `Black` model • `SABR` for `SABR` model • `Normal` for `Normal` model • `IRTree` for `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` models • `IRMonteCarlo` for `HullWhite`, `BlackKarasinski`, or `LinearGaussian2F` models
`FixedBondOption`	• `HullWhite` • `BlackKarasinski` • `BlackDermanToy` • `BraceGatarekMusiela` • `SABRBraceGatarekMusiela` • `LinearGaussian2F`	• `IRTree` for `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` models • `IRMonteCarlo` for `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models
`OptionEmbeddedFixedBond`	• `HullWhite` • `BlackKarasinski` • `BlackDermanToy` • `BraceGatarekMusiela` • `SABRBraceGatarekMusiela` • `LinearGaussian2F`	• `IRTree` for `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` models • `IRMonteCarlo` for `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models
`OptionEmbeddedFloatBond`	• `HullWhite` • `BlackKarasinski` • `BlackDermanToy` • `BraceGatarekMusiela` • `SABRBraceGatarekMusiela` • `LinearGaussian2F`	• `IRTree` for `HullWhite`, `BlackKarasinski`, `BlackDermanToy` models • `IRMonteCarlo` for `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models

Interest-Rate Instrument Type	Available Models	Available Pricers
Swap	• `ratecurve` object • `HullWhite` • `BlackKarasinski` • `BlackDermanToy` • `LinearGaussian2F`	• `Discount` • `IRTree` for `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` models • `IRMonteCarlo` for `HullWhite`, `BlackKarasinski`, or `LinearGaussian2F` models

`FixedBond`	• Use a `ratecurve` object for a `Discount` pricer.
	• Use one of the following models:
	  • `HullWhite`
	  • `BlackDermanToy`
	  • `BlackKarasinski`
	  • `BraceGatarekMusiela`
	  • `SABRBraceGatarekMusiela`
	  • `LinearGaussian2F`
		• `Discount`
		• `IRTree` for `HullWhite`, `BlackDermanToy`, or `BlackKarasinski` models
		• `IRMonteCarlo` for `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models

`FloatBond`	• Use a object for a `Discount` pricer.
	• Use one of the following models:
	  • `HullWhite`
	  • `BlackDermanToy`
	  • `BlackKarasinski`
	  • `BraceGatarekMusiela`
	  • `SABRBraceGatarekMusiela`
	  • `LinearGaussian2F`
		• `Discount`
		• `IRTree` for `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` models
		• `IRMonteCarlo` for `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models

Interest-Rate Instrument Type	Available Models	Available Pricers

`FloatBondOption`	• Use a `ratecurve` object for a `Discount` pricer.
	• Use one of the following models:
	  • `HullWhite`
	  • `BlackDermanToy`
	  • `BlackKarasinski`
	  • `BraceGatarekMusiela`
	  • `SABRBraceGatarekMusiela`
	  • `LinearGaussian2F`
		• `Discount`
		• `IRTree` for `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` models
		• `IRMonteCarlo` for `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models

`ConvertibleBond`	• `BlackScholes`	• `FiniteDifference` for `BlackScholes` model
`Deposit`	Use a `ratecurve` object.	• `Discount`
`FRA`	Use a `ratecurve` object.	• `Discount`
`OvernightIndexedSwap`	Use a `ratecurve` object.	• `Discount` • `irbootstrap` for curve construction
`STIRFuture`	Use a `ratecurve` object.	• `Discount` • `irbootstrap` for curve construction
`OISFuture`	Use a `ratecurve` object.	• `Discount` • `irbootstrap` for curve construction
`BondFuture`	Use a `ratecurve` object.	• `Future`

Equity, Commodity, FX, and Energy Instruments with Associated Models and Pricers

The following table lists the equity, commodity, FX , and energy instrument objects with models and pricers.

Equity, Commodity, FX Instrument Type	Available Models	Available Pricers
`Asian`	• `BlackScholes` • `Heston` • `Merton` • `Bates`	• `Levy` for `BlackScholes` model • `AssetTree` for a Cox-Ross-Rubinstein (CRR), equal-probability (EQP), Leisen-Reimer (LR), or Standard Trinomial (ST) lattice tree using a `BlackScholes` model • `KemnaVorst` for `BlackScholes` model • `TurnbullWakeman` or `BlackScholes` model • `AssetMonteCarlo` for `BlackScholes`, `Heston`, `Merton`, `Bates` models
`Barrier`	• `BlackScholes` • `Heston` • `Merton` • `Bates`	• `BlackScholes` for `BlackScholes` model • `AssetTree` for a Cox-Ross-Rubinstein (CRR), equal-probability (EQP), Leisen-Reimer (LR), or Standard Trinomial (ST) lattice tree using a `BlackScholes` model • `VannaVolga` for `BlackScholes` model • `FiniteDifference` for `BlackScholes`, `Heston`, `Merton`, `Bates` models • `AssetMonteCarlo` for `BlackScholes`, `Heston`, `Merton`, `Bates` models
`DoubleBarrier`	• `BlackScholes` • `Heston` • `Merton` • `Bates`	• `IkedaKunitomo` for `BlackScholes` model • `VannaVolga` for `BlackScholes` model • `FiniteDifference` for `BlackScholes`, `Heston`,,`Merton`, or `Bates` model • `AssetMonteCarlo` for `BlackScholes`, `Heston`, `Merton`, `Bates` models

Equity, Commodity, FX Instrument Type	Available Models	Available Pricers
Lookback	• BlackScholes • Heston • Merton • Bates	• ConzeViswanathan for BlackScholes model • AssetTree for a Cox-Ross-Rubinstein (CRR), equal-probability (EQP), Leisen-Reimer (LR), or Standard Trinomial (ST) lattice tree using a BlackScholes model • GoldmanSosinGatto for BlackScholes model • AssetMonteCarlo for BlackScholes, Heston, Merton, Bates models
PartialLookback	• BlackScholes • Heston • Merton • Bates	• HeynenKat for BlackScholes model • AssetMonteCarlo for BlackScholes, Heston, Merton, Bates models
Spread	• BlackScholes • Bachelier	For BlackScholes model: • Kirk • BjerksundStensland • AssetMonteCarlo For Bachelier model: • AssetMonteCarlo
VarianceSwap	• ratecurve object • Heston	For ratecurve object: • ReplicatingVarianceSwap For Heston model: • Heston

Equity, Commodity, FX Instrument Type	Available Models	Available Pricers
`Vanilla`	• `BlackScholes` • `Bachelier` • `Heston` • `Merton` • `Bates` • `Dupire`	For `BlackScholes` model: • `BlackScholes` • `VannaVolga` • `BjerksundStensland` • `RollGeskeWhaley` • `FiniteDifference` • `AssetMonteCarlo` • `AssetTree` for a Cox-Ross-Rubinstein (CRR), equal-probability (EQP), Leisen-Reimer (LR), or Standard Trinomial (ST) lattice tree For `Heston` model: • `FFT` • `FiniteDifference` • `NumericalIntegration` • `AssetMonteCarlo` For `Merton` model: • `FFT` • `FiniteDifference` • `NumericalIntegration` • `AssetMonteCarlo` For `Bates` model: • `FFT` • `FiniteDifference` • `NumericalIntegration` • `AssetMonteCarlo` For `Dupire` model: • `FiniteDifference` For `Bachelier` model: • `AssetMonteCarlo`

Equity, Commodity, FX Instrument Type	Available Models	Available Pricers
Touch	• BlackScholes • Heston • Merton • Bates	• BlackScholes for BlackScholes model • VannaVolga for BlackScholes model • AssetMonteCarlo for Bachelier model
DoubleTouch	• BlackScholes • Heston • Merton • Bates	• BlackScholes for BlackScholes model • VannaVolga for BlackScholes model • AssetMonteCarlo for BlackScholes, Heston, Merton, or Bates models
Cliquet	• BlackScholes • Heston • Merton • Bates	• Rubinstein for BlackScholes model • AssetMonteCarlo for BlackScholes, Heston, Merton, or Bates models
Binary	• BlackScholes • Bachelier • Heston • Merton • Bates	• BlackScholes for BlackScholes model • AssetMonteCarlo for BlackScholes, Heston, Merton, Bachelier, or Bates models
CommodityFuture	Use a ratecurve object.	• Future
FXFuture	Use a ratecurve object.	• Future
EquityIndexFuture	Use a ratecurve object.	• Future
ConvertibleBond	• BlackScholes	• FiniteDifference for BlackScholes model

Inflation Instruments with Associated Models and Pricers

The following table lists the inflation instrument objects with models and pricers.

Inflation Instrument Type	Available Models	Available Pricers
InflationBond	Use an inflationcurve object and a ratecurve object.	• Inflation
YearYearInflationSwap	Use an inflationcurve object and a ratecurve object.	• Inflation
ZeroCouponInflationSwap	Use an inflationcurve object and a ratecurve object.	• Inflation

Credit Derivative Instruments with Associated Models and Pricers

The following table lists the credit derivative instrument objects with models and pricers.

Credit Derivative Instrument Type	Available Models	Available Pricers
`CDS`	Use a `defprobcurve` object and a `ratecurve` object.	• `Credit`
`CDSOption`	`CDSBlack`	• `CDSBlack`

See Also

`fininstrument` | `finmodel` | `finpricer`

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Supported Exercise Styles” on page 1-62
- “Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers” on page 1-70

Supported Exercise Styles

The following table lists the interest-rate instrument objects with their associated models and pricers and supported `Exercise` styles.

Interest-Rate Instrument Type	Exercise Styles
`Swaption`	• `"European"` — `HullWhite` pricer for `HullWhite` model • `"European"` — `Black` pricer for `Black` model • `"European"` — `SABR` pricer for `SABR` model • `"European"` — `Normal` pricer for `Normal` model • `"European"` or `"American"` — `IRTree` pricer for `HullWhite` or `BlackKarasinski` models • `"European"` or `"American"` — `IRMonteCarlo` pricer for `HullWhite`, or `LinearGaussian2F` models
`FixedBondOption`	• `"European"`, `"American"`, or `"Bermudan"` — `IRTree` pricer for `HullWhite` or `BlackKarasinski` models • `"European"`, `"American"`, or `"Bermudan"` — `IRMonteCarlo` pricer for `HullWhite`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models
`FloatBondOption`	• `"European"`, `"American"`, or `"Bermudan"` — `IRTree` pricer for `HullWhite` or `BlackKarasinski` models • `"European"`, `"American"`, or `"Bermudan"` — `IRMonteCarlo` pricer for `HullWhite`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models
`OptionEmbeddedFixedBond`	• `"European"`, `"American"`, or `"Bermudan"` — `IRTree` pricer for `HullWhite` or `BlackKarasinski` models • `"European"`, `"American"`, or `"Bermudan"` — `IRMonteCarlo` pricer for `HullWhite`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models

Interest-Rate Instrument Type	Exercise Styles
`OptionEmbeddedFloatBond`	• `"European"`, `"American"`, or `"Bermudan"` — `IRTree` pricer for `HullWhite` or `BlackKarasinski` models • `"European"`, `"American"`, or `"Bermudan"` — `IRMonteCarlo` pricer for `HullWhite`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` models
`ConvertibleBond`	For `BlackScholes` model: • `"European"`, `"American"`, or `"Bermudan"` — `FiniteDifference` pricer

The following table lists the equity, commodity, or FX instrument objects with their associated models and pricers and the supported `Exercise` styles.

Equity, Commodity, FX Instrument Type	Exercise Styles
`Asian`	For `BlackScholes` model: • `"European"` — `Levy` pricer • `"European"` — `KemnaVorst` pricer • `"European"` — `TurnbullWakeman` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer • `"European"` or `"American"` — `AssetTree` pricer For `Bates` model: • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Merton` model: • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Heston` model: • `"European"` or `"American"` — `AssetMonteCarlo` pricer

Equity, Commodity, FX Instrument Type	Exercise Styles
`Barrier`	For `BlackScholes` model: • `"European"` — `BlackScholes` pricer • `"European"` or `"American"` — `FiniteDifference` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer • `"European"` or `"American"` — `AssetTree` pricer For `Bates` model: • `"European"` or `"American"` — `FiniteDifference` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Merton` model: • `"European"` or `"American"` — `FiniteDifference` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Heston` model: • `"European"` or `"American"` — `FiniteDifference` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer

Equity, Commodity, FX Instrument Type	Exercise Styles
`DoubleBarrier`	For `BlackScholes` model: • `"European"` — `IkedaKunitomo` pricer • `"European"` or `"American"` — `FiniteDifference` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Bates` model: • `"European"` or `"American"` — `FiniteDifference` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Merton` model: • `"European"` or `"American"` — `FiniteDifference` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Heston` model: • `"European"` or `"American"` — `FiniteDifference` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer

Equity, Commodity, FX Instrument Type	Exercise Styles
`Lookback`	For `BlackScholes` model: • `"European"` — `ConzeViswanathan` pricer • `"European"` — `GoldmanSosinGatto` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer • `"European"` or `"American"` — `AssetTree` pricer For `Bates` model: • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Merton` model: • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Heston` model: • `"European"` or `"American"` — `AssetMonteCarlo` pricer
`PartialLookback`	For `BlackScholes` model: • `"European"` — `HeynenKat` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Bates` model: • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Merton` model: • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Heston` model: • `"European"` or `"American"` — `AssetMonteCarlo` pricer

Equity, Commodity, FX Instrument Type	Exercise Styles
`Spread`	For `BlackScholes` model: • `"European"` — `Kirk` pricer • `"European"` — `BjerksundStensland` pricer • `"European"` or `"American"` — `AssetMonteCarlo` pricer For `Bachelier` model: • `"European"` or `"American"` — `AssetMonteCarlo` pricer

Equity, Commodity, FX Instrument Type	Exercise Styles
`Vanilla`	For `BlackScholes` model: • `"European"` — `BlackScholes` pricer • `"American"` — `BjerksundStensland` pricer • `"American"` — `RollGeskeWhaley` pricer • `"European"`, `"American"`, or `"Bermudan"` —`FiniteDifference` pricer • `"European"`, `"American"`, or `"Bermudan"` — `AssetMonteCarlo` pricer • `"European"`, `"American"`, or `"Bermudan"` — `AssetTree` pricer For `Heston` model: • `"European"` — `FFT` pricer • `"European"`, `"American"`, or `"Bermudan"` — `FiniteDifference` pricer • `"European"` — `NumericalIntegration` pricer • `"European"`, `"American"`, or `"Bermudan"` — `AssetMonteCarlo` pricer For `Merton` model: • `"European"` — `FFT` pricer • `"European"`, `"American"`, or `"Bermudan"` — `FiniteDifference` pricer • `"European"` — `NumericalIntegration` pricer • `"European"`, `"American"`, or `"Bermudan"` — `AssetMonteCarlo` pricer For `Bates` model: • `"European"` — `FFT` pricer • `"European"`, `"American"`, or `"Bermudan"` — `FiniteDifference` pricer • `"European"` — `NumericalIntegration` pricer • `"European"`, `"American"`, or `"Bermudan"` — `AssetMonteCarlo` pricer For `Dupire` model: • `"European"`, `"American"`, or `"Bermudan"` — `FiniteDifference` pricer For `Bachelier` model:

Equity, Commodity, FX Instrument Type	Exercise Styles
	• `"European"`, `"American"`, `"Bermudan"` — `AssetMonteCarlo` pricer
`Binary`	For `BlackScholes` model: • `"European"` — `BlackScholes` pricer For `Heston`, `Bates`, and `Merton` models: • `"European"` — `AssetMonteCarlo` pricer For `Bachelier` model: • `"European"` — `AssetMonteCarlo` pricer
`Cliquet`	For `BlackScholes` model: • `"European"` — `Rubinstein` pricer For `BlackScholes`, `Heston`, `Bates`, and `Merton` models: • `"European"` — `AssetMonteCarlo` pricer
`ConvertibleBond`	For `BlackScholes` model: • `"European"`, `"American"`, or `"Bermudan"` — `FiniteDifference` pricer

See Also

`fininstrument` | `finmodel` | `finpricer`

More About

- “Choose Instruments, Models, and Pricers” on page 1-53
- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers” on page 1-70

Mapping Financial Instruments Toolbox Functions to Object-Based Framework for Instruments, Models, and Pricers

Financial Instruments Toolbox allows you to use either a function-based framework or an alternative object-based framework to price financial instruments.

In the function-based framework, a typical workflow to price a bond with embedded options is as follows.

1 Create a `RateSpec` instrument using `intenvset`.

```
% Zero Data
Settle = datetime(2018,9,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = -1;
Basis = 1;

% Instrument parameters
Maturity = datetime(2024,9,15);
CouponRate = 0.035;
Strike = 100;
ExerciseDates = datetime(2022,9,15);
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
Period = 1;

% HW Parameters
Vol = 0.01;
Alpha = 0.1;
TreeDates = Settle + calyears(1:10);

RateSpec = intenvset('Compounding', Compounding,'StartDates', Settle,...
    'EndDates', ZeroDates,'Rates', ZeroRates,'Basis',Basis);
```

2 Create a Hull-White tree object using `hwvolspec`, `hwtimespec`, and `hwtree`.

```
HWVolSpec = hwvolspec(Settle, TreeDates, Vol,TreeDates, Alpha);
HWTimeSpec = hwtimespec(Settle, TreeDates, 1);
HWTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec);
OldPrice = optembndbyhw(HWTree,CouponRate,Settle,Maturity,'call',Strike,ExerciseDates,'Perioc
```

3 Price the bond with embedded options using an Hull-White interest-rate tree with `optembndbyhw`.

```
OldPrice = optembndbyhw(HWTree,CouponRate,Settle,Maturity,'call',Strike,ExerciseDates,'Perioc
```

```
OldPrice =

  109.4814
```

By contrast, in the Financial Instruments Toolbox object-based workflow, you price an instrument using instrument, model, and pricer objects:

1 Create an `OptionEmbeddedFixedBond` instrument using `OptionEmbeddedFixedBond`.

```
% Zero Data
Settle = datetime(2018,9,15);
```

```
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = -1;
Basis = 1;

% Instrument parameters
Maturity = datetime(2024,9,15);
CouponRate = 0.035;
Strike = 100;
ExerciseDates = datetime(2022,9,15);
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
Period = 1;

% HW Parameters
Vol = 0.01;
Alpha = 0.1;
TreeDates = Settle + calyears(1:10);

CallableBond = fininstrument("OptionEmbeddedFixedBond", "Maturity",Maturity,...
    'CouponRate',CouponRate,'Period',Period, ...
    'CallSchedule',CallSchedule,'Name',"CallableBond",'Basis',Basis);
```

2 Create a `ratecurve` object using `ratecurve`.

```
myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates,'Basis',Basis);
```

3 Create a `HullWhite` model object using `HullWhite`.

```
HWModel = finmodel("HullWhite","Alpha",Alpha,"Sigma",Vol);
```

4 Create an `IRTree` pricer object using `IRTree`.

```
HWPricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',myRC,'TreeDates',TreeDates');
```

5 Price the bond instrument using `price`.

```
NewPrice = price(HWPricer, CallableBond)
```

```
NewPrice =

  109.4814
```

Note The function-based and object-based workflows can return different instrument prices even if you use the same data. The difference is because the existing Financial Instruments Toolbox functions internally use `datetime` and the object-based framework use `yearfrac` for date handling.

For a mapping of function-based instrument pricing to the object-based instrument pricing, see:

- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84
- “Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94

- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

See Also

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53
- “Supported Exercise Styles” on page 1-62
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects

The following table lists the Financial Instruments Toolbox functions for interest-rate instruments mapped to the associated workflow using the object-based framework for instruments, models, and pricers.

Financial Instruments Toolbox Function	Object-Based Workflow
`capbyblk`	Create the following objects: 1 `Cap` instrument 2 `Black` model 3 `ratecurve` 4 `Black` pricer Compute the price of the `Cap` instrument using `price`.
`capbynormal`	Create the following objects: 1 `Cap` instrument 2 `Normal` model 3 `ratecurve` 4 `Normal` pricer Compute the price of the `Cap` instrument using `price`.
`capbybk`	Create the following objects: 1 `Cap` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Cap` instrument using `price`.
`capbybdt`	Create the following objects: 1 `Cap` instrument 2 `BlackDermanToy` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Cap` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`capbyhw`	Create the following objects: 1 `Cap` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Cap` instrument using `price`.
`capbylg2f`	Create the following objects: 1 `Cap` instrument 2 `LinearGaussian2F` model 3 `ratecurve` 4 `IRMonteCarlo` pricer Compute the price of the `Cap` instrument using `price`.
`floorbyblk`	Create the following objects: 1 `Floor` instrument 2 `Black` model 3 `ratecurve` 4 `Black` pricer Compute the price of the `Floor` instrument using `price`.
`floorbynormal`	Create the following objects: 1 `Floor` instrument 2 `Normal` model 3 `ratecurve` 4 `Normal` pricer Compute the price of the `Floor` instrument using `price`.
`floorbybk`	Create the following objects: 1 `Floor` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Floor` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`floorbybdt`	Create the following objects: 1 `Floor` instrument 2 `BlackDermanToy` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Floor` instrument using `price`.
`floorbyhw`	Create the following objects: 1 `Floor` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Floor` instrument using `price`.
`floorbylg2f`	Create the following objects: 1 `Floor` instrument 2 `LinearGaussian2F` model 3 `ratecurve` 4 `IRMonteCarlo` pricer Compute the price of the associated instrument using `price`.
`swaptionbyblk`	Create the following objects: 1 `Swaption` instrument 2 `Black` model 3 `ratecurve` 4 `Black` pricer Compute the price of the `Swaption` instrument using `price`.
`swaptionbynormal`	Create the following objects: 1 `Swaption` instrument 2 `Normal` model 3 `ratecurve` 4 `Normal` pricer Compute the price of the `Swaption` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`swaptionbybk`	Create the following objects: 1 `Swaption` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Swaption` instrument using `price`.
`swaptionbybdt`	Create the following objects: 1 `Swaption` instrument 2 `BlackDermanToy` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Swaption` instrument using `price`.
`swaptionbyhw`	Create the following objects: 1 `Swaption` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Swaption` instrument using `price`.
`swaptionbylg2f`	Create the following objects: 1 `Swaption` instrument 2 `LinearGaussian2F` model 3 `ratecurve` 4 `IRMonteCarlo` pricer Compute the price of the associated instrument using `price`.
`fixedbyzero`	Create the following objects: 1 `FixedBond` instrument 2 `ratecurve` 3 `Discount` pricer Compute the price of the `FixedBond` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`fixedbybk`	Create the following objects: 1 `FixedBond` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `FixedBond` instrument using `price`.
`fixedbybdt`	Create the following objects: 1 `FixedBond` instrument 2 `BlackDermanToy` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `FixedBond` instrument using `price`.
`fixedbyhw`	Create the following objects: 1 `FixedBond` instrument 2 `HullWhite` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `FixedBond` instrument using `price`.
`bondbyzero`	Create the following objects: 1 `FixedBond` instrument 2 `ratecurve` 3 `Discount` pricer Compute the price of the `FixedBond` instrument using `price`.
`tfutbyprice`	Create the following objects: 1 `BondFuture` instrument 2 `ratecurve` 3 `Future` pricer Compute the price of the `BondFuture` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`floatbyzero`	Create the following objects: 1 `FloatBond` instrument 2 `ratecurve` 3 `Discount` pricer Compute the price of the `FloatBond` instrument using `price`.
`floatbybk`	Create the following objects: 1 `FloatBond` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `FloatBond` instrument using `price`.
`floatbybdt`	Create the following objects: 1 `FloatBond` instrument 2 `BlackDermanToy` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `FloatBond` instrument using `price`.
`floatbyhw`	Create the following objects: 1 `FloatBond` instrument 2 `HullWhite` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `FloatBond` instrument using `price`.
`optbndbyhw`	Create the following objects: 1 `FixedBondOption` instrument 2 `HullWhite` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `FixedBondOption` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`optbndbybk`	Create the following objects: **1** `FixedBondOption` instrument **2** `BlackKarasinski` model **3** `ratecurve` **4** `IRTree` pricer Compute the price of the `FixedBondOption` instrument using `price`.
`optbndbybdt`	Create the following objects: **1** `FixedBondOption` instrument **2** `BlackDermanToy` model **3** `ratecurve` **4** `IRTree` pricer Compute the price of the `FixedBondOption` instrument using `price`.
`optfloatbybk`	Create the following objects: **1** `FloatBondOption` instrument **2** `BlackKarasinski` model **3** `ratecurve` **4** `IRTree` pricer Compute the price of the `FloatBondOption` instrument using `price`.
`optfloatbybdt`	Create the following objects: **1** `FloatBondOption` instrument **2** `BlackDermanToy` model **3** `ratecurve` **4** `IRTree` pricer Compute the price of the `FloatBondOption` instrument using `price`.
`optfloatbyhw`	Create the following objects: **1** `FloatBondOption` instrument **2** `BlackKarasinski` model **3** `ratecurve` **4** `IRTree` pricer Compute the price of the `FloatBondOption` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`optembndbyhw`	Create the following objects: 1 `OptionEmbeddedFixedBond` instrument 2 `HullWhite` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `OptionEmbeddedFixedBond` instrument using `price`.
`optembndbybk`	Create the following objects: 1 `OptionEmbeddedFixedBond` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `OptionEmbeddedFixedBond` instrument using `price`.
`optembndbybdt`	Create the following objects: 1 `OptionEmbeddedFixedBond` instrument 2 `BlackDermanToy` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `OptionEmbeddedFixedBond` instrument using `price`.
`optemfloatbybk`	Create the following objects: 1 `OptionEmbeddedFloatBond` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `OptionEmbeddedFloatBond` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`optembndbybdt`	Create the following objects: 1 `OptionEmbeddedFloatBond` instrument 2 `BlackDermanToy` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `OptionEmbeddedFloatBond` instrument using `price`.
`optemfloatbybk`	Create the following objects: 1 `OptionEmbeddedFloatBond` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `OptionEmbeddedFloatBond` instrument using `price`.
`optemfloatbyhw`	Create the following objects: 1 `OptionEmbeddedFloatBond` instrument 2 `HullWhite` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `OptionEmbeddedFloatBond` instrument using `price`.
`optemfloatbybdt`	Create the following objects: 1 `OptionEmbeddedFloatBond` instrument 2 `BlackDermanToy` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `OptionEmbeddedFloatBond` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`swapbyzero`	Create the following objects: 1 `Swap` instrument 2 `ratecurve` 3 `Discount` pricer Compute the price of the `Swap` instrument using `price`.
`swapbybk`	Create the following objects: 1 `Swap` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Swap` instrument using `price`.
`swapbybdt`	Create the following objects: 1 `Swap` instrument 2 `BlackDermanToy` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Swap` instrument using `price`.
`swapbyhw`	Create the following objects: 1 `Swap` instrument 2 `BlackKarasinski` model 3 `ratecurve` 4 `IRTree` pricer Compute the price of the `Swap` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`LiborMarketModel`	Create the following objects: **1** `Cap`, `Floor`, `FixedBond`, `FloatBond`, `FloatBondOption` `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument **2** `SABRBraceGatarekMusiela` or `BraceGatarekMusiela` model **3** `ratecurve` **4** `IRMonteCarlo` pricer Compute the price of the associated instrument using `price`.
`LinearGaussian2F`	Create the following objects: **1** `Cap`, `Floor`, `Swap`, `Swaption`, `FixedBond`, `FloatBond`, `FloatBondOption` `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument **2** `LinearGaussian2F` model **3** `ratecurve` **4** `IRMonteCarlo` pricer Compute the price of the associated instrument using `price`.

See Also

`fininstrument` | `finmodel` | `finpricer`

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84
- “Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects

The following table lists the Financial Instruments Toolbox functions for equity, FX, or commodity instruments mapped to the associated workflow using the object-based framework for instruments, models, and pricers.

Financial Instruments Toolbox Function	Object-Based Workflow
`asianbycrr`	Create the following objects: 1 `Asian` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Asian` instrument using `price`.
`asianbyeqp`	Create the following objects: 1 `Asian` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Asian` instrument using `price`.
`asianbykv` and `asiansensbykv`	Create the following objects: 1 `Asian` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `KemnaVorst` pricer Compute the price of the `Asian` instrument using `price`.
`asianbylevy` and `asiansensbylevy`	Create the following objects: 1 `Asian` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `Levy` pricer Compute the price of the `Asian` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`asianbytw` and `asiansensbytw`	Create the following objects: 1 `Asian` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `TurnbullWakeman` pricer Compute the price of the `Asian` instrument using `price`.
`asianbyls` and `asiansensbyls`	Create the following objects: 1 `Asian` instrument 2 `BlackScholes`, `Merton`, `Bates`, or `Heston` model 3 `ratecurve` 4 `AssetMonteCarlo` pricer Compute the price of the `Asian` instrument using `price`.
`asianbystt`	Create the following objects: 1 `Asian` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Asian` instrument using `price`.
`barrierbycrr`	Create the following objects: 1 `Barrier` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Barrier` instrument using `price`.
`barrierbyeqp`	Create the following objects: 1 `Barrier` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Barrier` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`barrierbybls` and `barriersensbybls`	Create the following objects: 1 `Barrier` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `BlackScholes` pricer Compute the price of the `Barrier` instrument using `price`.
`barrierbyfd` and `barriersensbyfd`	Create the following objects: 1 `Barrier` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `FiniteDifference` pricer Compute the price of the `Barrier` instrument using `price`.
`barrierbyls` and `barriersensbyls`	Create the following objects: 1 `Barrier` instrument 2 `BlackScholes`, `Merton`, `Bates`, or `Heston` model 3 `ratecurve` 4 `AssetMonteCarlo` pricer Compute the price of the `Barrier` instrument using `price`.
`barrierbystt`	Create the following objects: 1 `Barrier` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Barrier` instrument using `price`.
`dblbarrierbybls` and `dblbarriersensbybls`	Create the following objects: 1 `DoubleBarrier` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `BlackScholes` pricer Compute the price of the `DoubleBarrier` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`dblbarrierbyfd` and `dblbarriersensbyfd`	Create the following objects: 1 `DoubleBarrier` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `FiniteDifference` pricer Compute the price of the `DoubleBarrier` instrument using `price`.
`touchbybls` and `touchsensbybls`	Create the following objects: 1 `Touch` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `BlackScholes` pricer Compute the price of the `Touch` instrument using `price`.
`dbltouchbybls` and `dbltouchsensbybls`	Create the following objects: 1 `DoubleTouch` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `BlackScholes` pricer Compute the price of the `DoubleTouch` instrument using `price`.
`cashbybls` and `cashsensbybls`	Create the following objects: 1 `Binary` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `BlackScholes` pricer
`assetbybls` and `assetsensbybls`	Create the following objects: 1 `Binary` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `BlackScholes` pricer

Financial Instruments Toolbox Function	Object-Based Workflow
`lookbackbycrr`	Create the following objects: 1 `Lookback` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Lookback` instrument using `price`.
`lookbackbyeqp`	Create the following objects: 1 `Lookback` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Lookback` instrument using `price`.
`lookbackbycvgsg` and `lookbacksensbycvgsg`	Create the following objects: 1 `Lookback` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `ConzeViswanathan` or `GoldmanSosinGatto` pricer Compute the price of the `Lookback` instrument using `price`.
`lookbackbyls` and `lookbacksensbyls`	Create the following objects: 1 `Lookback` instrument 2 `BlackScholes`, `Merton`, `Bates`, or `Heston` model 3 `ratecurve` 4 `AssetMonteCarlo` pricer Compute the price of the `Lookback` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`lookbackbystt`	Create the following objects: 1 `Lookback` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Lookback` instrument using `price`.
`spreadbykirk` and `spreadsensbykirk`	Create the following objects: 1 `Spread` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `Kirk` pricer Compute the price of the `Spread` instrument using `price`.
`spreadbybjs` and `spreadsensbybjs`	Create the following objects: 1 `Spread` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `BjerksundStensland` pricer Compute the price of the `Spread` instrument using `price`.
`spreadbyls` and `spreadsensbyls`	Create the following objects: 1 `Spread` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetMonteCarlo` pricer Compute the price of the `Spread` instrument using `price`.
`optstockbycrr`	Create the following objects: 1 `Vanilla` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Vanilla` instrument using `price`

Financial Instruments Toolbox Function	Object-Based Workflow
`optstockbyeqp`	Create the following objects: 1 `Vanilla` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Vanilla` instrument using `price`
`optstockbylr` and `optstocksensbylr`	Create the following objects: 1 `Vanilla` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Vanilla` instrument using `price`
`optstockbybls` and `optstocksensbybls`	Create the following objects: 1 `Vanilla` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `BlackScholes` pricer Compute the price of the `Vanilla` instrument using `price`.
`optstockbybjs` and `optstocksensbybjs`	Create the following objects: 1 `Vanilla` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `BjerksundStensland` pricer Compute the price of the `Vanilla` instrument using `price`.
`optstockbyrgw` and `optstocksensbyrgw`	Create the following objects: 1 `Vanilla` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `RollGeskeWhaley` pricer Compute the price of the `Vanilla` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`optstockbyfd` and `optstocksensbyfd`	Create the following objects: 1 `Vanilla` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `FiniteDifference` pricer Compute the price of the `Vanilla` instrument using `price`.
`optstockbyls` and `optstocksensbyls`	Create the following objects: 1 `Vanilla` instrument 2 `BlackScholes`, `Merton`, `Bates`, or `Heston` model 3 `ratecurve` 4 `AssetMonteCarlo` pricer Compute the price of the `Vanilla` instrument using `price`.
`optstockbystt`	Create the following objects: 1 `Vanilla` instrument 2 `BlackScholes` model 3 `ratecurve` 4 `AssetTree` pricer Compute the price of the `Vanilla` instrument using `price`.
`optByHestonFFT` and `optSensByHestonFFT`	Create the following objects: 1 `Vanilla` instrument 2 `Heston` model 3 `ratecurve` 4 `FFT` pricer Compute the price of the `Vanilla` instrument using `price`.
`optByHestonFD` and `optSensByHestonFD`	Create the following objects: 1 `Vanilla` instrument 2 `Heston` model 3 `ratecurve` 4 `FiniteDifference` pricer Compute the price of the `Vanilla` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`optByHestonNI` and `optSensByHestonNI`	Create the following objects: 1 `Vanilla` instrument 2 `Heston` model 3 `ratecurve` 4 `NumericalIntegration` pricer Compute the price of the `Vanilla` instrument using `price`.
`optByBatesFFT` and `optSensByBatesFFT`	Create the following objects: 1 `Vanilla` instrument 2 `Bates` model 3 `ratecurve` 4 `FFT` pricer Compute the price of the `Vanilla` instrument using `price`.
`optByBatesFD` and `optSensByBatesFD`	Create the following objects: 1 `Vanilla` instrument 2 `Bates` model 3 `ratecurve` 4 `FiniteDifference` pricer Compute the price of the `Vanilla` instrument using `price`.
`optByBatesNI` and `optSensByBatesNI`	Create the following objects: 1 `Vanilla` instrument 2 `Bates` model 3 `ratecurve` 4 `NumericalIntegration` pricer Compute the price of the `Vanilla` instrument using `price`.
`optByMertonFFT` and `optSensByMertonFFT`	Create the following objects: 1 `Vanilla` instrument 2 `Merton` model 3 `ratecurve` 4 `FFT` pricer Compute the price of the `Vanilla` instrument using `price`.

Financial Instruments Toolbox Function	Object-Based Workflow
`optByMertonFD` and `optSensByMertonFD`	Create the following objects: 1 `Vanilla` instrument 2 `Merton` model 3 `ratecurve` 4 `FiniteDifference` pricer Compute the price of the `Vanilla` instrument using `price`.
`optByMertonNI` and `optSensByMertonNI`	Create the following objects: 1 `Vanilla` instrument 2 `Merton` model 3 `ratecurve` 4 `NumericalIntegration` pricer Compute the price of the `Vanilla` instrument using `price`.
`optByLocalVolFD` and `optSensByLocalVolFD`	Create the following objects: 1 `Vanilla` instrument 2 `Dupire` model 3 `ratecurve` 4 `FiniteDifference` pricer Compute the price of the `Vanilla` instrument using `price`.

See Also

`fininstrument` | `finmodel` | `finpricer`

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73
- “Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects

The following table lists the Financial Instruments Toolbox functions for credit derivative instruments mapped to the associated workflow using the object-based framework for instruments, models, and pricers.

Financial Instruments Toolbox Function	Object-Based Workflow
`cdsprice`	Create the following objects: 1 `CDS` instrument 2 `defprobcurve` 3 `ratecurve` 4 `Credit` pricer Compute the price of the `CDS` instrument using `price`.
`cdsoptprice`	Create the following objects: 1 `CDSOption` instrument 2 `CDSBlack` model 3 `ratecurve` 4 `CDSBlack` pricer Compute the price of the `CDSOption` instrument using `price`.

See Also

`fininstrument` | `finmodel` | `finpricer`

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework

Financial Instruments Toolbox allows you to use either a function-based framework or an alternative object-based framework to create and analyze financial curves.

In the function-based framework, a typical workflow to create an interest-rate curve uses `intenvset` or `IRDataCurve`.

```
CurveSettle = datetime(2016,3,2);
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = datemnth(CurveSettle,12*[1 2 3 5 7 10 20 30]);
irdc = IRDataCurve('Zero',CurveSettle,Dates,Data)
```

```
irdc =

             Type: Zero
           Settle: 736391 (02-Mar-2016)
      Compounding: 2
            Basis: 0 (actual/actual)
     InterpMethod: linear
            Dates: [8x1 double]
             Data: [8x1 double]
```

By contrast, in the Financial Instruments Toolbox object-based workflow, you create a `ratecurve` object:

```
Settle = datetime(2017,9,15);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =

  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10×1 datetime]
               Rates: [10×1 double]
              Settle: 15-Sep-2017
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Note The function-based and object-based workflows can return different prices even if you use the same data. This is because the existing Financial Instruments Toolbox curve functions use `date2time` and the object-based framework use `yearfrac` for date handling.

The following table lists the Financial Instruments Toolbox curve functions mapped to the associated object-based framework.

Financial Instruments Toolbox Curve Function	Object-Based Framework
`IRDataCurve`	`ratecurve`
`getForwardRates`	`forwardrates`
`getZeroRates`	`zerorates`
`getDiscountFactors`	`discountfactors`
`bootstrap`	`irbootstrap`
`IRFunctionCurve`	`parametercurve`
`getForwardRates`	`forwardrates`
`getZeroRates`	`zerorates`
`getDiscountFactors`	`discountfactors`
`fitNelsonSiegel`	`fitNelsonSiegel`
`fitSvensson`	`fitSvensson`
`cdsbootstrap`	`defprobstrip`
Not supported	`defprobcurve`
Not supported	`survprobs`
Not supported	`hazardrates`
Not supported	`STIRFuture` using `irbootstrap` to create `ratecurve` object
Not supported	`OISFuture` using `irbootstrap` to create `ratecurve` object
Not supported	`OvernightIndexedSwap` using `irbootstrap` to create `ratecurve` object

See Also

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53
- “Convert RateSpec to a ratecurve Object” on page 1-49
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84
- “Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94

Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers

This example shows how to compare European `Vanilla` instrument call option prices using a `BlackScholes` model and different pricing methods. The pricing methods for this comparison are the Cox-Ross-Rubinstein, Leisen-Reimer, finite difference, and the Black-Scholes analytical formula.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,01,01);
Maturity = datetime(2022,01,01);
Rate = 0.0111;
Compounding = -1;
ZeroCurve = ratecurve('zero',Settle,Maturity,Rate,'Compounding',Compounding);
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
Volatility = .35;
BSModel = finmodel("BlackScholes",'Volatility',Volatility);
```

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
ExerciseDates = datetime(2019,09,01);
Strike = 30;
OptionType = 'call';

EuropeanCallOption = fininstrument("Vanilla",'ExerciseDate',ExerciseDates,'Strike',Strike,...
    'OptionType',OptionType,'Name',"vanilla_call_option");
```

Create `Analytic`, `AssetTree`, and `FiniteDifference` Pricer Objects

Create two scenarios for the Vanilla option. In the first scenario, the option is out of the money (OTM). In the second scenario the option is at the money (ATM).

```
% Define the number of levels of the tree for AssetTree pricer
NumPeriods = 55;
```

Calculate Vanilla Option Price for OTM Option

Use `finpricer` to create an `BlackScholes`, `AssetTree`, and `FiniteDifference` pricer objects for the OTM option and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
SpotPriceOTM = 25;

% Analytic Pricer
AnalyticPricerOTM = finpricer('Analytic', 'Model', BSModel, 'SpotPrice', SpotPriceOTM, 'DiscountC
PriceBLSOTM = price(AnalyticPricerOTM, EuropeanCallOption);

% AssetTree Pricer
```

```
CRRPricerOTM = finpricer("AssetTree",'DiscountCurve',ZeroCurve,'Model',BSModel, 'SpotPrice',SpotP
                         'PricingMethod',"CoxRossRubinstein",'NumPeriods', NumPeriods, 'Maturity
PriceCRROTM = price(CRRPricerOTM, EuropeanCallOption);


LRPricerOTM = finpricer("AssetTree",'DiscountCurve',ZeroCurve,'Model',BSModel, 'SpotPrice',SpotPr
                        'PricingMethod',"LeisenReimer",'NumPeriods', NumPeriods, 'Maturity', Exer
PriceLROTM = price(LRPricerOTM, EuropeanCallOption);

% FiniteDifference Pricer
FDPricerOTM = finpricer('FiniteDifference', 'Model', BSModel, 'SpotPrice', SpotPriceOTM, 'Discou
PriceFDOTM = price(FDPricerOTM, EuropeanCallOption);
```

Calculate Vanilla Option Price for ATM Option

Use `finpricer` to create an `BlackScholes`, `AssetTree`, and `FiniteDifference` pricer objects for the ATM option and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
SpotPriceATM = 30;

% Analytic Pricer
AnalyticPricerATM = finpricer('Analytic', 'Model', BSModel, 'SpotPrice', SpotPriceATM, 'DiscountC
PriceBLSATM = price(AnalyticPricerATM, EuropeanCallOption);

% AsetTree Pricer
CRRPricerATM = finpricer("AssetTree",'DiscountCurve',ZeroCurve,'Model',BSModel, 'SpotPrice',SpotP
                         'PricingMethod',"CoxRossRubinstein",'NumPeriods', NumPeriods, 'Maturity
PriceCRRATM = price(CRRPricerATM, EuropeanCallOption);


LRPricerATM = finpricer("AssetTree",'DiscountCurve',ZeroCurve,'Model',BSModel, 'SpotPrice',SpotPr
                        'PricingMethod',"LeisenReimer",'NumPeriods', NumPeriods, 'Maturity', Exer
PriceLRATM = price(LRPricerATM, EuropeanCallOption);

% FiniteDifference Pricer
FDPricerATM = finpricer('FiniteDifference', 'Model', BSModel, 'SpotPrice', SpotPriceATM, 'Discou
PriceFDATM = price(FDPricerATM, EuropeanCallOption);
```

Vanilla Option Price Comparison When Option Is OTM

Use the `displayPricesVanillaCallOption` in Local Functions on page 1-101 to compare the Vanilla call prices for OTM.

```
displayPricesVanillaCallOption("OTM", PriceBLSOTM, PriceCRROTM, PriceLROTM, PriceFDOTM)
```

```
Comparison of Vanilla Call Option Prices OTM:

Black-Scholes:        1.280591
Cox-Ross-Rubinstein:  1.278306
Leisen-Reimer:        1.280651
Finite-Difference:    1.280599
```

Vanilla Option Price Comparison When Option Is ATM

Use the `displayPricesVanillaCallOption` in Local Functions on page 1-101 to compare the Vanilla call prices for ATM.

```
displayPricesVanillaCallOption("ATM", PriceBLSATM, PriceCRRATM, PriceLRATM, PriceFDATM)
```

```
Comparison of Vanilla Call Option Prices ATM:

Black-Scholes:        3.505323
Cox-Ross-Rubinstein:  3.520559
Leisen-Reimer:        3.505377
Finite-Difference:    3.505452
```

Analyze Effect of Number of Tree Levels on Price of Options When Using `AssetTree` Pricer

Create graphs to visualize how convergence changes as the number of steps in the binomial calculation increases for the Cox-Ross-Rubinstein and Leisen-Reimer tree models, as well as the impact on convergence to changes to the asset price.

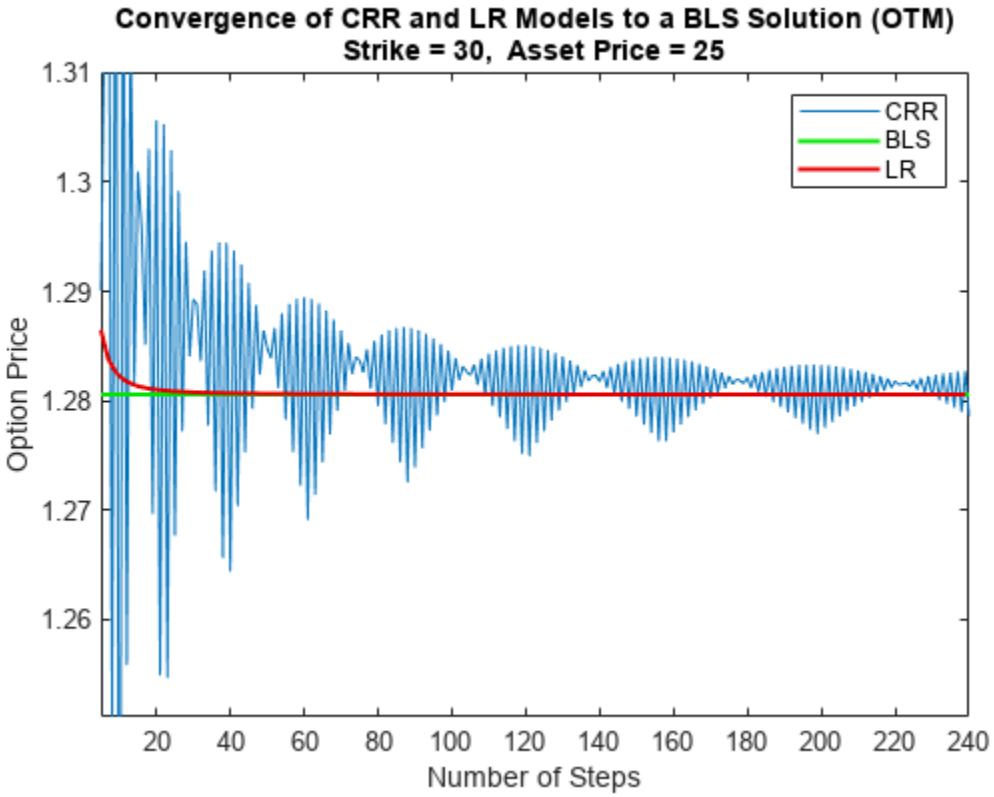
```
% Define the number of time steps of the tree
NPoints = 240;

% Cox-Ross-Rubinstein
NumPeriodCRR  = 5 : 1 : NPoints;
NbStepCRR     = length(NumPeriodCRR);
PriceOTMCRR = nan(NbStepCRR, 1);
PriceATMCRR = PriceOTMCRR;

for i = 1 : NbStepCRR

    PricerCRROTM = finpricer("AssetTree",'DiscountCurve',ZeroCurve,'Model',BSModel, 'SpotPrice',S
                   'PricingMethod',"CoxRossRubinstein",'NumPeriods', NumPeriodCRR(i), 'Maturity'
    PriceOTMCRR(i) = price(PricerCRROTM, EuropeanCallOption);

    PricerCRRATM = finpricer("AssetTree",'DiscountCurve',ZeroCurve,'Model',BSModel, 'SpotPrice',S
                   'PricingMethod',"CoxRossRubinstein",'NumPeriods', NumPeriodCRR(i), 'Maturity'
    PriceATMCRR(i) = price(PricerCRRATM, EuropeanCallOption);

end

% Leisen-Reimer
NumPeriodLR  = 5 : 2 : NPoints;
NbStepLR     = length(NumPeriodLR);
PriceOTMLR = nan(NbStepLR, 1);
PriceATMLR = PriceOTMLR;

for i = 1 : NbStepLR
    PricerLROTM = finpricer("AssetTree",'DiscountCurve',ZeroCurve,'Model',BSModel, 'SpotPrice',Sp
                   'PricingMethod',"LeisenReimer",'NumPeriods', NumPeriodLR(i), 'Maturity', Exerc
    PriceOTMLR(i) = price(PricerLROTM, EuropeanCallOption);

    PricerLRATM = finpricer("AssetTree",'DiscountCurve',ZeroCurve,'Model',BSModel, 'SpotPrice',Sp
                   'PricingMethod',"LeisenReimer",'NumPeriods', NumPeriodLR(i), 'Maturity', Exerc
    PriceATMLR(i) = price(PricerLRATM, EuropeanCallOption);
end
```

First Scenario: OTM Vanilla Call Option

Plot the convergence of CRR and LR models to a Black-Scholes solution for an OTM option.

```
% Cox-Ross-Rubinstein
plot(NumPeriodCRR, PriceOTMCRR);
hold on;
plot(NumPeriodCRR, PriceBLSOTM*ones(NbStepCRR,1),'Color',[0 0.9 0], 'linewidth', 1.5);
```

```
% Leisen-Reimer
plot(NumPeriodLR, PriceOTMLR, 'Color',[0.9 0 0], 'linewidth', 1.5);

% Concentrate the area of interest by clipping on the Y axis at five times the
% LR price:
YLimDelta = 5*abs(PriceOTMLR(1) - PriceBLSOTM);
ax = gca;
ax.YLim = [PriceBLSOTM - YLimDelta PriceBLSOTM + YLimDelta];
ax.XLim = [5 NPoints];

% Annotate plot
titleString = sprintf('\nConvergence of CRR and LR Models to a BLS Solution (OTM)\nStrike = %d,
title(titleString)
ylabel('Option Price')
xlabel('Number of Steps')
legend('CRR', 'BLS', 'LR', 'Location', 'NorthEast')
```



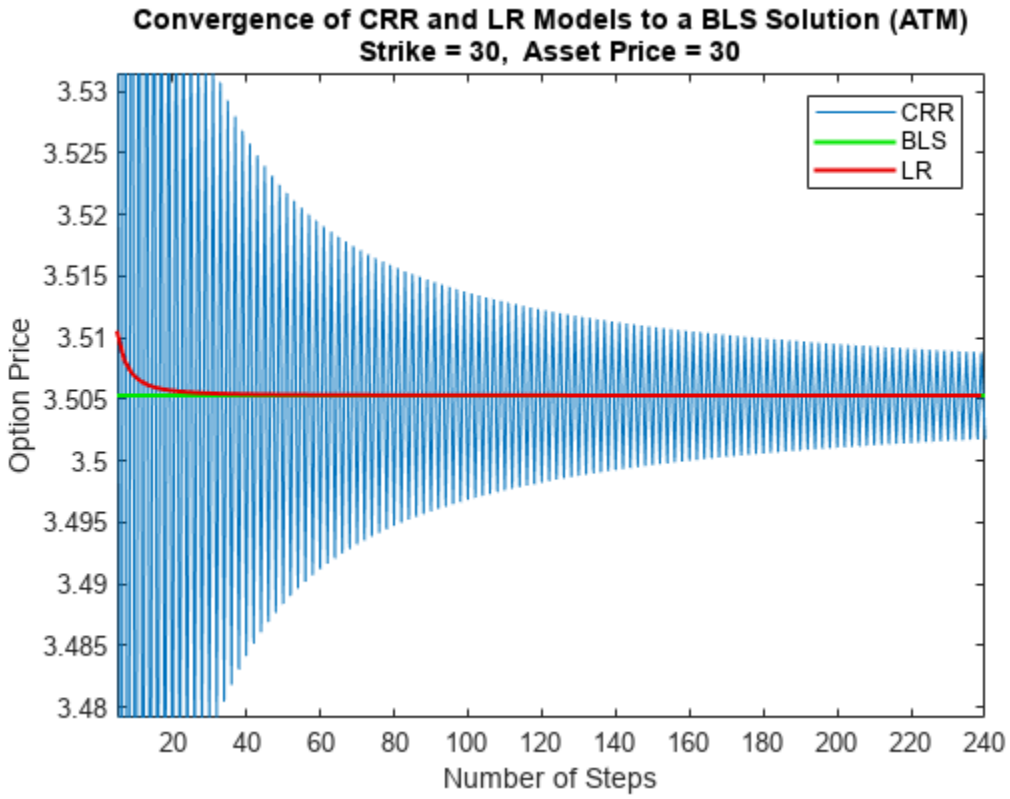
Observe that the Leisen-Reimer model removes the oscillation and produces estimates close to the Black-Scholes model using only a small number of steps.

Second Scenario: ATM Vanilla Call Option

Plot the convergence of CRR and LR models to a Black-Scholes solution for an ATM option.

```
% Cox-Ross-Rubinstein
figure;
plot(NumPeriodCRR, PriceATMCRR);
```

```
hold on;
plot(NumPeriodCRR, PriceBLSATM*ones(NbStepCRR,1),'Color',[0 0.9 0], 'linewidth', 1.5);

% Leisen-Reimer
plot(NumPeriodLR, PriceATMLR, 'Color',[0.9 0 0], 'linewidth', 1.5);

% Concentrate the area of interest by clipping on the Y axis at five times the
% LR price:
YLimDelta = 5*abs(PriceATMLR(1) - PriceBLSATM);
ax = gca;
ax.YLim = [PriceBLSATM - YLimDelta PriceBLSATM + YLimDelta];
ax.XLim = [5 NPoints];
% Annotate plot
titleString = sprintf('\nConvergence of CRR and LR Models to a BLS Solution (ATM)\nStrike = %d,
title(titleString)
ylabel('Option Price')
xlabel('Number of Steps')
legend('CRR', 'BLS', 'LR', 'Location', 'NorthEast')
```



While the CRR binomial model and the Black-Scholes model converge as the number of time steps increases, this convergence, except for the at-the-money options, is anything but smooth or uniform.

Local Functions

```
function displayPricesVanillaCallOption(type, PriceBLS, PriceCRR, PriceLR, PriceFD)
fprintf('Comparison of Vanilla Call Option Prices %s:\n', type);
fprintf('\n');
fprintf('Black-Scholes:        %f\n', PriceBLS);
```

```
fprintf('Cox-Ross-Rubinstein:  %f\n', PriceCRR);
fprintf('Leisen-Reimer:        %f\n', PriceLR);
fprintf('Finite-Difference:    %f\n', PriceFD);
fprintf('\n');
end
```

See Also

`FiniteDifference` | `AssetTree` | `BlackScholes` | `Vanilla`

Related Examples

- “Price Spread Instrument for a Commodity Using Black-Scholes Model and Analytic Pricers” on page 3-123

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53
- “Supported Exercise Styles” on page 1-62

External Websites

- Object-Based Framework for Pricing Financial Instruments (2 min 42 sec)

Price Weather Derivatives

This example demonstrates a workflow for pricing weather derivatives based on historically observed temperature data.

The workflow for pricing call and put options for weather derivatives includes these steps:

1. Get raw temperature data on page 1-104.
2. Load and process observed termperature data on page 1-105.
3. Visualize temperatures on page 1-106.
4. Analyze of seasonality on page 1-106.
5. Estimate deterministic trends in a time series on page 1-107.
6. Create and fit ARIMA/GARCH models on page 1-109.
7. Perform Monte Carlo simulation from a model on page 1-110.
8. Evaluate the number of heating degree days from simulation on page 1-111.
9. Price sample call and put options on page 1-113.

The techniques used in this example are based on the approach described in Alaton [1 on page 1-114].

What Is a Weather Derivative?

A weather derivative is a financial instrument used by companies or individuals to hedge against the risk of weather-related losses. Weather derivatives are index-based instruments that use observed weather data at a weather station to create an index on which a payout is based. The seller of a weather derivative agrees to bear the risk of disasters in return for a premium. If no damages occur before the expiration of the contract, the seller makes a profit—and in the event of unexpected or adverse weather, the buyer of the derivative claims the agreed amount.

How to Price Weather Derivatives

A number of different contracts for weather derivatives are traded on the over-the-counter (OTC) market. This example explores a simple option (call or put) based on the accumulation of heating degree days (HDD), which are the number of degrees that the temperature deviates from a reference level on a given day.

$$H_n = \sum_{i=1}^{n} \max\{18 - T_i, 0\}$$

There is no standard model for valuing weather derivatives similar to the Black-Scholes formula for pricing European equity options and similar derivatives. The underlying asset of the weather derivative is not tradeable, which violates a number of key assumptions of the Black-Scholes model. In this example, you price the option using the following formula from Alaton [1 on page 1-114]:

$$\chi = \alpha \max\{H_n - K, 0\} \qquad K = \text{strike level} \qquad \alpha = \text{tick size}$$

In general, the process of pricing weather derivative options is: obtain temperature data, clean the temperature data, and model the temperature to forecast the option price.

Get Raw Temperature Data

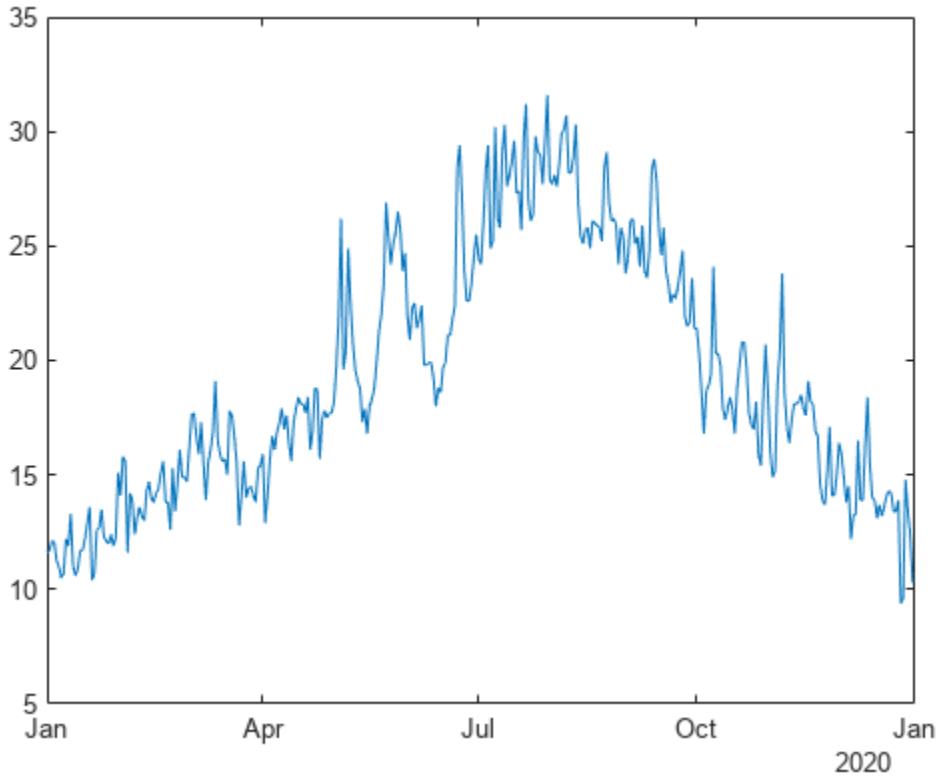
All data in this example is obtained from NOAA. Use `getGHCNData` in Local Functions on page 1-114 to extract the data, given a `stationID` and start and end dates. A REST API is also available for extracting data from NOAA. For more information on using a REST API, see https://github.com/mathworks/Climate-IAM-Explorer/tree/master/examples/Pricing%20Weather%20Derivatives.

To obtain the full temperature data for one year and plot the results, use `getGHCNData` in Local Functions on page 1-114.

```
stationID = 'SPE00119783';
Element = 'TAVG';
startDate = datetime(2020,1,1);
endDate = datetime(2020,12,31);
T = getGHCNData(stationID,Element,startDate,endDate);
head(T)
```

```
       Date        TAVG
    ___________    ____

    01-Jan-2020    11.6
    02-Jan-2020    11.7
    03-Jan-2020    12.1
    04-Jan-2020    12.1
    05-Jan-2020    11.3
    06-Jan-2020      11
    07-Jan-2020    10.5
    08-Jan-2020    10.7
```

```
figure
plot(T.Date, T.TAVG)
```



Load and Process Observed Temperature Data

This example uses data containing temperatures from 1978 to the end of 2020 near Stockholm. You can obtain this data by using `getGHCNData` in Local Functions on page 1-114.

```
stationID = 'SW000008525';
Element = 'TMAX';
startDate = datetime(1978,1,1);
endDate = datetime(2020,12,31);
Temp = getGHCNData(stationID,Element,startDate,endDate)
```

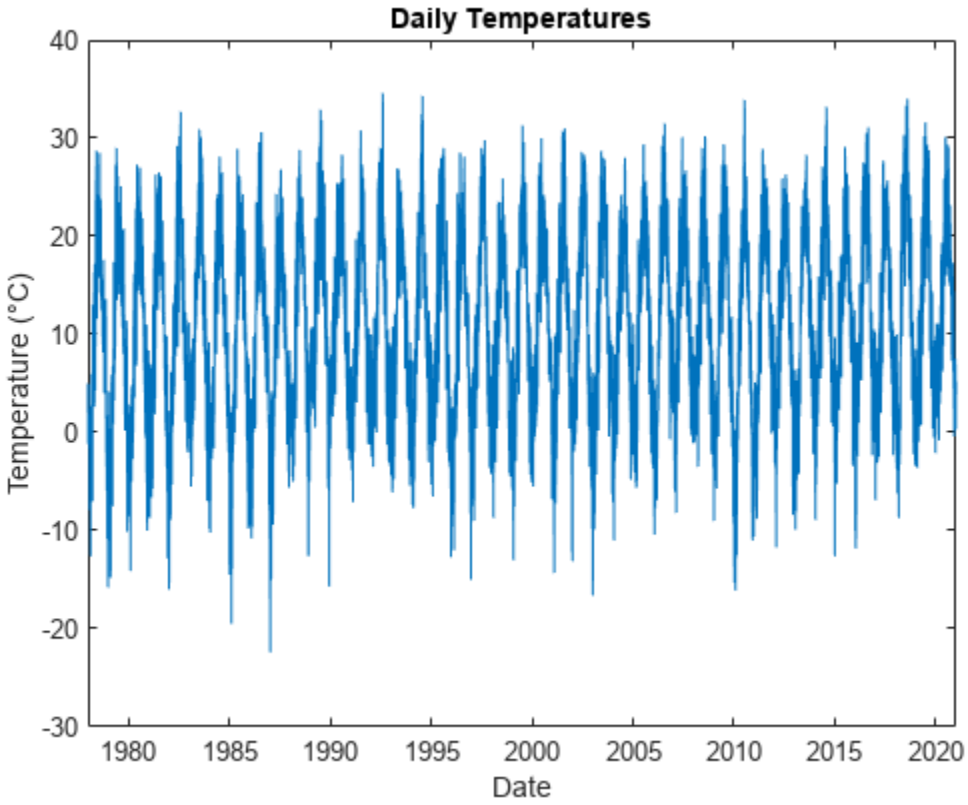
```
Temp=15706×1 timetable
       Date        TMAX
    ___________    ____

    01-Jan-1978     4.7
    02-Jan-1978       5
    03-Jan-1978     3.1
    04-Jan-1978    -0.5
    05-Jan-1978    -1.2
    06-Jan-1978     2.7
    07-Jan-1978     4.5
    08-Jan-1978     4.9
    09-Jan-1978     1.8
    10-Jan-1978     3.2
    11-Jan-1978     1.5
    12-Jan-1978    -0.5
    13-Jan-1978     0.6
```

```
14-Jan-1978       2
15-Jan-1978     2.5
16-Jan-1978     2.4
   ⋮
```

Visualize Temperatures

Use `plotTemperature` in Local Functions on page 1-114 to visualize the daily temperatures.

```
figure
plot( Temp.Date, Temp.TMAX)
xlabel("Date")
ylabel("Temperature (" + char(176) + "C)")
title("Daily Temperatures")
```



In the remaining sections of this example, you model the temperature and then evalauate sample option prices.

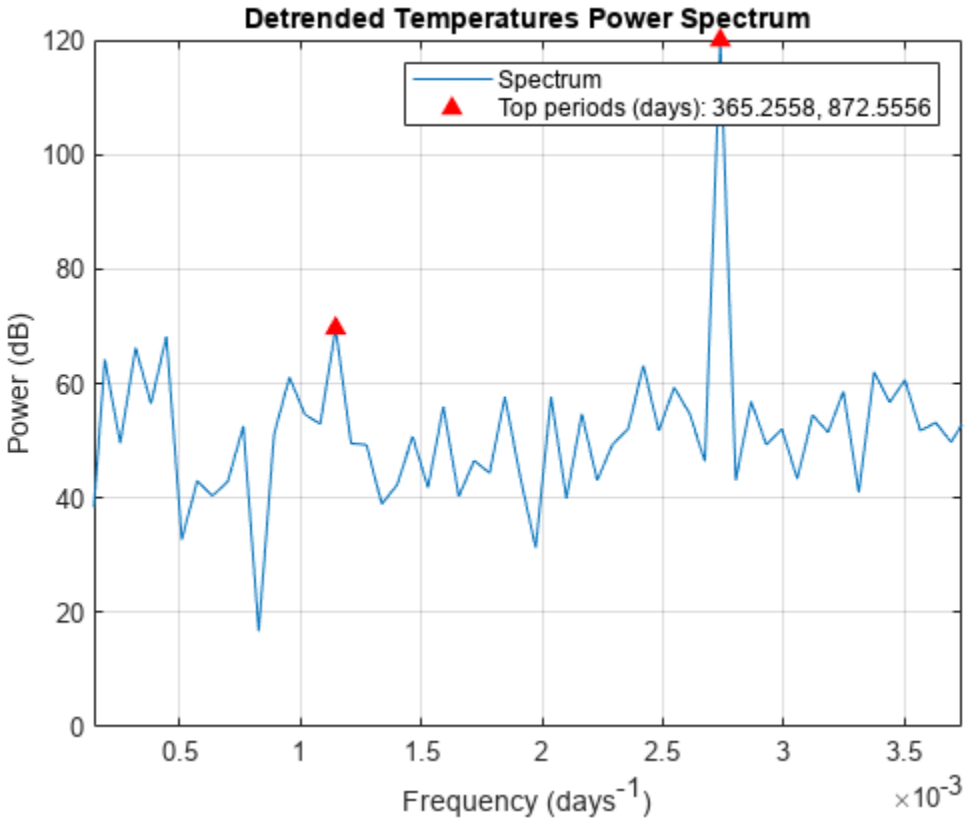
Determine Seasonality in Data

Assume that the deterministic component of the temperature model comprises a linear trend and seasonal terms. To estimate the main frequencies present in the time series, apply the Fourier transform.

Use `determineSeasonality` in Local Functions on page 1-114 to remove the linear trend by subtracting the best-fit line. Then, use the `periodogram` (Signal Processing Toolbox) function to

compute the spectrum (with a sample rate of one observation per day), and to visualize the spectrum. In addition, the `determineSeasonality` function identifies the top two component frequencies and periods in the data and adds these to the power spectrum.

```
determineSeasonality(Temp)
```



The plot illustrates that the dominant seasonal components in the data are the annual and 6-monthly cycles.

Fit Deterministic Trend

Based on the previous results, fit the following function for the temperature using the `fitlm` function.

$$\text{TemperatureFcn} = A + B\,t + C\sin(2\pi\,t) + D\cos(2\pi\,t) + E\cos(4\pi\,t)$$

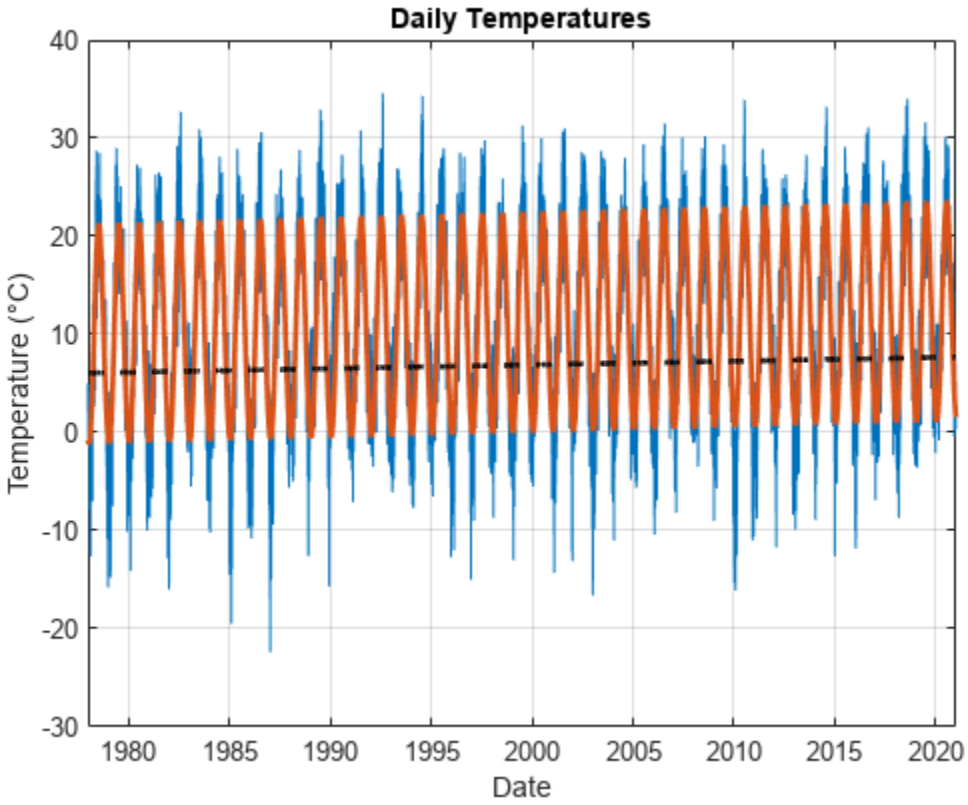
```
elapsedTime = years(Temp.Date - Temp.Date(1));
designMatrix = @(t) [t, cos( 2 * pi * t ), sin( 2 * pi * t ), cos( 4 * pi * t )];
trendPreds = ["t", "cos(2*pi*t)", "sin(2*pi*t)", "cos(4*pi*t)"];
trendModel = fitlm(designMatrix(elapsedTime), Temp.TMAX, "VarNames", [trendPreds, "Temperature"])
```

```
trendModel =
Linear regression model:
    Temperature ~ 1 + t + cos(2*pi*t) + sin(2*pi*t) + cos(4*pi*t)

Estimated Coefficients:
                   Estimate       SE         tStat       pValue
```

```
                    ________    _________    _______    __________

    (Intercept)       9.7095     0.064138     151.39             0
    t                0.054648    0.0025836     21.152    6.1629e-98
    cos(2*pi*t)       -10.909     0.045348    -240.56             0
    sin(2*pi*t)        -2.772     0.045357    -61.115             0
    cos(4*pi*t)       0.39298     0.045348     8.6659    4.9102e-18


Number of observations: 15706, Error degrees of freedom: 15701
Root Mean Squared Error: 4.02
R-squared: 0.798,  Adjusted R-Squared: 0.798
F-statistic vs. constant model: 1.55e+04, p-value = 0
```

Use the `plotDeterministicTrend` in Local Functions on page 1-114 to visualize the fitted trend.

```
trendModel = plotDeterministicTrend(Temp, trendModel, 1978, 2020)
```




```
trendModel =
Linear regression model:
    Temperature ~ 1 + t + cos(2*pi*t) + sin(2*pi*t) + cos(4*pi*t)

Estimated Coefficients:
                    Estimate       SE          tStat       pValue
                    ________    _________    _______    __________

    (Intercept)       9.7095     0.064138     151.39             0
    t                0.054648    0.0025836     21.152    6.1629e-98
```

```
    cos(2*pi*t)     -10.909     0.045348    -240.56             0
    sin(2*pi*t)      -2.772     0.045357    -61.115             0
    cos(4*pi*t)     0.39298     0.045348     8.6659    4.9102e-18


Number of observations: 15706, Error degrees of freedom: 15701
Root Mean Squared Error: 4.02
R-squared: 0.798,  Adjusted R-Squared: 0.798
F-statistic vs. constant model: 1.55e+04, p-value = 0
```

The coefficient of the *t* term is significant with a value of `0.038`. This value suggests that this geographic location has a warming trend of approximately `0.038`°C per year.

Analyze and Fit Model Residuals

Create an ARIMA/GARCH model for the residuals using `arima` (Econometrics Toolbox).

Based on the preceding results, create a separate time-series model for the regression model residuals using the following workflow:

1. Create an `arima` (Econometrics Toolbox) object and use the ARMA terms help to model the autocorrelation in the residual series.
2. Set the constant term to `0`, since the regression model above already includes a constant.
3. Use a *t*-distribution for the innovation series.
4. Use a `garch` (Econometrics Toolbox) model for the variance of the residuals to model the autocorrelation in the squared residuals.

```
trendRes = trendModel.Residuals.Raw;

resModel = arima( "ARLags", 1, ...
    "MALags", 1, ...
    "Constant", 0, ...
    "Distribution", "t", ...
    "Variance", garch( 1, 1 ) );
resModel = estimate(resModel,trendRes);


    ARIMA(1,0,1) Model (t Distribution):

                 Value      StandardError    TStatistic      PValue
                ________    _____________    __________    __________

    Constant          0               0           NaN             NaN
    AR{1}       0.76851       0.0063873        120.32               0
    MA{1}      0.075023        0.010201        7.3545       1.9168e-13
    DoF          10.369         0.75567        13.722       7.4771e-43


    GARCH(1,1) Conditional Variance Model (t Distribution):

                 Value      StandardError    TStatistic      PValue
                ________    _____________    __________    __________

    Constant    0.67469        0.091761        7.3527       1.9426e-13
```

```
    GARCH{1}     0.82015       0.019319        42.453              0
    ARCH{1}     0.070964      0.0069394        10.226      1.5119e-24
    DoF           10.369        0.75567        13.722      7.4771e-43
```

The residuals are not normally distributed or independent. Also, the residuals show evidence of GARCH effects.

Simulate Future Temperature Scenarios

Now that you have a calibrated temperature model, you can use it to simulate future temperature paths.

Prepare the simulation time.

```
nDays = 730;
simDates = Temp.Date(end) + caldays(1:nDays).';
simTime = years(simDates - Temp.Date(1));
```

Use `predict` to predict the deterministic trend.

```
trendPred = predict(trendModel, designMatrix(simTime));
```

Simulate from the ARIMA/GARCH model using `simulate` (Econometrics Toolbox).

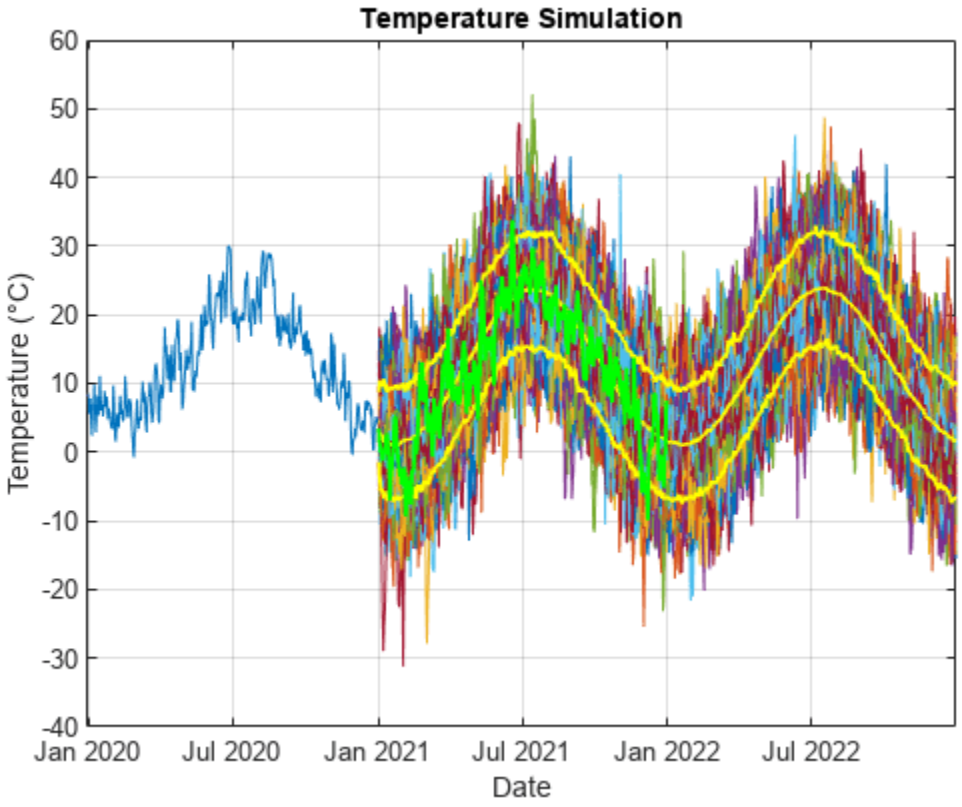
```
simRes = simulate(resModel, nDays, "NumPaths", 1000, "Y0", trendRes);
```

Add the deterministic trend to the simulated residuals.

```
simTemp = simRes + trendPred;
```

Use `visualizeScenarios` in Local Functions on page 1-114 to visualize the temperature scenarios.

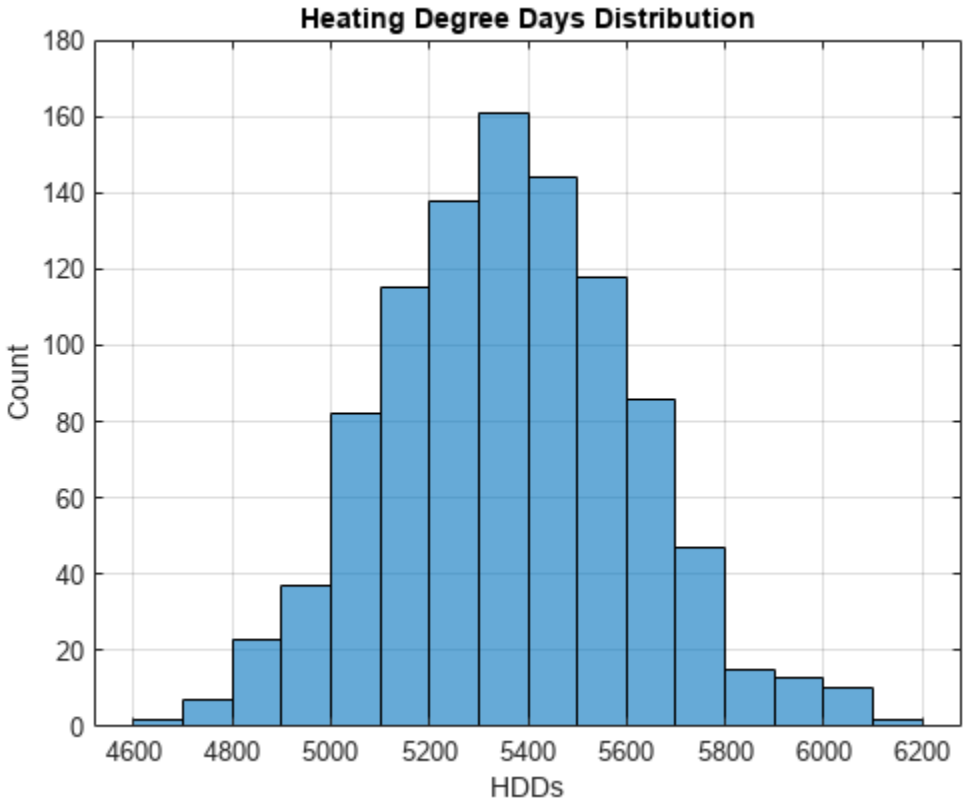
```
visualizeScenarios(Temp, simTemp, simDates)
```



Evaluate Number of Heating Degree Days from Simulation

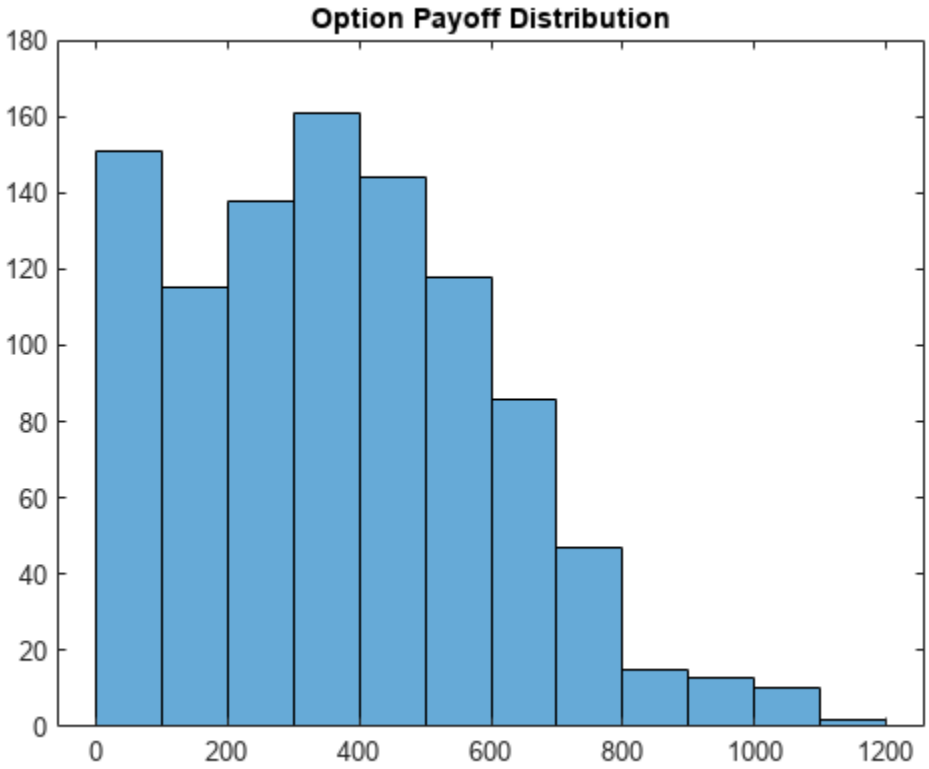
The number of heating degree days is given by $H = \sum_{i=1}^{n} \max(18 - T_i, 0)$, where $T_1, T_2, \ldots, T_n$ are the simulated temperatures.

```
H = sum(max(18 - simTemp, 0));

figure
histogram(H)
xlabel("HDDs")
ylabel("Count")
title("Heating Degree Days Distribution")
grid on
```



Using the original formula $\chi = \alpha \max\{H_n - K, 0\}$ from Alaton [1 on page 1-114], you can obtain a payoff distribution. When `K = 7400`, the `max` of `H_n-K` and `0` is always `0` for the set of simulation results in this example. Consequently, the histogram shows the distribution of 1000 zeros, that is, a single bar centered at `0` with a height of `1000`. As `K` decreases to `5000`, `H_n-K` has more opportunities to be positive, so the histogram changes shape.

```
K =    5000                          ;
figure
alpha = 1;
chi = alpha*max(H-K, 0); % Payoff of option
histogram(chi)
title("Option Payoff Distribution")
```



Price Sample Call and Put Options

The weather derivative option pricing calculation is based on H (the number of heating degree days) which is defined in the Evaluate Number of Heating Degree Days From Simulation on page 1-111 section.

First, compute the mean and standard deviation of H.

```
muH = mean(H);
sigmaH = std(H);
```

Define the sample option parameters.

```
r = 0.01; % Risk-free interest rate
K = 5000; % Strike value
TTM = 0.25; % Expiry time
```

Evaluate α.

```
a = (K - muH) / sigmaH;
```

Compute the price of the call and put options.

```
call_option = exp(-r*TTM) * ((muH - K) * normcdf(-a) + (sigmaH / sqrt(2*pi) * exp(-0.5*a^2)))
```

```
call_option = 371.9754
```

```
put_option = exp(-r*TTM) * ((K - muH) * (normcdf(a) - normcdf(-muH / sigmaH)) + sigmaH / sqrt(2*p
```

```
put_option = 8.5494
```

References

[1] Alaton, P., Djehiche, B. and D. Stillberger, D. *On Modelling and Pricing Weather Derivatives.* Applied Mathematical Finance. 9(1): 1–20, 2002.

[2] Climate Data Online: Web Services Documentation at https://www.ncdc.noaa.gov/cdo-web/webservices/v2/.

Local Functions

```
function TT_out = getGHCNData(stationID,Element,startDate,endDate)
% Function to retrieve GHCN Data

% Import options for GHCN Data files. For more information, see:
% https://www1.ncdc.noaa.gov/pub/data/ghcn/daily/readme.txt
DataStartLine = 1;
NumVariables = 31*4 + 4;
varNames_Array = (["Value","MFlag","QFlag","SFlag"] + (1:31)')';
VariableNames  = ["ID","Year","Month","Element",varNames_Array(:)'];
VariableWidths = [11 4 2 4 repmat([5 1 1 1],1,31)];
DataType       = [{'char','double','double','char'} ...
    repmat({'double' 'char' 'char' 'char'},1,31)];
opts = fixedWidthImportOptions('NumVariables',NumVariables, ...
    'DataLines',DataStartLine, ...
    'VariableNames',VariableNames, ...
    'VariableWidths',VariableWidths, ...
    'VariableTypes',DataType);

% Read the full data set.
TT_Full = readtable("https://www1.ncdc.noaa.gov/pub/data/ghcn/daily/all/" + ...
    stationID + ".dly",opts);

% Extract the Element data.
TT_Element_Full = TT_Full(TT_Full.Element == string(Element),["ID" "Year" ...
    "Month" "Element" "Value" + (1:31)]);

reqYears = year(startDate):year(endDate);
TT_Element = TT_Element_Full(ismember(TT_Element_Full.Year,reqYears),:);

fullyearDays = datetime(year(startDate),1,1):datetime(year(endDate),12,31);

tmpData = TT_Element{:,5:end}';
tmpData = tmpData(:);
tmpData(tmpData == -9999) = []; % Remove any empty data

Date = (startDate:endDate)';
reqIndex = ismember(fullyearDays,Date);
TT_out = timetable(Date,tmpData(reqIndex)/10,'VariableNames',{Element});

end

function determineSeasonality(Temp)
```

```
% Assume that the deterministic component of the temperature model comprises a linear trend and s
% To estimate the main frequencies present in the time series, apply the Fourier transform.
% First, remove the linear trend by subtracting the best-fit line.

detrendedTemps = detrend(Temp.TMAX);

% Next, use the periodogram function to compute the spectrum. The sampling frequency is one obse
numObs = length(detrendedTemps);
Fs = 1;
[pow, freq] = periodogram(detrendedTemps, [], numObs, Fs);

% Visualize the spectrum.
powdB = db(pow);
figure
plot(freq, powdB)
xlabel("Frequency (days^{-1})")
ylabel("Power (dB)")
title("Detrended Temperatures Power Spectrum")
grid on

% Identify the top two component frequencies and periods in the data.
[topPow, idx] = findpeaks(powdB, "NPeaks", 2, ...
    "SortStr", "descend", ...
    "MinPeakProminence", 20);
topFreq = freq(idx);
topPeriods = 1 ./ topFreq;

% Add the top two component frequencies and periods to the spectrum.
hold on
plot(topFreq, topPow, "r^", "MarkerFaceColor", "r")
xlim([min( topFreq ) - 1e-3, max( topFreq ) + 1e-3])
legend("Spectrum", "Top periods (days): " + join( string( topPeriods ), ", " ))
% Note that the dominant seasonal components in the data are the annual and 6-monthly cycles.

end

function trendModel = plotDeterministicTrend(Temp, trendModel, from, to)
% Visualize the fitted trend.
figure;
plot(Temp.Date, Temp.TMAX)
xlabel("Date")
ylabel("Temperature (" + char( 176 ) + "C)")
title("Daily Temperatures")
grid on
hold on
plot(Temp.Date, trendModel.Fitted, "LineWidth", 2)
plot(Temp.Date,6.06+0.038*years(Temp.Date - Temp.Date(1)),'k--','LineWidth',2)
xlim([datetime(from,1,1), datetime(to,12,31)])
end

function visualizeScenarios(Temp, simTemp, simDates)
figure
plot(Temp.Date, Temp.TMAX)
hold on
plot(simDates, simTemp)

% Plot the simulation percentiles.
simPrc = prctile(simTemp, [2.5, 50, 97.5], 2);
```

```
plot(simDates, simPrc, "y", "LineWidth", 1.5)
xlim([Temp.Date(end) - calyears(1), simDates(end)])
xlabel("Date")
ylabel("Temperature (" + char( 176 ) + "C)")
title("Temperature Simulation")
grid on

% Get the data for 2021.
stationID = 'SW000008525';
Element = 'TMAX';
startDate = datetime(2021,1,1);
endDate = datetime(2021,12,31);
T = getGHCNData(stationID,Element,startDate,endDate);
plot(T.Date,  T.TMAX, "g", "LineWidth", 1.5)

end
```

See Also

External Websites

- Pricing Weather Options with MATLAB (15 min 14 sec)

Interest-Rate Derivatives

Supported Interest-Rate Instrument Functions

In this section...

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 Instruments 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.
`bondbycir`	Price bonds using a CIR tree model.
`bondbyzero`	Price a bond using a set of zero curves.
`instbond`	Construct a bond instrument.

Convertible Bond

A convertible bond is a financial instrument that combines equity and debt features. It is a bond with the embedded option to turn it into a fixed number of shares. The holder of a convertible bond has the right, but not the obligation, to exchange the convertible security for a predetermined number of equity shares at a preset price. The debt component is derived from the coupon payments and the principal. The equity component is provided by the conversion feature.

Convertible bonds have several defining features:

- Coupon — The coupon in convertible bonds are typically lower than coupons in vanilla bonds since investors are willing to take the lower coupon for the opportunity to participate in the company's stock via the conversion.
- Maturity — Most convertible bonds are issued with long-stated maturities. Short-term maturity convertible bonds usually do not have call or put provisions.
- Conversion ratio — Conversion ratio is the number of shares that the holder of the convertible bond receives from exercising the call option of the convertible bond:

  ```
  Conversion ratio = par value convertible bond/conversion price of equity
  ```

  For example, a conversion ratio of 25 means a bond can be exchanged for 25 shares of stock. This also implies a conversion price of $40 (1000/25). This, $40, would be the price at which the owner would buy the shares. This can be expressed as a ratio or as the conversion price and is specified in the contract along with other provisions.
- Option type:
  - Callable Convertible: a convertible bond that is callable by the issuer. The issuer of the bond forces conversion, removing the advantage that conversion is at the discretion of the bondholder. Upon call, the bondholder can either convert the bond or redeem at the call price.

This option enables the issuer to control the price of the convertible bond and if necessary refinance the debt with a new cheaper one.

- Puttable Convertible: a convertible bond with a put feature that allows the bondholder to sell back the bond at a premium on a specific date. This option protects the holder against rising interest rates by reducing the year to maturity.

Function	Purpose
`cbondbycrr`	Price convertible bonds using a CRR binomial tree with the Tsiveriotis and Fernandes model.
`cbondbyeqp`	Price convertible bonds using an EQP binomial tree with the Tsiveriotis and Fernandes model.
`cbondbyitt`	Price convertible bonds using an implied trinomial tree with the Tsiveriotis and Fernandes model.
`cbondbystt`	Price convertible bonds using a standard trinomial tree with the Tsiveriotis and Fernandes model.
`instcbond`	Construct a `cbond` instrument for a convertible bond.

Stepped Coupon Bonds

A step-up and step-down bond is a debt security with a predetermined coupon structure over time. With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. For more information on options features (call and puts), see “Stepped Coupon Bonds with Calls and Puts” on page 2-8. The following functions have a modified `CouponRate` argument to support a new variable coupon schedule allowing pricing of stepped coupon bonds.

Function	Purpose
`bondbyzero`	Price bonds using a term structure model.
`bondbybdt`	Price bonds using a BDT tree model.
`bondbyhjm`	Price bonds using an HJM tree model.
`bondbyhw`	Price bonds using an HW tree model.
`bondbybk`	Price bonds using a BK tree model.
`bondbycir`	Price bonds using a CIR tree model.
`instbond`	Construct a bond instrument.
`instoptbnd`	Construct a bond option instrument.
`instdisp`	Display instruments stored in a variable.

Sinking Fund Bonds

A sinking fund bond is a coupon bond with a sinking fund provision. This provision obligates the issuer to amortize portions of the principal before maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity. For more information on options support for sinking fund bonds, see “Sinking Fund Bonds with an Embedded Option” on page 2-8. The following functions have a modified `Face` argument to support a variable face schedule for pricing bonds with a sinking provisions.

Function	Purpose
`bondbyzero`	Price bonds using a term structure model.
`bondbybdt`	Price bonds using a BDT tree model.
`bondbyhjm`	Price bonds using an HJM tree model.
`bondbyhw`	Price bonds using an HW tree model.
`bondbybk`	Price bonds using a BK tree model.
`bondbycir`	Price bonds using a CIR tree model.
`instoptbnd`	Construct a bond option instrument.
`instbond`	Construct a bond instrument.
`instdisp`	Display instruments stored in a variable.

Bonds with an Amortization Schedule

A bond with an amortization schedule repays part of the principal (face value) along with the coupon payments. An amortizing bond is a special case of a sinking fund bond when there is no market purchase option and no call provision. The following functions have a modified `Face` argument to support an amortization schedule.

Function	Purpose
`bondbyzero`	Price bonds using a term structure model.
`bondbybdt`	Price bonds using a BDT tree model.
`bondbyhjm`	Price bonds using an HJM tree model.
`bondbyhw`	Price bonds using an HW tree model.
`bondbybk`	Price bonds using a BK tree model.
`bondbycir`	Price bonds using a CIR tree model.

Bond Options

Financial Instruments Toolbox 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 Instruments 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.

Function	Purpose
`optbndbycir`	Price a bond option price using a CIR interest-rate tree.
`instoptbnd`	Construct a bond option instrument.

Bond with Embedded Options

A bond with embedded options allows the issuer to buy back (callable) or redeem (puttable) the bond at a predetermined price at specified future dates. Financial Instruments Toolbox 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*

  In the callable case, the holder bought a bond and sold a call option to the issuer. For example, if interest rates go down by the time of the call date, the issuer is able to refinance its debt at a cheaper level and can call the bond.
- For a puttable bond: *PricePuttableBond* = *PriceBond* + *PricePutOption*

  In the puttable case, the holder bought a bond and a put option. For example, if interest rates rise, the future value of coupon payments becomes less valuable. Therefore, the investor can sell the bond back to the issuer and then lend proceeds elsewhere at a higher rate.

In addition, Option Adjusted Spread (OAS) is a useful way to value and compare securities with embedded options, like callable or puttable bonds. For more information on OAS, see “OAS for Callable and Puttable Bonds” on page 2-64.

Financial Instruments 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.
`optembndbycir`	Price a bond with embedded options using a CIR interest-rate tree.
`instoptembnd`	Construct a bond-with-embedded-options instrument.
`oasbybdt`	Determine an option adjusted spread using Black-Derman-Toy model.
`oasbybk`	Determine an option adjusted spread using Black-Karasinski model.

Function	Purpose
oasbyhjm	Determine an option adjusted spread using Heath-Jarrow-Morton model.
oasbyhw	Determine an option adjusted spread using Hull-White model.
oasbycir	Determine an option adjusted spread using Cox-Ingersoll-Ross model.
agencyoas	Compute the OAS of the callable bond using the Agency OAS model.
agencyprice	Price the callable bond OAS using the Agency OAS model.

Stepped Coupon Bonds with Calls and Puts

A step-up and step-down bond is a debt security with a predetermined coupon structure over time. For more information on stepped coupon bonds, see “Stepped Coupon Bonds” on page 2-5. Stepped coupon bonds can have options features (call and puts). The following functions have a modified `CouponRate` argument to support a new variable coupon schedule allowing pricing stepped coupon bonds with callable and puttable features:

Function	Purpose
optembndbybdt	Price bonds with embedded options using a BDT model tree.
optembndbyhjm	Price bonds with embedded options using an HJM model tree.
optembndbybk	Price bonds with embedded options using a BK model tree.
optembndbyhw	Price bonds with embedded options using an HW model tree.
optembndbycir	Price bonds with embedded options using a CIR model tree.
instbond	Construct a bond instrument.
instoptbnd	Construct a bond option instrument.
instoptembnd	Construct a bond with an embedded option instrument.
instdisp	Display instruments stored in a variable.

Sinking Fund Bonds with an Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision. For more information on sinking fund bonds, see “Sinking Fund Bonds” on page 2-5. The sinking fund bond can have a sinking fund option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper.

If interest rates are high, then the issuer buys back the required amount of bonds from the market since bonds are cheap. But if interest rates are low (bond prices are high), then most likely the issuer buys the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a nonsinking fund bond. The following functions have a modified `Face` argument to support a variable face schedule for pricing bonds with a sinking fund option provision.

Function	Purpose
optembndbybdt	Price bonds with embedded options using a BDT model tree.
optembndbyhjm	Price bonds with embedded options using an HJM model tree.
optembndbybk	Price bonds with embedded options using a BK model tree.
optembndbyhw	Price bonds with embedded options using an HW model tree.
optembndbycir	Price bonds with embedded options using a CIR model tree.
instbond	Construct a bond instrument.
instoptbnd	Construct a bond option instrument.
instdisp	Display instruments stored in a variable.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face` input argument. An amortizing callable bond give the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Function	Purpose
optembndbybdt	Price bonds with embedded options using a BDT model tree.
optembndbyhjm	Price bonds with embedded options using an HJM model tree.
optembndbybk	Price bonds with embedded options using a BK model tree.
optembndbyhw	Price bonds with embedded options using an HW model tree.
optembndbycir	Price bonds with embedded options using a CIR model tree.
oasbybdt	Determine an option adjusted spread using Black-Derman-Toy model.
oasbybk	Determine an option adjusted spread using Black-Karasinski model.
oasbyhjm	Determine an option adjusted spread using Heath-Jarrow-Morton model.
oasbyhw	Determine an option adjusted spread using Hull-White model.
oasbycir	Determine an option adjusted spread using Cox-Ingersoll-Ross model.

Fixed-Rate Note

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

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.

Function	Purpose
`fixedbycir`	Price a fixed-rate note using a CIR 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.

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.
`floatbycir`	Price a floating-rate note using a CIR interest-rate tree.
`floatbyzero`	Price a floating-rate note using a set of zero curves.
`instfloat`	Construct a floating-rate note instrument.

Floating-Rate Note with an Amortization Schedule

A floating-rate note with an amortization schedule repays part of the principal (face value) along with the coupon payments. The following functions have a `Principal` argument to support an amortization schedule.

Function	Purpose
`floatbyzero`	Price floating-rate note from set of zero curves.
`floatbybdt`	Price floating-rate note from Black-Derman-Toy interest-rate tree.
`floatbyhjm`	Price floating-rate note from Heath-Jarrow-Morton interest-rate tree.
`floatbyhw`	Price floating-rate note from Hull-White interest-rate tree.
`floatbybk`	Price floating-rate note from Black-Karasinski interest-rate tree.
`floatbycir`	Price a floating-rate note using a CIR interest-rate tree.

Floating-Rate Note with Caps, Collars, and Floors

A floating-rate note with caps, collars, and floors. This type of instrument can carry restrictions on the maximum (cap) or minimum (floor) coupon rate paid. A cap is an unattractive feature for an investor, since they constrain the coupon rates from increasing. A floor is an attractive feature, since it allows investors to get a minimum coupon rate when market rates decrease below a certain level. Also, a floating-rate note can have a collar which is a combination of a cap and a floor together. The following functions have a `CapRate` and `FloorRate` argument to support a capped, collared, or floored floating-rate note.

Function	Purpose
`floatbybdt`	Price a capped floating-rate note from a Black-Derman-Toy interest-rate tree.
`floatbyhjm`	Price a capped floating-rate note from a Heath-Jarrow-Morton interest-rate tree.
`floatbyhw`	Price a capped floating-rate note from a Hull-White interest-rate tree.
`floatbybk`	Price a capped floating-rate note from a Black-Karasinski interest-rate tree.
`floatbycir`	Price a floating-rate note using a CIR interest-rate tree.
`instfloat`	Create a capped floating-rate note instrument.
`instadd`	Add a capped floating-rate note instrument to a portfolio.

Floating-Rate Note Options

Financial Instruments Toolbox supports three types of put and call options on floating-rate notes:

- 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 only exercise it on predetermined dates, usually monthly.

Financial Instruments Toolbox supports the following for pricing and specifying a floating-rate note option:

Function	Purpose
`optfloatbybdt`	Price an option for floating-rate note using a Black-Derman-Toy interest-rate tree.
`optfloatbyhjm`	Price an option for floating-rate note using a Heath-Jarrow-Morton interest-rate tree.
`optfloatbyhw`	Price an option for floating-rate note using a Hull-White interest-rate tree.
`optfloatbycir`	Price an option for floating-rate note using a Cox-Ingersoll-Ross interest-rate tree.
`optfloatbybk`	Price an option for floating-rate note using a Black-Karasinski interest-rate tree.
`instoptfloat`	Define the option instrument for floating-rate note.

Floating-Rate Note with Embedded Options

A floating-rate note with an embedded option enables floating-rate notes to have early redemption features. An FRN with an embedded option gives investors or issuers the option to retire the outstanding principal prior to maturity. An embedded call option gives the right to retire the note prior to the maturity date (callable floater), and an embedded put option gives the right to sell the note back at a specific price (puttable floater).

Financial Instruments Toolbox supports the following for pricing and specifying a floating-rate note with an embedded option:

Function	Purpose
`optemfloatbybdt`	Price an embedded option for floating-rate note using a Black-Derman-Toy interest-rate tree.
`optemfloatbybk`	Price an embedded option for floating-rate note using a Black-Karasinski interest-rate tree.
`optemfloatbyhjm`	Price an embedded option for floating-rate note using a Heath-Jarrow-Morton interest-rate tree.
`optemfloatbyhw`	Price an embedded option for floating-rate note using a Hull-White interest-rate tree.
`optemfloatbycir`	Price an embedded option for floating-rate note using a Cox-Ingersoll-Ross interest-rate tree.
`instoptemfloat`	Define the floating-rate note with embedded option 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:

$\max(CurrentRate - CapRate, 0)$

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.
`capbycir`	Price a cap instrument using a CIR interest-rate tree.
`capbyblk`	Price a cap instrument using the Black option pricing model.
`capbylg2f`	Price a cap using Linear Gaussian two-factor model.
`capbynormal`	Price a cap instrument with negative rates using the Normal (Bachelier) option pricing model.
`capvolstrip`	Strip caplet volatilities from flat cap volatilities.
`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:

$\max(FloorRate - CurrentRate, 0)$

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.
`floorbycir`	Price a floor instrument using a CIR interest-rate tree.
`floorbyblk`	Price a floor instrument using the Black option pricing model.
`floorbylg2f`	Price a floor using Linear Gaussian two-factor model.
`floorbynormal`	Price a floor instrument with negative rates using the Normal (Bachelier) option pricing model.
`floorvolstrip`	Strip floorlet volatilities from flat floor volatilities.
`instfloor`	Construct a floor instrument.

Range Note

A range note is a structured (market-linked) security whose coupon-rate is equal to the reference rate as long as the reference rate is within a certain range. If the reference rate is outside of the range, the coupon-rate is 0 for that period. This type of instrument entitles the holder to cash flows that depend on the level of some reference interest-rate that is floored to be positive and gives the holder of the note direct exposure to the reference rate. This type of instrument is useful for cases where you believe that interest rates will stay within a certain range. In return for the drawback that no interest is paid for the time the range is left, a range note offers higher coupon rates than comparable standard products, like vanilla floating notes.

Function	Purpose
`instrangefloat`	Create a range note instrument.
`rangefloatbybdt`	Price range floating note using a BDT tree.
`rangefloatbybk`	Price range floating note using a BK tree.
`rangefloatbyhjm`	Price range floating note using an HJM tree.
`rangefloatbyhw`	Price range floating note using an HW tree.
`rangefloatbycir`	Price range floating note using a CIR tree.

Swap

A swap is contract between two parties obligating the parties to exchange future cash flows. A vanilla swap is composed of a floating-rate leg and a fixed-rate leg.

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.

Function	Purpose
`swapbycir`	Price a swap instrument using a CIR interest-rate tree.
`swapbyzero`	Price a swap instrument using a set of zero curves and price cross currency swaps.
`instswap`	Construct a swap instrument.

Swap with an Amortization Schedule

A swap with an amortization schedule repays part of the principal (face value) along with the coupon payments. A swap with an amortization schedule is used to manage interest rate risk and serve as a cash flow management tool. For this particular type of swap, the notional amount decreases over time. This means that interest payments decrease not only on the floating leg but also on the fixed leg. The following swap functions have a `Principal` argument to support an amortization schedule.

Function	Purpose
`swapbyzero`	Price swap instrument from set of zero curves.
`swapbybdt`	Price swap instrument from Black-Derman-Toy interest-rate tree.
`swapbyhjm`	Price swap instrument from Heath-Jarrow-Morton interest-rate tree.
`swapbyhw`	Price swap instrument from Hull-White interest-rate tree.
`swapbybk`	Price swap instrument from Black-Karasinski interest-rate tree.
`swapbycir`	Price a swap instrument using a CIR interest-rate tree.
`instswap`	Construct swap instrument.

Forward Swap

In a forward interest-rate swap, a fixed interest-rate loan is exchanged for a floating interest-rate loan at a future specified date. The following functions have a `StartDate` argument to support the future date for the forward swap.

Function	Purpose
`swapbyzero`	Price a forward swap from a zero curve.
`swapbybdt`	Price a forward swap from a Black-Derman-Toy interest-rate tree.
`swapbyhjm`	Price a forward swap from a Heath-Jarrow-Morton interest-rate tree.
`swapbyhw`	Price a forward swap from a Hull-White interest-rate tree.
`swapbybk`	Price a forward swap from a Black-Karasinski interest-rate tree.
`swapbycir`	Price a swap instrument using a CIR interest-rate tree.
`instswap`	Create a forward swap instrument.
`instadd`	Add a capped floating-rate note instrument to a portfolio.

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.

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.
`swaptionbycir`	Price a swaption instrument using a CIR interest-rate tree.
`swaptionbyblk`	Price swaptions using the Black model with a forward on a swap.
`swaptionbylg2f`	Price European swaptions using Linear Gaussian two-factor model.
`swaptionbynormal`	Price swaptions for negative rates using the Normal (Bachelier) model with a forward on a swap.
`instswaption`	Construct a swaption instrument.

Use `swaptionbyblk` to price a swaption using the Black model. The Black model is standard model used in the swaption market when pricing European swaptions. This type of model is widely used by when speed is important to quickly obtain a price at settlement date, even if the price is less accurate than other swaption pricing models based on interest-rate tree models.

Bond Futures

Bond futures are futures contracts where the commodity for delivery is a government bond. There are established global markets for government bond futures. Bond futures provide a liquid alternative for managing interest-rate risk.

In the US market, the Chicago Mercantile Exchange (CME) offers futures on Treasury bonds and notes with maturities of 2, 5, 10, and 30 years. Typically, the following bond future contracts from the CME have maturities of 3, 6, 9, and 12 months:

- 30-year U.S. Treasury bond
- 10-year U.S. Treasury bond
- 5-year U.S. Treasury bond
- 2-year U.S. Treasury bond

The short position in a Treasury bond or note future contract must deliver to the long position in one of many possible existing Treasury bonds. For example, in a 30-year Treasury bond future, the short position must deliver a Treasury bond with at least 15 years to maturity. Because these bonds have different values, the bond future contract is standardized by computing a conversion factor. The conversion factor normalizes the price of a bond to a theoretical bond with a coupon of 6%. The price of a bond future contract is represented as:

$InvoicePrice = FutPrice \times CF + AI$

where:

FutPrice is the price of the bond future.

CF is the conversion factor for a bond to deliver in a futures contract.

AI is the accrued interest.

The short position in a futures contract has the option of which bond to deliver and, in the US bond market, when in the delivery month to deliver the bond. The short position typically chooses to deliver the bond known as the Cheapest to Deliver (CTD). The CTD bond most often delivers on the last delivery day of the month.

Financial Instruments Toolbox supports the following bond futures:

- US Treasury bonds and notes
- German Bobl, Bund, Buxl, and Schatz
- UK gilts
- Japanese government bonds (JGBs)

The functions supporting all bond futures are:

Function	Purpose
`convfactor`	Calculates bond conversion factors for US Treasury bonds, German Bobl, Bund, Buxl, and Schatz, UK gilts, and JGBs.
`bndfutprice`	Prices bond future given repo rates.
`bndfutimprepo`	Calculates implied repo rates for a bond future given price.

The functions supporting US Treasury bond futures are:

Function	Purpose
`tfutbyprice`	Calculates future prices of Treasury bonds given the spot price.
`tfutbyyield`	Calculates future prices of Treasury bonds given current yield.
`tfutimprepo`	Calculates implied repo rates for the Treasury bond future given price.
`tfutpricebyrepo`	Calculates Treasury bond futures price given the implied repo rates.
`tfutyieldbyrepo`	Calculates Treasury bond futures yield given the implied repo rates.

For more information on bond futures, see “Bond Futures” on page 7-10.

See Also

`instbond` | `instcap` | `instcf` | `instfixed` | `instfloat` | `instfloor` | `instoptbnd` | `instoptembnd` | `instoptfloat` | `instoptemfloat` | `instrangefloat` | `instswap` | `instswaption` | `intenvset` | `bondbyzero` | `cfbyzero` | `fixedbyzero` | `floatbyzero` | `intenvprice` | `intenvsens` | `swapbyzero` | `floatmargin` | `floatdiscmargin` | `hjmtimespec` | `hjmtree` | `hjmvolspec` | `bondbyhjm` | `capbyhjm` | `cfbyhjm` | `fixedbyhjm` | `floatbyhjm` | `floorbyhjm` | `hjmprice` | `hjmsens` | `mmktbyhjm` | `oasbyhjm` | `optbndbyhjm` | `optfloatbyhjm` | `optembndbyhjm` | `optemfloatbyhjm` | `rangefloatbyhjm` | `swapbyhjm` | `swaptionbyhjm` | `bdttimespec` | `bdttree` | `bdtvolspec` | `bdtprice` | `bdtsens` | `bondbybdt` | `capbybdt` | `cfbybdt` | `fixedbybdt` | `floatbybdt` | `floorbybdt` | `mmktbybdt` | `oasbybdt` | `optbndbybdt` | `optfloatbybdt` | `optembndbybdt` | `optemfloatbybdt` | `rangefloatbybdt` | `swapbybdt` | `swaptionbybdt` | `hwtimespec` | `hwtree` | `hwvolspec` | `bondbyhw` | `capbyhw` | `cfbyhw` | `fixedbyhw` | `floatbyhw` | `floorbyhw` | `hwcalbycap` | `hwcalbyfloor` | `hwprice` | `hwsens` | `oasbyhw` | `optbndbyhw` | `optfloatbyhw` | `optembndbyhw` | `optemfloatbyhw` | `rangefloatbyhw` |

`swapbyhw` | `swaptionbyhw` | `bktimespec` | `bktree` | `bkvolspec` | `bkprice` | `bksens` | `bondbybk` | `capbybk` | `cfbybk` | `fixedbybk` | `floatbybk` | `floorbybk` | `oasbybk` | `optbndbybk` | `optfloatbybk` | `optembndbybk` | `optemfloatbybk` | `rangefloatbybk` | `swapbybk` | `swaptionbybk` | `capbyblk` | `floorbyblk` | `swaptionbyblk` | `blackvolbysabr` | `optsensbysabr` | `agencyoas` | `agencyprice` | `bndfutimprepo` | `bndfutprice` | `convfactor` | `tfutbyprice` | `tfutbyyield` | `tfutimprepo` | `tfutpricebyrepo` | `tfutyieldbyrepo` | `capbylg2f` | `floorbylg2f` | `swaptionbylg2f` | `blackvolbyrebonato` | `hwcalbycap` | `hwcalbyfloor`

Related Examples

- “Overview of Interest-Rate Tree Models” on page 2-44
- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Graphical Representation of Trees” on page 2-219
- “Pricing Using Interest-Rate Tree Models” on page 2-81
- “Understanding Interest-Rate Tree Models” on page 2-66
- “Understanding the Interest-Rate Term Structure” on page 2-48

More About

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Work with Negative Interest Rates Using Functions

Interest-Rate Modeling Options for Negative Rates

Financial Instruments Toolbox computes prices for caps, floors, and swaptions when modeling for negative interest-rates using a Normal volatility model, shifted Black model, or shifted SABR model:

- Normal volatility model (Bachelier model) for interest-rate options to handle negative rates with the following:
  - `swaptionbynormal`
  - `capbynormal`
  - `floorbynormal`
  - `normalvolbysabr`
- Shifted Black model and the shifted SABR model for interest-rate options using an optional `Shift` argument to handle negative rates with the following:
  - `blackvolbysabr` (Shifted SABR)
  - `optsensbysabr` (Shifted SABR)
  - `swaptionbyblk` (Shifted Black)
  - `capbyblk` (Shifted Black)
  - `floorbyblk` (Shifted Black)
  - `capvolstrip` (Shifted Black)
  - `floorvolstrip` (Shifted Black)

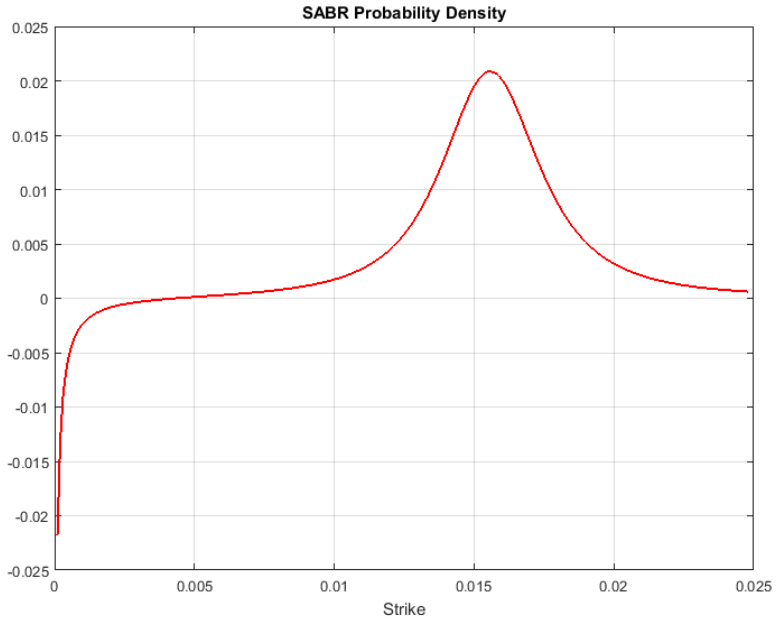
Modeling Negative Rates

The original authors of the SABR model provided a closed form approximation of the implied Black volatility in terms of the SABR model parameters (known as “Hagan’s formula”), so that the option price could be computed by inserting the computed SABR Black volatility into the Black formula:

$$Call(K,T) = Black_{call}(F,K,r,T,\sigma_{Black}(\alpha,\beta,\rho,\nu,F,K,T))$$

However, these methods started to break down with the introduction of negative interest rates, due to the assumption of the Black model that the underlying rates are lognormally distributed (and therefore cannot be negative).

In addition, even when the underlying rate is positive, the closed form approximation of the SABR implied Black volatility (Hagan et al., 2002) is known to become increasingly inaccurate as the strike approaches zero. Even without crossing the zero strike boundary, the implied probability density of the underlying rate at option expiry can become negative at low positive strikes, although probability densities clearly should not be negative:



Options with negative strikes cannot be represented by Black volatilities. To work around this problem, the market started to quote the cap, floor, and swaption prices also in terms of either Normal volatilities or Shifted Black volatilities. Instead of the Black model, both types of volatilities come from alternative models that allow negative rates.

Normal Model

The Normal volatilities are associated with the Normal model (also known as the Bachelier model):

$$dF = \sigma_{Normal} dW$$

where the underlying rates are assumed to be normally distributed. Unlike in a lognormal model (where rates have a lower bound), the rates in the Normal model can be both infinitely positive and infinitely negative.

Shifted Black

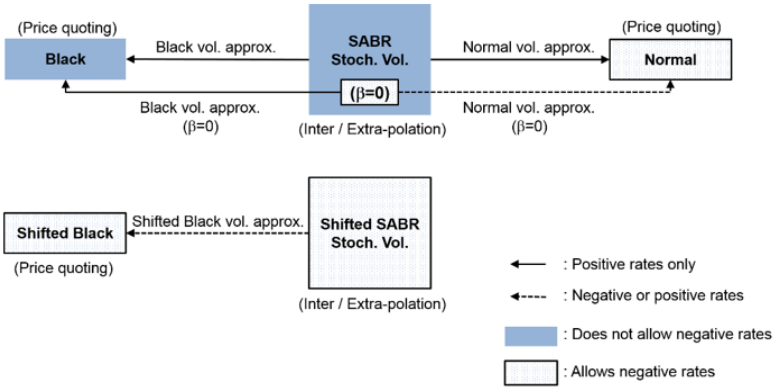
The Shifted Black volatilities are associated with the Shifted Black model (also known as “Displaced Diffusion” or “Shifted Lognormal” model):

$$dF = \sigma_{Shifted\_Black}(F + Shift) dW$$

The Shifted Black model is essentially the same as the Black model, except that it models the movements of (F + $Shift$) as the underlying asset, instead of F (where F is the forward swap rate in the case of swaptions, and the forward rate in the case of caplets and floorlets). So, the Shifted Black model allows negative rates, with a fixed negative lower bound defined by the amount of shift, that is, the zero lower bound of the Black model has been shifted.

Shifted SABR

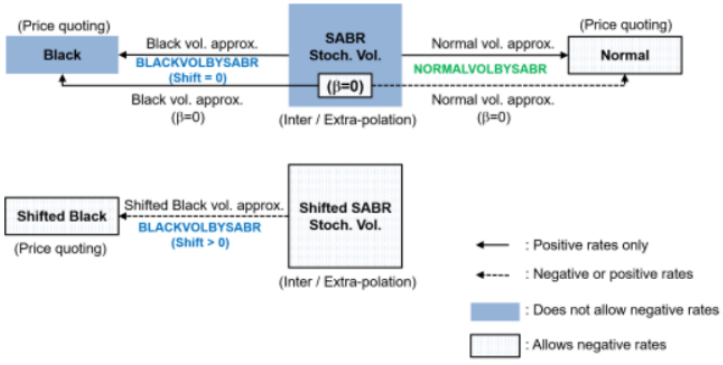
The introduction of negative interest rates also called for an update in the method for interpolating the volatilities quoted in the market. The following shows the connections between the volatilities and the SABR models:



As shown, the Black and Normal volatility approximations allow you to use the SABR model with the Black and Normal model option pricing formulas. However, although the Normal model itself allows negative rates and the SABR model has an implied Normal volatility approximation, the underlying dynamics of the SABR model do not allow negative rates, unless $\beta = 0$. In the Shifted SABR model, the Shifted Black volatility approximation can be used to allow negative rates with a fixed negative lower bound defined by the amount of shift.

Implied Normal Volatility and SABR

You can compute the implied Normal volatility in terms of the SABR model parameters, for either $\beta = 0$ (Normal SABR), or any other value of β allowed by the SABR model ($0 \le \beta \le 1$) using `normalvolbysabr`.



`normalvolbysabr`computes the implied Normal volatility σ_N in terms of the SABR model parameters. Using `normalvolbysabr` to compute σ_N, you can then you this with other functions for Normal model pricing (for example, `capbynormal`, `floorbynormal`, and `swaptionbyblk`).

See Also

`swaptionbynormal` | `capbynormal` | `floorbynormal` | `swaptionbyblk` | `capbyblk` | `floorbyblk` | `normalvolbysabr`

Related Examples

- “Price Swaptions with Negative Strikes Using the Shifted SABR Model” on page 2-26
- “Calibrate the SABR Model Using Normal (Bachelier) Volatilities with Negative Strikes” on page 2-163

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Work with Negative Interest Rates Using Objects” on page 2-22

Work with Negative Interest Rates Using Objects

Interest-Rate Modeling Options for Negative Rates

Financial Instruments Toolbox computes prices for a `Cap`, `Floor`, or `Swaption` instrument when modeling for negative interest-rates using a Normal volatility model, shifted Black model, or shifted SABR model:

- Normal volatility model (Bachelier model) for interest-rate options to handle negative rates with the following:
  - `Normal` model object
  - `Normal` pricer object
  - `SABR` model object
  - `SABR` pricer object
- Shifted Black model and the shifted SABR model for interest-rate options to handle negative rates with the following:
  - `Black` model object (Shifted Black model, specified by the `'Shift'` name-value argument set to a positive value.)
  - `Black` pricer object (Shifted Black model, specified by the `'Shift'` name-value argument set to a positive value.)
  - `SABR` model object (Shifted SABR model, specified by the `'VolatilityType'` name-value argument set to `"Black"` and the `'Shift'` name-value argument set to a positive value.)
  - `SABR` pricer object (Shifted SABR model, specified by the `'VolatilityType'` name-value argument set to `"Black"` and the `'Shift'` name-value argument set to a positive value.)

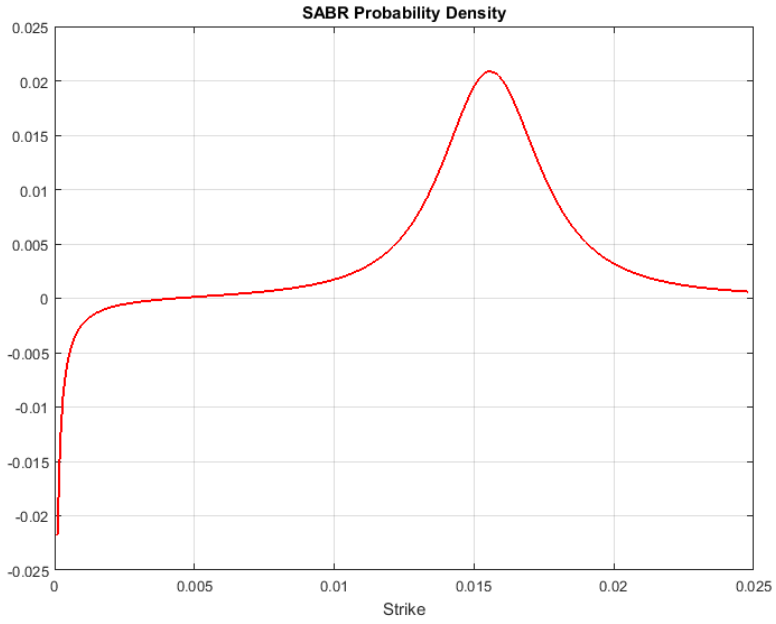
Modeling Negative Rates

The original authors of the SABR model provided a closed form approximation of the implied Black volatility in terms of the SABR model parameters (known as “Hagan’s formula”), so that the option price could be computed by inserting the computed SABR Black volatility into the Black formula:

$$Call(K,T) = Black_{call}(F,K,r,T,\sigma_{Black}(\alpha,\beta,\rho,\nu,F,K,T))$$

However, these methods started to break down with the introduction of negative interest rates, due to the assumption of the Black model that the underlying rates are lognormally distributed (and therefore cannot be negative).

In addition, even when the underlying rate is positive, the closed form approximation of the SABR implied Black volatility (Hagan et al., 2002) is known to become increasingly inaccurate as the strike approaches zero. Even without crossing the zero strike boundary, the implied probability density of the underlying rate at option expiry can become negative at low positive strikes, although probability densities clearly should not be negative:



Options with negative strikes cannot be represented by Black volatilities. To work around this problem, the market started to quote the cap, floor, and swaption prices also in terms of either Normal volatilities or Shifted Black volatilities. Instead of the Black model, both types of volatilities come from alternative models that allow negative rates.

Normal Model

The Normal volatilities are associated with the Normal model (also known as the Bachelier model):

$$dF = \sigma_{Normal} dW$$

where the underlying rates are assumed to be normally distributed. Unlike in a lognormal model (where rates have a lower bound), the rates in the Normal model can be both infinitely positive and infinitely negative.

Shifted Black

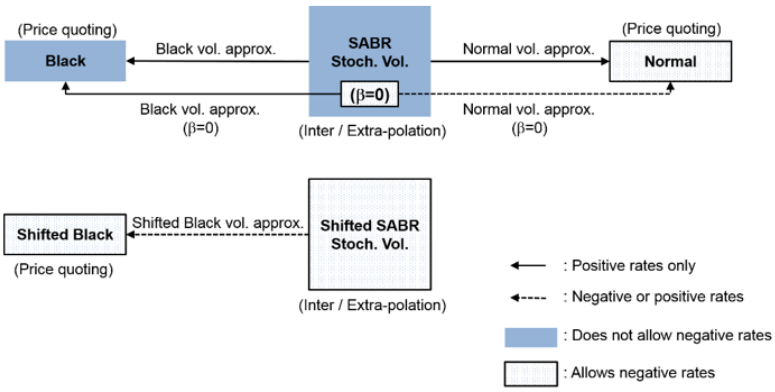
The Shifted Black volatilities are associated with the Shifted Black model (also known as “Displaced Diffusion” or “Shifted Lognormal” model):

$$dF = \sigma_{Shifted\_Black}(F + Shift)dW$$

The Shifted Black model is essentially the same as the Black model, except that it models the movements of (F + $Shift$) as the underlying asset, instead of F (where F is the forward swap rate in the case of swaptions, and the forward rate in the case of caplets and floorlets). So, the Shifted Black model allows negative rates, with a fixed negative lower bound defined by the amount of shift, that is, the zero lower bound of the Black model has been shifted.

Normal SABR and Shifted SABR

The introduction of negative interest rates also called for an update in the method for interpolating the volatilities quoted in the market. The following shows the connections between the volatilities and the SABR models:



As shown, the Black and Normal volatility approximations allow you to use the SABR model with the Black and Normal model option pricing formulas. However, although the Normal model itself allows negative rates and the SABR model has an implied Normal volatility approximation, the underlying dynamics of the SABR model do not allow negative rates, unless $\beta = 0$. When the β (Beta) parameter of the SABR model is set to zero, the model is a Normal SABR model, which allows computing the implied Normal volatilities for negative rates.

In the Shifted SABR model, the Shifted Black volatility approximation can be used to allow negative rates with a fixed negative lower bound defined by the amount of shift. This is achieved by setting the `'Shift'` name-value argument of the SABR model to a positive value.

Implied Volatilities and SABR

You can compute the implied volatilities in terms of the SABR model parameters, for either $\beta = 0$ (Normal SABR), or any other value of β allowed by the SABR model ($0 \le \beta \le 1$) using the `volatilities` function for the SABR analytic pricer.

The following three types of implied volatilities are supported by the SABR analytic pricer, and the type of implied volatilities computed by the `volatilities` function depends on the parameters of the SABR model when using the SABR analytic pricer:

- Implied Black volatilities — The SABR model `'VolatilityType'` name-value argument is set to `"Black"` and the `'Shift'` name-value argument is set to zero. Negative rates are not allowed.
- Implied Sifted Black volatilities — The SABR model `'VolatilityType'` name-value argument is set to `"Black"` and the `'Shift'` name-value argument is set to a positive value. Negative rates are allowed with lower bound defined by the amount of shift.
- Implied Normal (Bachelier) volatilities — The SABR model `'VolatilityType'` name-value argument is set to `"Black"` and the `'Shift'` name-value argument is set to zero. Negative rates are allowed when the `Beta` input argument is set to zero.

See Also

`Cap` | `Floor` | `Swaption`

Related Examples

- “Calibrate Shifted SABR Model Parameters for Swaption Instrument” on page 2-167
- “Calibrate SABR Model Using Normal (Bachelier) Volatilities with Analytic Pricer” on page 2-177

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53
- “Work with Negative Interest Rates Using Functions” on page 2-18

Price Swaptions with Negative Strikes Using the Shifted SABR Model

This example shows how to price swaptions with negative strikes by using the Shifted SABR model. The market Shifted Black volatilities are used to calibrate the Shifted SABR model parameters. The calibrated Shifted SABR model is then used to compute the Shifted Black volatilities for negative strikes.

The swaptions with negative strikes are then priced using the computed Shifted Black volatilities and the `swaptionbyblk` function with the `'Shift'` parameter set to the prespecified shift. Similarly, Shifted SABR Greeks can be computed by using the `optsensbysabr` function by setting the `'Shift'` parameter. Finally, from the swaption prices, the probability density of the underlying asset is computed to show that the swaption prices imply positive probability densities for some negative strikes.

Load the market data.

First, load the market interest rates and swaption volatility data. The market swaption volatilities are quoted in terms of Shifted Black volatilities with a `0.8` percent shift.

Define `RateSpec`.

```
ValuationDate = '5-Apr-2016';
EndDates = datemnth(ValuationDate,[1 2 3 6 9 12*[1 2 3 4 5 6 7 8 9 10 12]])';
ZeroRates = [-0.34 -0.29 -0.25 -0.13 -0.07 -0.02 0.010 0.025 ...
    0.031 0.040 0.052 0.090 0.190 0.290 0.410 0.520]'/100;
Compounding = 1;
RateSpec = intenvset('ValuationDate',ValuationDate,'StartDates',ValuationDate, ...
'EndDates',EndDates,'Rates',ZeroRates,'Compounding',Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [16x1 double]
            Rates: [16x1 double]
         EndTimes: [16x1 double]
       StartTimes: [16x1 double]
          EndDates: [16x1 double]
       StartDates: 736425
    ValuationDate: 736425
            Basis: 0
     EndMonthRule: 1
```

Define the swaption.

```
SwaptionSettle = '5-Apr-2016';
SwaptionExerciseDate = '5-Apr-2017';
SwapMaturity = '5-Apr-2022';
Reset = 1;
OptSpec = 'call';
TimeToExercise = yearfrac(SwaptionSettle,SwaptionExerciseDate);
```

Use `swapbyzero` to compute the forward swap rate.

```
LegRate = [NaN 0];  % To compute the forward swap rate, set the fixed rate to NaN.
[~, CurrentForwardValue] = swapbyzero(RateSpec,LegRate,SwaptionSettle,SwapMaturity,...
'StartDate',SwaptionExerciseDate)
```

```
CurrentForwardValue = 6.6384e-04
```

Specify amount of shift in decimals for Shifted Black and Shifted SABR models.

```
Shift = 0.008;  % 0.8 percent shift
```

Load the market implied Shifted Black volatility data for swaptions.

```
MarketShiftedBlackVolatilities = [21.1; 15.3; 14.0; 14.6; 16.0; 17.7; 19.8; 23.9; 26.2]/100;
StrikeGrid = [-0.5; -0.25; -0.125; 0; 0.125; 0.25; 0.5; 1.0; 1.5]/100;
MarketStrikes = CurrentForwardValue + StrikeGrid;
ATMShiftedBlackVolatility = MarketShiftedBlackVolatilities(StrikeGrid==0);
```

Calibrate the Shifted SABR model parameters.

To better represent the market at-the-money volatility, the `Alpha` parameter value is implied by the market at-the-money volatility. This is similar to the "Method 2" in “Calibrate the SABR Model” on page 2-33. However, note the addition of `Shift` to `CurrentForwardValue` and the use of the `'Shift'` parameter with `blackvolbysabr`. The `Beta` parameter is predetermined at `0.5`.

```
Beta = 0.5;
```

This function solves the Shifted SABR at-the-money volatility equation as a polynomial of `Alpha`. Note the addition of `Shift` to `CurrentForwardValue`.

```
alpharoots = @(Rho,Nu) roots([...
    (1 - Beta)^2*TimeToExercise/24/(CurrentForwardValue + Shift)^(2 - 2*Beta) ...
    Rho*Beta*Nu*TimeToExercise/4/(CurrentForwardValue + Shift)^(1 - Beta) ...
    (1 + (2 - 3*Rho^2)*Nu^2*TimeToExercise/24) ...
    -ATMShiftedBlackVolatility*(CurrentForwardValue + Shift)^(1 - Beta)]);
```

This function converts at-the-money volatility into `Alpha` by picking the smallest positive real root.

```
atmVol2ShiftedSabrAlpha = @(Rho,Nu) min(real(arrayfun(@(x) ...
    x*(x>0) + realmax*(x<0 || abs(imag(x))>1e-6), alpharoots(Rho,Nu))));
```

Fit `Rho` and `Nu` (while converting at-the-money volatility into `Alpha`). Note the `'Shift'` parameter of `blackvolbysabr` is set to the prespecified shift.

```
objFun = @(X) MarketShiftedBlackVolatilities - ...
    blackvolbysabr(atmVol2ShiftedSabrAlpha(X(1), X(2)), ...
    Beta, X(1), X(2), SwaptionSettle, SwaptionExerciseDate, CurrentForwardValue, ...
    MarketStrikes, 'Shift', Shift);

options = optimoptions('lsqnonlin','Display','none');
X = lsqnonlin(objFun, [0 0.5], [-1 0], [1 Inf], options);
Rho = X(1);
Nu = X(2);
```

Get the final `Alpha` from the calibrated parameters.

```
Alpha = atmVol2ShiftedSabrAlpha(Rho, Nu)
```

```
Alpha = 0.0133
```

Show the calibrated Shifted SABR parameters.

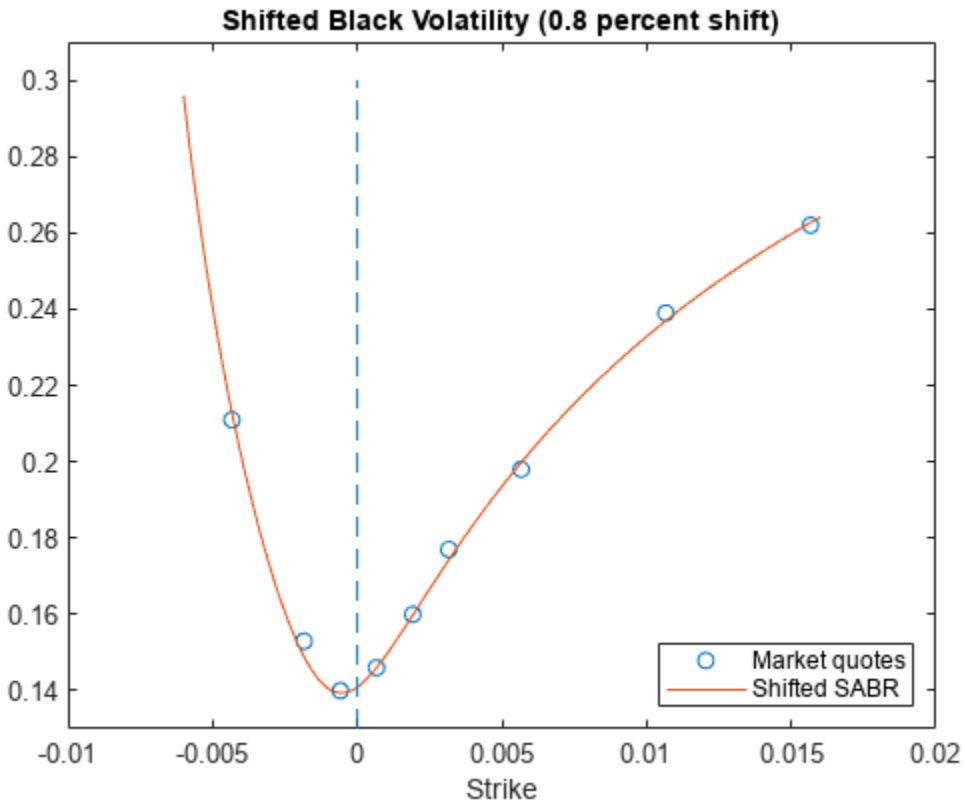
```
CalibratedPrameters = array2table([Shift Alpha Beta Rho Nu],...
    'VariableNames',{'Shift' 'Alpha' 'Beta' 'Rho' 'Nu'},...
    'RowNames',{'1Y into 5Y'})
```

```
CalibratedPrameters=1×5 table
                  Shift     Alpha      Beta      Rho        Nu
                  _____    ________    ____    _______    _______

    1Y into 5Y    0.008    0.013345    0.5     0.46698    0.49816
```

Compute the swaption volatilities using the calibrated Shifted SABR model.

Use `blackvolbysabr` with the `'Shift'` parameter.

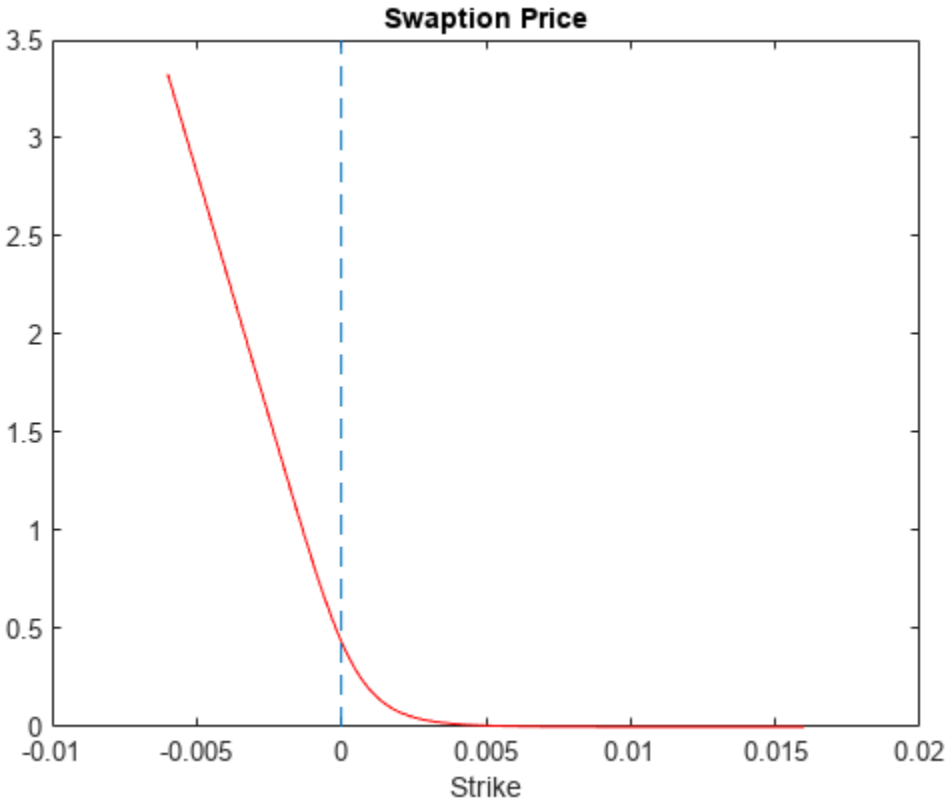
```
Strikes = (-0.6:0.01:1.6)'/100; % Include negative strikes.
SABRShiftedBlackVolatilities = blackvolbysabr(Alpha, Beta, Rho, Nu, SwaptionSettle, ...
    SwaptionExerciseDate, CurrentForwardValue, Strikes, 'Shift', Shift);

figure;
plot(MarketStrikes, MarketShiftedBlackVolatilities, 'o', ...
    Strikes, SABRShiftedBlackVolatilities);
h = gca;
line([0,0],[min(h.YLim),max(h.YLim)],'LineStyle','--');
ylim([0.13 0.31])
xlabel('Strike');
legend('Market quotes','Shifted SABR', 'location', 'southeast');
title (['Shifted Black Volatility (',num2str(Shift*100),' percent shift)']);
```



Price the swaptions, including those with negative strikes.

Use `swaptionbyblk` with the `'Shift'` parameter to compute swaption prices using the Shifted Black model.

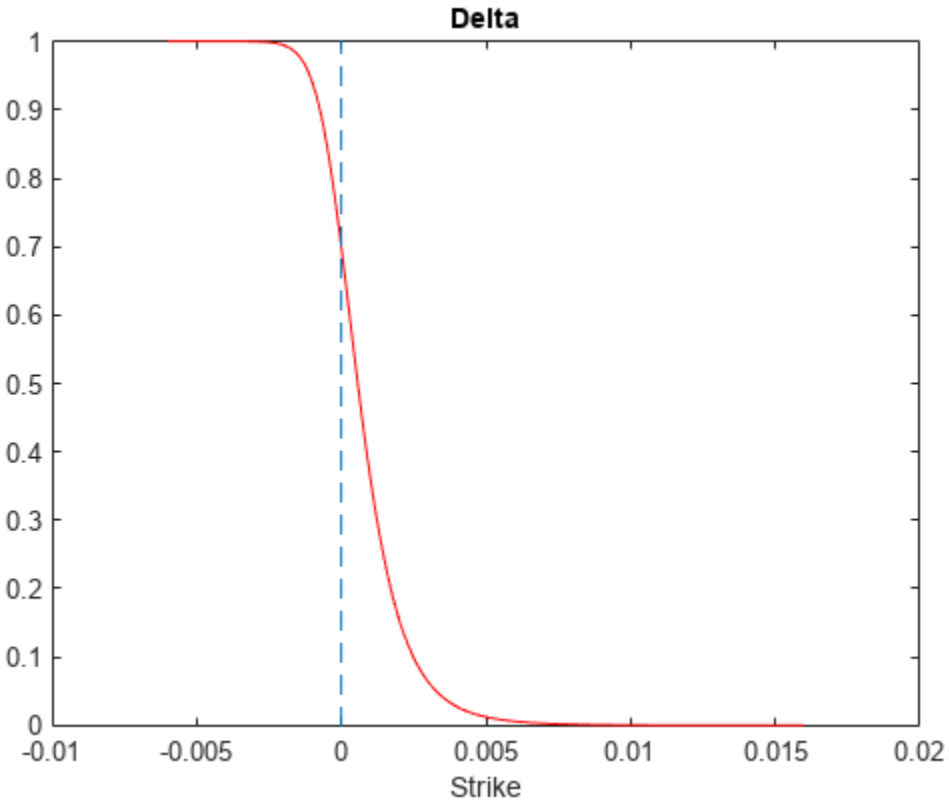
```
SwaptionPrices = swaptionbyblk(RateSpec, OptSpec, Strikes, SwaptionSettle, SwaptionExerciseDate,
    SwapMaturity, SABRShiftedBlackVolatilities, 'Reset', Reset, 'Shift', Shift);
figure;
plot(Strikes, SwaptionPrices, 'r');
h = gca;
line([0,0],[min(h.YLim),max(h.YLim)],'LineStyle','--');
xlabel('Strike');
title ('Swaption Price');
```



Compute Shifted SABR Delta.

Use `optsensbysabr` with the `'Shift'` parameter to compute `Delta` using the Shifted SABR model.

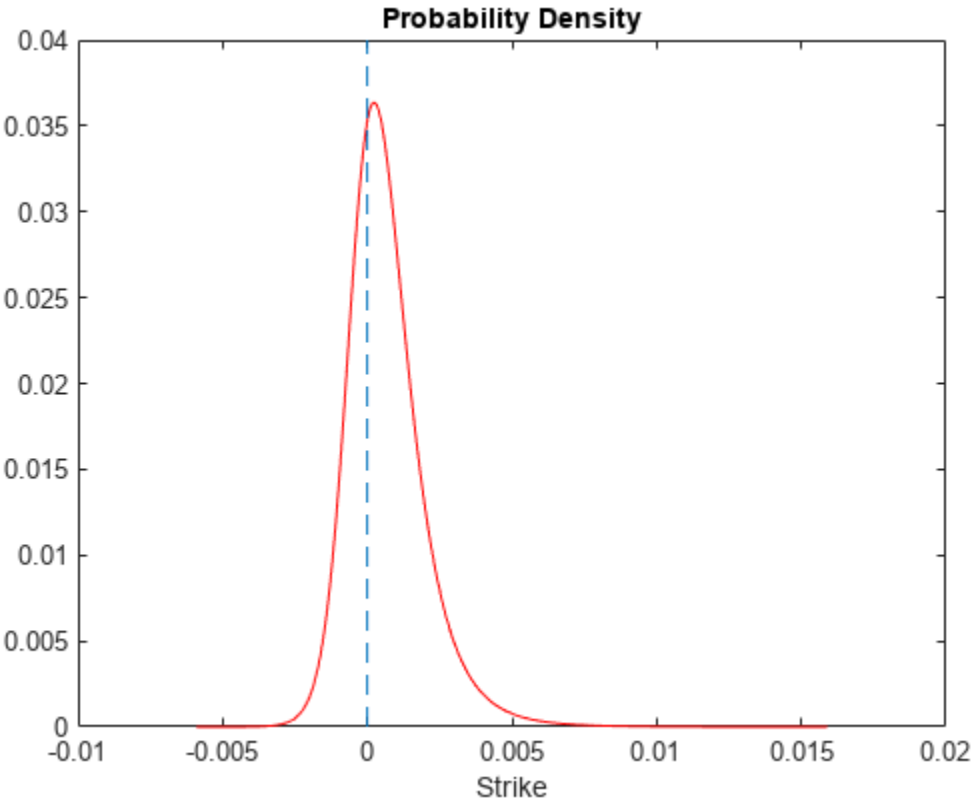
```
ShiftedSABRDelta = optsensbysabr(RateSpec, Alpha, Beta, Rho, Nu, SwaptionSettle, ...
SwaptionExerciseDate, CurrentForwardValue, Strikes, OptSpec, 'Shift', Shift);

figure;
plot(Strikes,ShiftedSABRDelta,'r-');
ylim([-0.002 1.002]);
h = gca;
line([0,0],[min(h.YLim),max(h.YLim)],'LineStyle','--');
xlabel('Strike');
title ('Delta');
```



Compute the probability density.

The risk-neutral probability density of the terminal underlying asset prices can be approximated as the second derivative of swaption prices with respect to strike (Breeden and Litzenberger, 1978). As can be seen in the plot below, the computed probability density is positive for some negative rates above -0.8 percent (the lower bound determined by `'Shift'`).

```
NumGrids = length(Strikes);
ProbDensity = zeros(NumGrids-2,1);
dStrike = mean(diff(Strikes));

for k = 2:(NumGrids-1)
    ProbDensity(k-1) = (SwaptionPrices(k-1) - 2*SwaptionPrices(k) + SwaptionPrices(k+1))/dStrike^
end

ProbDensity = ProbDensity./sum(ProbDensity);
ProbStrikes = Strikes(2:end-1);

figure;
plot(ProbStrikes,ProbDensity,'r-');
h = gca;
line([0,0],[min(h.YLim),max(h.YLim)],'LineStyle','--');
xlabel('Strike');
title ('Probability Density');
```



References

Hagan, P. S., Kumar, D., Lesniewski, A. S. and Woodward, D. E. "Managing Smile Risk." *Wilmott Magazine*. 2002.

Kienitz, J. *Interest Rate Derivatives Explained*. Vol. 1. Palgrave MacMillan, 2014.

Breeden, D. T. and Litzenberger, R. H. "Prices of State-Contingent Claims Implicit in Option Prices." *Journal Business*. Vol. 51. 1978.

See Also

`optsensbysabr` | `capbyblk` | `floorbyblk` | `capvolstrip` | `floorvolstrip` | `swaptionbyblk` | `capbynormal` | `floorbynormal` | `swaptionbynormal`

Related Examples

- “Calibrate the SABR Model” on page 2-33
- “Price a Swaption Using the SABR Model” on page 2-38

More About

- “Work with Negative Interest Rates Using Functions” on page 2-18
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Calibrate the SABR Model

This example shows how to use two different methods to calibrate the SABR stochastic volatility model from market implied Black volatilities. Both approaches use `blackvolbysabr`.

In this section...

Load Market Implied Black Volatility Data

This example shows how to set up hypothetical market implied Black volatilities for European swaptions over a range of strikes before calibration. The swaptions expire in three years from the `Settle` date and have 10-year swaps as the underlying instrument. The rates are expressed in decimals. (Changing the units affect the numerical value and interpretation of the `Alpha` input parameter to the function `blackvolbysabr`.)

Load the market implied Black volatility data for swaptions expiring in three years.

```
Settle = '12-Jun-2013';
ExerciseDate = '12-Jun-2016';

MarketStrikes = [2.0 2.5 3.0 3.5 4.0 4.5 5.0]'/100;
MarketVolatilities = [45.6 41.6 37.9 36.6 37.8 39.2 40.0]'/100;
```

At the time of `Settle`, define the underlying forward rate and the at-the-money volatility.

```
CurrentForwardValue = MarketStrikes(4)
ATMVolatility = MarketVolatilities(4)
```

```
CurrentForwardValue =

    0.0350


ATMVolatility =

    0.3660
```

Method 1: Calibrate Alpha, Rho, and Nu Directly

This example shows how to calibrate the `Alpha`, `Rho`, and `Nu` input parameters directly. The value of `Beta` is predetermined either by fitting historical market volatility data or by choosing a value deemed appropriate for that market [1].

Define the predetermined `Beta`.

```
Beta1 = 0.5;
```

After fixing the value of β (`Beta`), the parameters α (`Alpha`), ρ (`Rho`), and υ(`Nu`) are all fitted directly. The Optimization Toolbox™ function `lsqnonlin` generates the parameter values that minimize the squared error between the market volatilities and the volatilities computed by `blackvolbysabr`.

```
% Calibrate Alpha, Rho, and Nu
objFun = @(X) MarketVolatilities - ...
    blackvolbysabr(X(1), Beta1, X(2), X(3), Settle, ...
    ExerciseDate, CurrentForwardValue, MarketStrikes);

X = lsqnonlin(objFun, [0.5 0 0.5], [0 -1 0], [Inf 1 Inf]);

Alpha1 = X(1);
Rho1 = X(2);
Nu1 = X(3);
```

```
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.
```

Method 2: Calibrate Rho and Nu by Implying Alpha from At-The-Money Volatility

This example shows how to use an alternative calibration method where the value of β (`Beta`) is again predetermined as in Method 1.

Define the predetermined `Beta`.

```
Beta2 = 0.5;
```

However, after fixing the value of β (`Beta`), the parameters ρ (`Rho`), and υ (`Nu`) are fitted directly while α (`Alpha`) is implied from the market at-the-money volatility. Models calibrated using this method produce at-the-money volatilities that are equal to market quotes. This approach is widely used in swaptions, where at-the-money volatilities are quoted most frequently and are important to match. To imply α (`Alpha`) from market at-the-money volatility (σ_{ATM}), the following cubic polynomial is solved for α (`Alpha`), and the smallest positive real root is selected [2].

$$\frac{(1-\beta)^2 T}{24F^{(2-2\beta)}}\alpha^3 + \frac{\rho\beta\upsilon T}{4F^{(1-\beta)}}\alpha^2 + \left(1 + \frac{2-3\rho^2}{24}\upsilon^2 T\right)\alpha - \sigma_{ATM}F^{(1-\beta)} = 0$$

where:

- F is the current forward value.
- T is the year fraction to maturity.

To accomplish this, define an anonymous function as:

```
% Year fraction from Settle to option maturity
T = yearfrac(Settle, ExerciseDate, 1);

% This function solves the SABR at-the-money volatility equation as a
% polynomial of Alpha
alpharoots = @(Rho,Nu) roots([...
    (1 - Beta2)^2*T/24/CurrentForwardValue^(2 - 2*Beta2) ...
    Rho*Beta2*Nu*T/4/CurrentForwardValue^(1 - Beta2) ...
```

```
    (1 + (2 - 3*Rho^2)*Nu^2*T/24) ...
    -ATMVolatility*CurrentForwardValue^(1 - Beta2)]);

% This function converts at-the-money volatility into Alpha by picking the
% smallest positive real root
atmVol2SabrAlpha = @(Rho,Nu) min(real(arrayfun(@(x) ...
    x*(x>0) + realmax*(x<0 || abs(imag(x))>1e-6), alpharoots(Rho,Nu))));
```

The function `atmVol2SabrAlpha` converts at-the-money volatility into α (`Alpha`) for a given set of ρ (`Rho`) and υ (`Nu`). This function is then used in the objective function to fit parameters ρ (`Rho`) and υ (`Nu`).

```
% Calibrate Rho and Nu (while converting at-the-money volatility into Alpha
% using atmVol2SabrAlpha)
objFun = @(X) MarketVolatilities - ...
    blackvolbysabr(atmVol2SabrAlpha(X(1), X(2)), ...
    Beta2, X(1), X(2), Settle, ExerciseDate, CurrentForwardValue, ...
    MarketStrikes);

X = lsqnonlin(objFun, [0 0.5], [-1 0], [1 Inf]);

Rho2 = X(1);
Nu2 = X(2);
```

```
Local minimum found.

Optimization completed because the size of the gradient is less than
the default value of the function tolerance.
```

The calibrated parameter α (`Alpha`) is computed using the calibrated parameters ρ (`Rho`) and υ (`Nu`).

```
% Obtain final Alpha from at-the-money volatility using calibrated parameters
Alpha2 = atmVol2SabrAlpha(Rho2, Nu2);

% Display calibrated parameters
C = {Alpha1 Beta1 Rho1 Nu1;Alpha2 Beta2 Rho2 Nu2};
CalibratedPrameters = cell2table(C,...
    'VariableNames',{'Alpha' 'Beta' 'Rho' 'Nu'},...
    'RowNames',{'Method 1';'Method 2'})
```

```
CalibratedPrameters =

                Alpha      Beta      Rho        Nu
               ________    ____    _______    _______

    Method 1    0.060277    0.5      0.2097    0.75091
    Method 2    0.058484    0.5     0.20568    0.79647
```

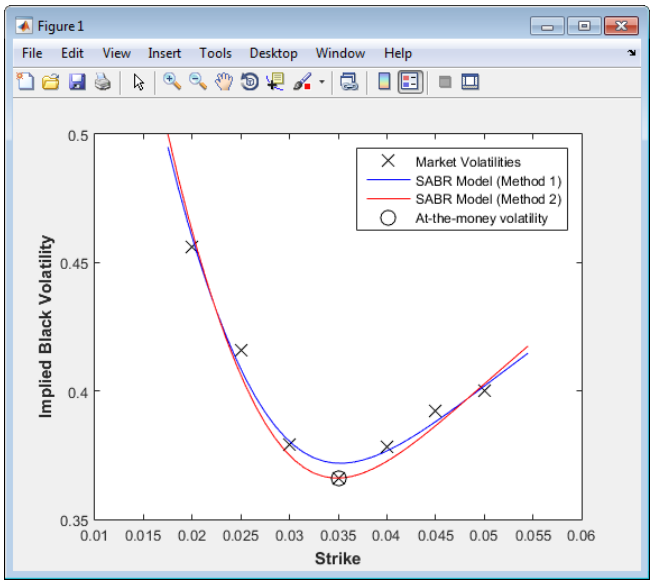
Use the Calibrated Models

This example shows how to use the calibrated models to compute new volatilities at any strike value.

Compute volatilities for models calibrated using Method 1 and Method 2 and plot the results.

```
PlottingStrikes = (1.75:0.1:5.50)'/100;

% Compute volatilities for model calibrated by Method 1
ComputedVols1 = blackvolbysabr(Alpha1, Beta1, Rho1, Nu1, Settle, ...
```

```
    ExerciseDate, CurrentForwardValue, PlottingStrikes);

% Compute volatilities for model calibrated by Method 2
ComputedVols2 = blackvolbysabr(Alpha2, Beta2, Rho2, Nu2, Settle, ...
    ExerciseDate, CurrentForwardValue, PlottingStrikes);

figure;
plot(MarketStrikes,MarketVolatilities,'xk',...
    PlottingStrikes,ComputedVols1,'b', ...
    PlottingStrikes,ComputedVols2,'r', ...
    CurrentForwardValue,ATMVolatility,'ok',...
    'MarkerSize',10);
xlim([0.01 0.06]);
ylim([0.35 0.5]);
xlabel('Strike', 'FontWeight', 'bold');
ylabel('Implied Black Volatility', 'FontWeight', 'bold');
legend('Market Volatilities', 'SABR Model (Method 1)',...
    'SABR Model (Method 2)', 'At-the-money volatility');
```



The model calibrated using Method 2 reproduces the market at-the-money volatility (marked with a circle) exactly.

References

[1] Hagan, P. S., Kumar, D., Lesniewski, A. S. and Woodward, D. E., *Managing smile risk*, Wilmott Magazine, 2002.

[2] West, G., “Calibration of the SABR Model in Illiquid Markets,” *Applied Mathematical Finance*, 12(4), pp. 371–385, 2004.

See Also

`blackvolbysabr` | `optsensbysabr` | `swaptionbyblk`

Related Examples

- “Price a Swaption Using the SABR Model” on page 2-38

More About

- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Price a Swaption Using the SABR Model

This example shows how to price a swaption using the SABR model. First, a swaption volatility surface is constructed from market volatilities. This is done by calibrating the SABR model parameters separately for each swaption maturity. The swaption price is then computed by using the implied Black volatility on the surface as an input to the `swaptionbyblk` function.

Step 1. Load market swaption volatility data.

Load the market implied Black volatility data for swaptions.

```
Settle = '12-Jun-2013';
ExerciseDates = {'12-Sep-2013';'12-Jun-2014';'12-Jun-2015';...
    '12-Jun-2016';'12-Jun-2017';'12-Jun-2018';'12-Jun-2020';...
    '12-Jun-2023'};

YearsToExercise = yearfrac(Settle, ExerciseDates, 1);
NumMaturities = length(YearsToExercise);

MarketVolatilities = [ ...
   57.6 53.7 49.4 45.6 44.1 41.1 35.2 32.0
   46.6 46.9 44.8 41.6 39.8 37.4 33.4 31.0
   35.9 39.3 39.6 37.9 37.2 34.7 30.5 28.9
   34.1 36.5 37.8 36.6 35.0 31.9 28.1 26.6
   41.0 41.3 39.5 37.8 36.0 32.6 29.0 26.0
   45.8 43.4 41.9 39.2 36.9 33.2 29.6 26.3
   50.3 46.9 44.0 40.0 37.5 33.8 30.2 27.3]/100;

MarketStrikes = [ ...
  1.00 1.25 1.68 2.00 2.26 2.41 2.58 2.62;
  1.50 1.75 2.18 2.50 2.76 2.91 3.08 3.12;
  2.00 2.25 2.68 3.00 3.26 3.41 3.58 3.62;
  2.50 2.75 3.18 3.50 3.76 3.91 4.08 4.12;
  3.00 3.25 3.68 4.00 4.26 4.41 4.58 4.62;
  3.50 3.75 4.18 4.50 4.76 4.91 5.08 5.12;
  4.00 4.25 4.68 5.00 5.26 5.41 5.58 5.62]/100;

CurrentForwardValues = MarketStrikes(4,:)
```

```
CurrentForwardValues = 1×8

    0.0250    0.0275    0.0318    0.0350    0.0376    0.0391    0.0408    0.0412
```

```
ATMVolatilities = MarketVolatilities(4,:)
```

```
ATMVolatilities = 1×8

    0.3410    0.3650    0.3780    0.3660    0.3500    0.3190    0.2810    0.2660
```

The current underlying forward rates and the corresponding at-the-money volatilities across the eight swaption maturities are represented in the fourth rows of the two matrices.

Step 2. Calibrate the SABR model parameters for each swaption maturity.

Using a model implemented in the function `blackvolbysabr`, a static SABR model, where the model parameters are assumed to be constant with respect to time, the parameters are calibrated separately for each swaption maturity (years to exercise) in a `for` loop. To better represent market at-the-money volatilities, the `Alpha` parameter values are implied by the market at-the-money volatilities (see "Method 2" for “Calibrate the SABR Model” on page 2-33).

Define the predetermined `Beta`, calibrate SABR model parameters for each swaption maturity and display calibrated parameters in a table.

```
Beta = 0.5;
Betas = repmat(Beta, NumMaturities, 1);
Alphas = zeros(NumMaturities, 1);
Rhos = zeros(NumMaturities, 1);
Nus = zeros(NumMaturities, 1);

options = optimoptions('lsqnonlin','Display','none');

for k = 1:NumMaturities
    % This function solves the SABR at-the-money volatility equation as a
    % polynomial of Alpha
    alpharoots = @(Rho,Nu) roots([...
        (1 - Beta)^2*YearsToExercise(k)/24/CurrentForwardValues(k)^(2 - 2*Beta) ...
        Rho*Beta*Nu*YearsToExercise(k)/4/CurrentForwardValues(k)^(1 - Beta) ...
        (1 + (2 - 3*Rho^2)*Nu^2*YearsToExercise(k)/24) ...
        -ATMVolatilities(k)*CurrentForwardValues(k)^(1 - Beta)]);

    % This function converts at-the-money volatility into Alpha by picking the
    % smallest positive real root
    atmVol2SabrAlpha = @(Rho,Nu) min(real(arrayfun(@(x) ...
        x*(x>0) + realmax*(x<0 || abs(imag(x))>1e-6), alpharoots(Rho,Nu))));

    % Fit Rho and Nu (while converting at-the-money volatility into Alpha)
    objFun = @(X) MarketVolatilities(:,k) - ...
        blackvolbysabr(atmVol2SabrAlpha(X(1), X(2)), ...
        Beta, X(1), X(2), Settle, ExerciseDates(k), CurrentForwardValues(k), ...
        MarketStrikes(:,k));

    X = lsqnonlin(objFun, [0 0.5], [-1 0], [1 Inf], options);
    Rho = X(1);
    Nu = X(2);

    % Get final Alpha from the calibrated parameters
    Alpha = atmVol2SabrAlpha(Rho, Nu);

    Alphas(k) = Alpha;
    Rhos(k) = Rho;
    Nus(k) = Nu;
end

CalibratedPrameters = array2table([Alphas Betas Rhos Nus],...
    'VariableNames',{'Alpha' 'Beta' 'Rho' 'Nu'},...
    'RowNames',{'3M into 10Y';'1Y into 10Y';...
    '2Y into 10Y';'3Y into 10Y';'4Y into 10Y';...
    '5Y into 10Y';'7Y into 10Y';'10Y into 10Y'})
```

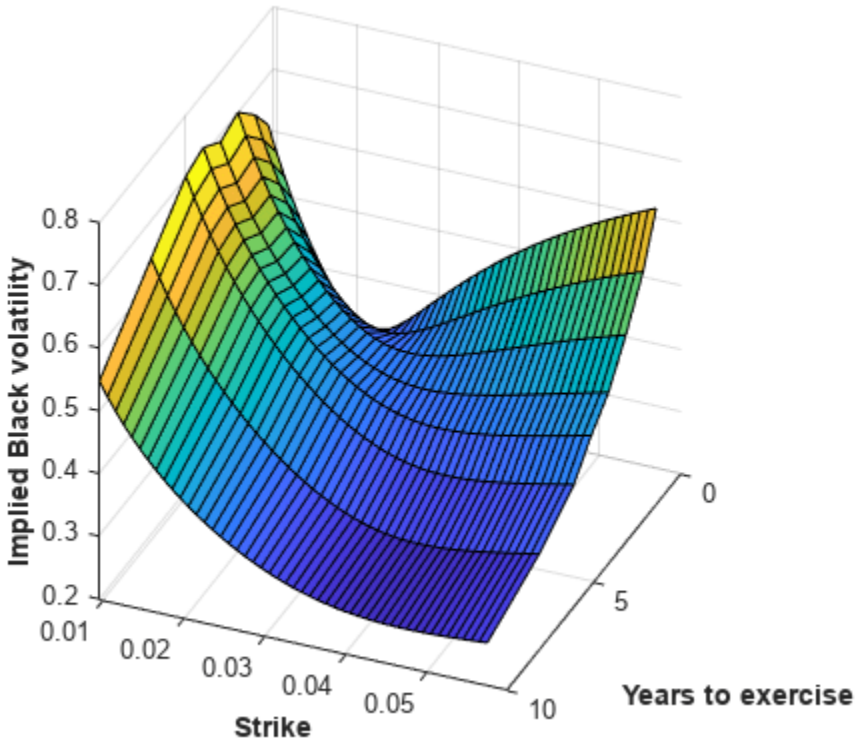
```
CalibratedPrameters=8×4 table
                      Alpha      Beta      Rho         Nu
                    _________    ____    ________    _______

    3M into 10Y      0.051947    0.5      0.39572     1.4146
    1Y into 10Y      0.054697    0.5       0.2955     1.1257
    2Y into 10Y      0.058433    0.5      0.24175    0.93463
    3Y into 10Y      0.058484    0.5      0.20568    0.79647
    4Y into 10Y      0.056054    0.5      0.13685    0.76993
    5Y into 10Y      0.051072    0.5     0.060285    0.73595
    7Y into 10Y       0.04475    0.5     0.083385    0.66341
    10Y into 10Y     0.044548    0.5      0.02261    0.49487
```

Step 3. Construct a volatility surface.

Use the calibrated model to compute new volatilities at any strike value to produce a smooth smile for a given maturity. This can be repeated for each maturity to form a volatility surface

Compute volatilities using the calibrated models for each maturity and plot the volatility surface.

```
PlottingStrikes = (0.95:0.1:5.8)'/100;
ComputedVols = zeros(length(PlottingStrikes), NumMaturities);

for k = 1:NumMaturities
ComputedVols(:,k) = blackvolbysabr(Alphas(k), Betas(k), Rhos(k), Nus(k), Settle, ...
    ExerciseDates(k), CurrentForwardValues(k), PlottingStrikes);
end

figure;
surf(YearsToExercise, PlottingStrikes, ComputedVols);
xlim([0 10]); ylim([0.0095 0.06]); zlim([0.2 0.8]);
view(113,32);
set(gca, 'Position', [0.13 0.11 0.775 0.815], ...
    'PlotBoxAspectRatioMode', 'manual');
xlabel('Years to exercise', 'Fontweight', 'bold');
ylabel('Strike', 'Fontweight', 'bold');
zlabel('Implied Black volatility', 'Fontweight', 'bold');
```



Note, in this volatility surface, the smiles tend to get flatter for longer swaption maturities (years to exercise). This is consistent with the `Nu` parameter values tending to decrease with swaption maturity, as shown previously in the table for `CalibratedPrameters`.

Step 4. Use `swaptionbyblk` to price a swaption.

Use the volatility surface to price a swaption that matures in five years. Define a swaption (for a 10-year swap) that matures in five years and use the interest-rate term structure at the time of the swaption `Settle` date to define the `RateSpec`. Use the `RateSpec` to compute the current forward swap rate using the `swapbyzero` function. Compute the SABR implied Black volatility for this swaption using the `blackvolbysabr` function (and it is marked with a red arrow in the figure that follows). Price the swaption using the SABR implied Black volatility as an input to the `swaptionbyblk` function.

```
% Define the swaption
SwaptionSettle = '12-Jun-2013';
SwaptionExerciseDate = '12-Jun-2018';
SwapMaturity = '12-Jun-2028';
Reset = 1;
OptSpec = 'call';
Strike = 0.0263;

% Define RateSpec
ValuationDate = '12-Jun-2013';
EndDates = {'12-Jul-2013';'12-Sep-2013';'12-Dec-2013';'12-Jun-2014';...
    '12-Jun-2015';'12-Jun-2016';'12-Jun-2017';'12-Jun-2018';...
    '12-Jun-2019';'12-Jun-2020';'12-Jun-2021';'12-Jun-2022';...
```

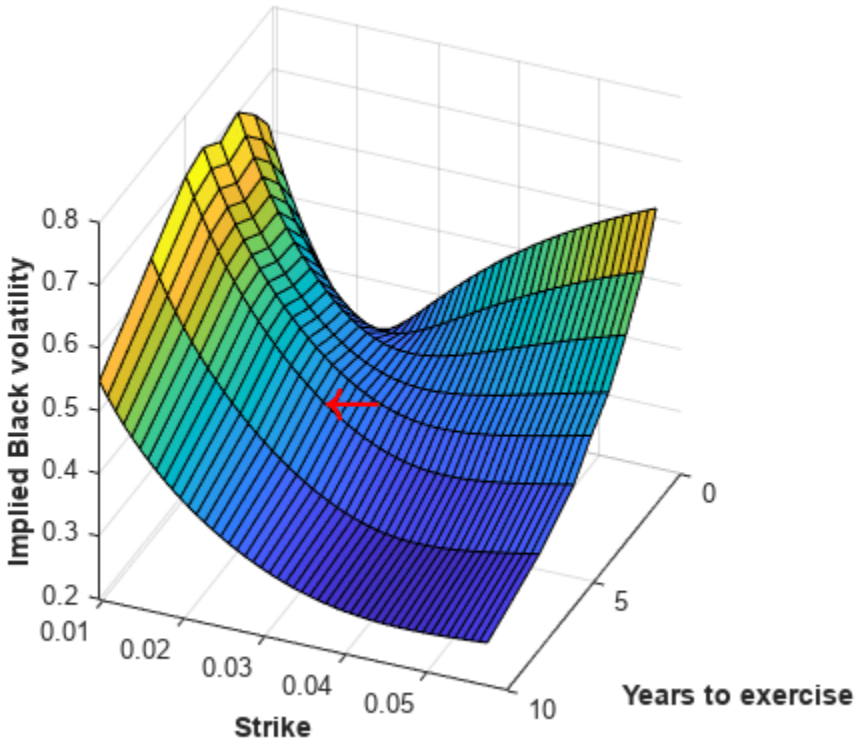
```
    '12-Jun-2023';'12-Jun-2025';'12-Jun-2028';'12-Jun-2033'};
Rates = [0.2 0.3 0.4 0.7 0.5 0.7 1.0 1.4 1.7 1.9 ...
    2.1 2.3 2.5 2.8 3.1 3.3]'/100;
Compounding = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [16x1 double]
            Rates: [16x1 double]
         EndTimes: [16x1 double]
       StartTimes: [16x1 double]
         EndDates: [16x1 double]
       StartDates: 735397
    ValuationDate: 735397
            Basis: 0
     EndMonthRule: 1
```

```
% Use swapbyzero
LegRate = [NaN 0]; % To compute the forward swap rate, set the coupon rate to NaN.
[~, CurrentForwardSwapRate] = swapbyzero(RateSpec, LegRate, SwaptionSettle, SwapMaturity,...
'StartDate', SwaptionExerciseDate);

% Use blackvolbysabr
SABRBlackVolatility = blackvolbysabr(Alphas(6), Betas(6), Rhos(6), Nus(6), SwaptionSettle, ...
    SwaptionExerciseDate, CurrentForwardSwapRate, Strike)
```

```
SABRBlackVolatility = 0.3932
```

```
text (YearsToExercise(6), Strike, SABRBlackVolatility, '\leftarrow',...
    'Color', 'r', 'FontWeight', 'bold', 'FontSize', 22);
```




```
% Use swaptionbyblk
Price = swaptionbyblk(RateSpec, OptSpec, Strike, SwaptionSettle, SwaptionExerciseDate, ...
SwapMaturity, SABRBlackVolatility, 'Reset', Reset)

Price = 14.2403
```

[1] Hagan, P. S., Kumar, D., Lesniewski, A. S. and Woodward, D. E., “Managing Smile Risk,” Wilmott Magazine, 2002.

[2] West, G., “Calibration of the SABR Model in Illiquid Markets,” *Applied Mathematical Finance*, 12(4), pp. 371–385, 2004.

See Also

`blackvolbysabr` | `swaptionbyblk` | `swapbyzero`

Related Examples

- “Calibrate the SABR Model” on page 2-33

More About

- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Overview of Interest-Rate Tree Models

In this section...

Interest-Rate Modeling

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

- The interest-rate term structure

  This model uses sets of zero-coupon bonds to predict changes in interest rates. A zero-coupon bond is 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.
- Heath-Jarrow-Morton (HJM) model

  The HJM model considers a given initial term structure of interest rates and a specification of the volatility of forward rates to build a tree representing the evolution of the interest rates, based on a statistical process.
- Black-Derman-Toy (BDT) model

  In the BDT model, all security prices and rates depend on the short rate (annualized one-period interest rate). The model uses long rates (the yield on a zero-coupon Treasury bond) 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 Instruments Toolbox is limited to one factor.
- Black-Karasinski (BK) model

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

For detailed information about interest-rate models, see:

- “Pricing Using Interest-Rate Term Structure” on page 2-61 for a discussion of price and sensitivity based on portfolios of zero-coupon bonds
- “Pricing Using Interest-Rate Tree Models” on page 2-81 for a discussion of price and sensitivity based on the HJM and BDT interest-rate models

---

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

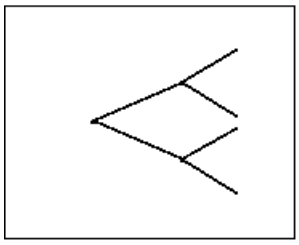
The HW and BK tree structures are similar to the BDT tree structure. To avoid needless repetition throughout this section, documentation is provided only where significant deviations from the BDT structure exist. Specifically, “HW and BK Tree Structures” on page 2-77 explains the few noteworthy differences among the various formats.

Rate and Price Trees

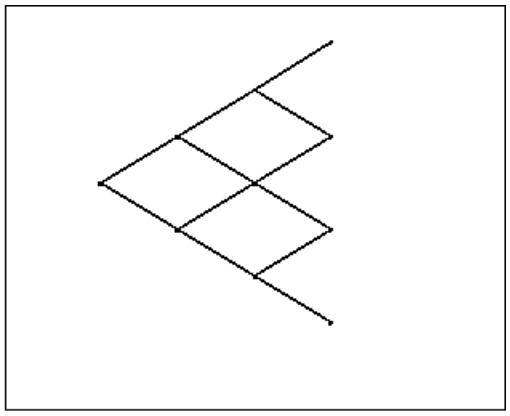
The interest-rate or price trees can be either binomial (two branches per node) or trinomial (three branches per node). Typically, binomial trees assume that underlying interest rates or prices can only either increase or decrease at each node. Trinomial trees allow for a more complex movement of rates or prices. With trinomial trees, the movement of rates or prices at each node is unrestricted (for example, up-up-up or unchanged-down-down). At any time step, the price or rate direction can be upward, neutral, or downward.

Types of Trees

Financial Instruments 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

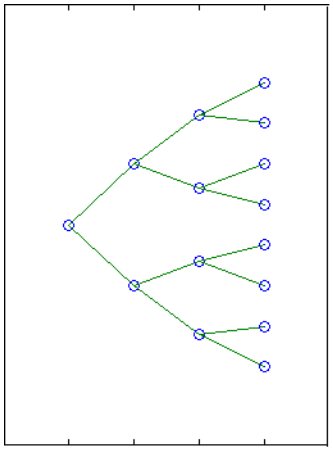
The Heath-Jarrow-Morton model works with bushy trees. The Black-Derman-Toy model, on the other hand, works with recombining binomial trees. The Hull-White and Black-Karasinsk interest-rate models work with recombining trinomial trees.

Viewing Rate or Price Movement

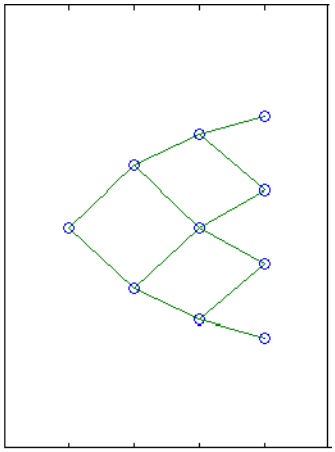
This toolbox provides the data file `deriv.mat` that contains four interest-rate based trees:

- `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.

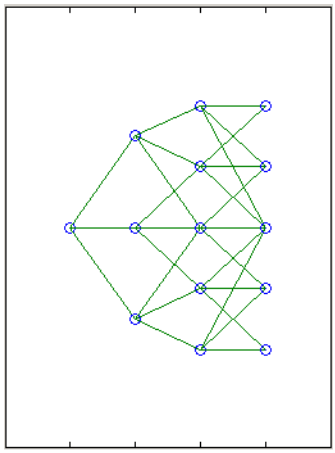


HJMTree (bushy)



BDTTree (recombining)

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



HWTree and BKTree (recombining)

See Also

`instbond` | `instcap` | `instcf` | `instfixed` | `instfloat` | `instfloor` | `instoptbnd` | `instoptembnd` | `instoptfloat` | `instoptemfloat` | `instrangefloat` | `instswap` | `instswaption` | `intenvset` | `bondbyzero` | `cfbyzero` | `fixedbyzero` | `floatbyzero` | `intenvprice` | `intenvsens` | `swapbyzero` | `floatmargin` | `floatdiscmargin` | `hjmtimespec` | `hjmtree` | `hjmvolspec` | `bondbyhjm` | `capbyhjm` | `cfbyhjm` | `fixedbyhjm` | `floatbyhjm` | `floorbyhjm` | `hjmprice` | `hjmsens` | `mmktbyhjm` | `oasbyhjm` | `optbndbyhjm` | `optfloatbyhjm` | `optembndbyhjm` | `optemfloatbyhjm` | `rangefloatbyhjm` | `swapbyhjm` | `swaptionbyhjm` | `bdttimespec` | `bdttree` | `bdtvolspec` | `bdtprice` | `bdtsens` | `bondbybdt` | `capbybdt` | `cfbybdt` | `fixedbybdt` | `floatbybdt` | `floorbybdt` | `mmktbybdt` | `oasbybdt` | `optbndbybdt` | `optfloatbybdt` | `optembndbybdt` | `optemfloatbybdt` | `rangefloatbybdt` | `swapbybdt` | `swaptionbybdt` | `hwtimespec` | `hwtree` | `hwvolspec` | `bondbyhw` | `capbyhw` | `cfbyhw` | `fixedbyhw` | `floatbyhw` | `floorbyhw` | `hwcalbycap` | `hwcalbyfloor` | `hwprice` | `hwsens` | `oasbyhw` | `optbndbyhw` | `optfloatbyhw` | `optembndbyhw` | `optemfloatbyhw` | `rangefloatbyhw` |

`swapbyhw` | `swaptionbyhw` | `bktimespec` | `bktree` | `bkvolspec` | `bkprice` | `bksens` | `bondbybk` | `capbybk` | `cfbybk` | `fixedbybk` | `floatbybk` | `floorbybk` | `oasbybk` | `optbndbybk` | `optfloatbybk` | `optembndbybk` | `optemfloatbybk` | `rangefloatbybk` | `swapbybk` | `swaptionbybk` | `capbyblk` | `floorbyblk` | `swaptionbyblk`

Related Examples

- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Pricing Using Interest-Rate Tree Models” on page 2-81
- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Graphical Representation of Trees” on page 2-219
- “Understanding the Interest-Rate Term Structure” on page 2-48

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

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, 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 Instruments 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 2-48) 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 six months. The vector `EndDates` is composed of dates separated by intervals of six months from the valuation date. This explains why the `EndTimes` vector is a progression of integers from 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. So, 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
```

See Also

`instbond` | `instcap` | `instcf` | `instfixed` | `instfloat` | `instfloor` | `instoptbnd` | `instoptembnd` | `instoptfloat` | `instoptemfloat` | `instrangefloat` | `instswap` | `instswaption` | `intenvset` | `bondbyzero` | `cfbyzero` | `fixedbyzero` | `floatbyzero` | `intenvprice` | `intenvsens` | `swapbyzero` | `floatmargin` | `floatdiscmargin` | `hjmtimespec` | `hjmtree` | `hjmvolspec` | `bondbyhjm` | `capbyhjm` | `cfbyhjm` | `fixedbyhjm` | `floatbyhjm` | `floorbyhjm` | `hjmprice` | `hjmsens` | `mmktbyhjm` | `oasbyhjm` | `optbndbyhjm` | `optfloatbyhjm` | `optembndbyhjm` | `optemfloatbyhjm` | `rangefloatbyhjm` | `swapbyhjm` | `swaptionbyhjm` | `bdttimespec` | `bdttree` | `bdtvolspec` | `bdtprice` | `bdtsens` | `bondbybdt` | `capbybdt` | `cfbybdt` | `fixedbybdt` | `floatbybdt` | `floorbybdt` | `mmktbybdt` | `oasbybdt` | `optbndbybdt` | `optfloatbybdt` | `optembndbybdt` | `optemfloatbybdt` | `rangefloatbybdt` | `swapbybdt` | `swaptionbybdt` | `hwtimespec` | `hwtree` | `hwvolspec` | `bondbyhw` | `capbyhw` | `cfbyhw` | `fixedbyhw` | `floatbyhw` | `floorbyhw` | `hwcalbycap` | `hwcalbyfloor` | `hwprice` | `hwsens` | `oasbyhw` | `optbndbyhw` | `optfloatbyhw` | `optembndbyhw` | `optemfloatbyhw` | `rangefloatbyhw` | `swapbyhw` | `swaptionbyhw` | `bktimespec` | `bktree` | `bkvolspec` | `bkprice` | `bksens` | `bondbybk` |

`capbybk` | `cfbybk` | `fixedbybk` | `floatbybk` | `floorbybk` | `oasbybk` | `optbndbybk` | `optfloatbybk` | `optembndbybk` | `optemfloatbybk` | `rangefloatbybk` | `swapbybk` | `swaptionbybk` | `capbyblk` | `floorbyblk` | `swaptionbyblk`

Related Examples

- “Modeling the Interest-Rate Term Structure” on page 2-57
- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Pricing Using Interest-Rate Tree Models” on page 2-81
- “Graphical Representation of Trees” on page 2-219

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

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 2-48.

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

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

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

% New Terms.
StartDates = ['15-Feb-2000'; '15-Aug-2000'; '15-Feb-2001';...
'15-Aug-2001'; '15-Feb-2002'];
EndDates =   ['15-Aug-2000'; '15-Feb-2001'; '15-Aug-2001';...
'15-Feb-2002'; '15-Aug-2002'];
% Find the new rates.
Rates = ratetimes(Compounding, RefRates, RefEndDates,...
RefStartDates, EndDates, StartDates, ValuationDate)
```

```
Rates =

    0.0500
    0.0620
    0.0680
    0.0801
    0.1155
```

Place these values in a table 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 six-month periods, and the valuation date equal to the date marking beginning of the first period (time factor = 0).

Now call `ratetimes` with the alternate syntax.

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

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

Rates =

    0.0500
    0.0560
    0.0600
    0.0650
    0.0750


EndTimes =

     1
     2
     3
     4
     5


StartTimes =

     0
     0
     0
     0
     0
```

`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
```

See Also

instbond | instcap | instcf | instfixed | instfloat | instfloor | instoptbnd | instoptembnd | instoptfloat | instoptemfloat | instrangefloat | instswap | instswaption | intenvset | bondbyzero | cfbyzero | fixedbyzero | floatbyzero | intenvprice | intenvsens | swapbyzero | floatmargin | floatdiscmargin | hjmtimespec | hjmtree | hjmvolspec | bondbyhjm | capbyhjm | cfbyhjm | fixedbyhjm | floatbyhjm | floorbyhjm | hjmprice | hjmsens | mmktbyhjm | oasbyhjm | optbndbyhjm | optfloatbyhjm | optembndbyhjm | optemfloatbyhjm | rangefloatbyhjm | swapbyhjm | swaptionbyhjm | bdttimespec | bdttree | bdtvolspec | bdtprice | bdtsens | bondbybdt | capbybdt | cfbybdt | fixedbybdt | floatbybdt | floorbybdt | mmktbybdt | oasbybdt | optbndbybdt | optfloatbybdt | optembndbybdt | optemfloatbybdt | rangefloatbybdt | swapbybdt | swaptionbybdt | hwtimespec | hwtree | hwvolspec | bondbyhw | capbyhw | cfbyhw | fixedbyhw | floatbyhw | floorbyhw | hwcalbycap | hwcalbyfloor | hwprice | hwsens | oasbyhw | optbndbyhw | optfloatbyhw | optembndbyhw | optemfloatbyhw | rangefloatbyhw | swapbyhw | swaptionbyhw | bktimespec | bktree | bkvolspec | bkprice | bksens | bondbybk | capbybk | cfbybk | fixedbybk | floatbybk | floorbybk | oasbybk | optbndbybk | optfloatbybk | optembndbybk | optemfloatbybk | rangefloatbybk | swapbybk | swaptionbybk | capbyblk | floorbyblk | swaptionbyblk

Related Examples

- “Modeling the Interest-Rate Term Structure” on page 2-57
- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Pricing Using Interest-Rate Tree Models” on page 2-81
- “Graphical Representation of Trees” on page 2-219

More About

- “Understanding the Interest-Rate Term Structure” on page 2-48
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Modeling the Interest-Rate Term Structure

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

- "Creating or Modifying (intenvset)" on page 2-57 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 2-58 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`.

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 Instruments Toolbox software work either with zero coupon rates or forward rates.

The other `intenvset` arguments are name-value pairs. The name-value pair arguments that can be specified or modified are:

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

For more information on `Basis`, see "Basis" on page 2-228.

Consider again the original table of interest rates (see "Calculating Discount Factors from Rates" on page 2-48).

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

From	To	Rate
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 six months is represented with a value of `0.5`, and a period of one year is represented by `1`. To obtain the vectors `StartTimes` and `EndTimes`, enter:

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

```
Times =

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

Since all the values in `StartDates` are the same as the valuation date, all `StartTimes` values are 0. On the other hand, the values in the `EndDates` vector are dates separated by six-month periods. Since the redefined value of compounding is 1, `EndTimes` becomes a sequence of numbers separated by increments of 0.5.

See Also

`instbond` | `instcap` | `instcf` | `instfixed` | `instfloat` | `instfloor` | `instoptbnd` | `instoptembnd` | `instoptfloat` | `instoptemfloat` | `instrangefloat` | `instswap` |

`instswaption` | `intenvset` | `bondbyzero` | `cfbyzero` | `fixedbyzero` | `floatbyzero` | `intenvprice` | `intenvsens` | `swapbyzero` | `floatmargin` | `floatdiscmargin` | `hjmtimespec` | `hjmtree` | `hjmvolspec` | `bondbyhjm` | `capbyhjm` | `cfbyhjm` | `fixedbyhjm` | `floatbyhjm` | `floorbyhjm` | `hjmprice` | `hjmsens` | `mmktbyhjm` | `oasbyhjm` | `optbndbyhjm` | `optfloatbyhjm` | `optembndbyhjm` | `optemfloatbyhjm` | `rangefloatbyhjm` | `swapbyhjm` | `swaptionbyhjm` | `bdttimespec` | `bdttree` | `bdtvolspec` | `bdtprice` | `bdtsens` | `bondbybdt` | `capbybdt` | `cfbybdt` | `fixedbybdt` | `floatbybdt` | `floorbybdt` | `mmktbybdt` | `oasbybdt` | `optbndbybdt` | `optfloatbybdt` | `optembndbybdt` | `optemfloatbybdt` | `rangefloatbybdt` | `swapbybdt` | `swaptionbybdt` | `hwtimespec` | `hwtree` | `hwvolspec` | `bondbyhw` | `capbyhw` | `cfbyhw` | `fixedbyhw` | `floatbyhw` | `floorbyhw` | `hwcalbycap` | `hwcalbyfloor` | `hwprice` | `hwsens` | `oasbyhw` | `optbndbyhw` | `optfloatbyhw` | `optembndbyhw` | `optemfloatbyhw` | `rangefloatbyhw` | `swapbyhw` | `swaptionbyhw` | `bktimespec` | `bktree` | `bkvolspec` | `bkprice` | `bksens` | `bondbybk` | `capbybk` | `cfbybk` | `fixedbybk` | `floatbybk` | `floorbybk` | `oasbybk` | `optbndbybk` | `optfloatbybk` | `optembndbybk` | `optemfloatbybk` | `rangefloatbybk` | `swapbybk` | `swaptionbybk` | `capbyblk` | `floorbyblk` | `swaptionbyblk`

Related Examples

- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Pricing Using Interest-Rate Tree Models” on page 2-81
- “Graphical Representation of Trees” on page 2-219

More About

- “Understanding the Interest-Rate Term Structure” on page 2-48
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Pricing Using 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 five instrument types of using interest-rate curves:

- Bonds
- Fixed-rate notes
- Floating-rate notes
- Swaps
- OAS for callable and puttable bonds
- Agency OAS

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

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 Instruments Toolbox 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 “Pricing Using Interest-Rate Tree Models” on page 2-81 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`.

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 Instruments 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`. So, 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
```

OAS for Callable and Puttable Bonds

Option Adjusted Spread (OAS) is a useful way to value and compare securities with embedded options, like callable or puttable bonds. Basically, when the constant or flat spread is added to the interest-rate curve/rates in the tree, the pricing model value equals the market price. Financial Instruments Toolbox supports pricing American, European, and Bermuda callable and puttable bonds using different interest rate models. The pricing for a bond with embedded options is:

- For a callable bond, where the holder has bought a bond and sold a call option to the issuer:

  `Price callable bond = Price Option free bond − Price call option`
- For a puttable bond, where the holder has bought a bond and a put option:

  `Price puttable bond = Price Option free bond + Price put option`

There are two additional sensitivities related to OAS for bonds with embedded options: Option Adjusted Duration and Option Adjusted Convexity. These are similar to the concepts of modified duration and convexity for option-free bonds. The measure Duration is a general term that describes how sensitive a bond's price is to a parallel shift in the yield curve. Modified Duration and Modified Convexity assume that the bond's cash flows do not change when the yield curve shifts. This is not true for OA Duration or OA Convexity because the cash flows may change due to the option risk component of the bond.

Function	Purpose
`oasbybdt`	Compute OAS using a BDT model.
`oasbybk`	Compute OAS using a BK model.
`oasbyhjm`	Compute OAS using an HJM model.
`oasbyhw`	Compute OAS using an HW model.

Agency OAS

Often bonds are issued with embedded options, which then makes standard price/yield or spread measures irrelevant. For example, a municipality concerned about the chance that interest rates may fall in the future might issue bonds with a provision that allows the bond to be repaid before the bond's maturity. This is a call option on the bond and must be incorporated into the valuation of the bond. Option-adjusted spread (OAS), which adjusts a bond spread for the value of the option, is the standard measure for valuing bonds with embedded options. Financial Instruments Toolbox supports computing option-adjusted spreads for bonds with single embedded options using the agency model.

The Securities Industry and Financial Markets Association (SIFMA) has a simplified approach to compute OAS for agency issues (Government Sponsored Entities like Fannie Mae and Freddie Mac) termed "Agency OAS." In this approach, the bond has only one call date (European call) and uses Black's model (see *The BMA European Callable Securities Formula* at https://www.sifma.org) to value the bond option. The price of the bond is computed as follows:

$\texttt{Price}_{\text{Callable}} = \texttt{Price}_{\text{NonCallable}} - \texttt{Price}_{\text{Option}}$

where

$\texttt{Price}_{\text{Callable}}$ is the price of the callable bond.

$\texttt{Price}_{\text{NonCallable}}$ is the price of the noncallable bond, that is, price of the bond using `bndspread`.

$Price_{Option}$ is the price of the option, that is, price of the option using Black's model.

The Agency OAS is the spread, when used in the previous formula, yields the market price. Financial Instruments Toolbox supports these functions:

Agency OAS

Agency OAS Functions	Purpose
`agencyoas`	Compute the OAS of the callable bond using the Agency OAS model.
`agencyprice`	Price the callable bond OAS using the Agency OAS model.

For more information on agency OAS, see "Agency Option-Adjusted Spreads" on page 6-2.

See Also

`instbond` | `instcap` | `instcf` | `instfixed` | `instfloat` | `instfloor` | `instoptbnd` | `instoptembnd` | `instoptfloat` | `instoptemfloat` | `instrangefloat` | `instswap` | `instswaption` | `intenvset` | `bondbyzero` | `cfbyzero` | `fixedbyzero` | `floatbyzero` | `intenvprice` | `intenvsens` | `swapbyzero` | `floatmargin` | `floatdiscmargin` | `hjmtimespec` | `hjmtree` | `hjmvolspec` | `bondbyhjm` | `capbyhjm` | `cfbyhjm` | `fixedbyhjm` | `floatbyhjm` | `floorbyhjm` | `hjmprice` | `hjmsens` | `mmktbyhjm` | `oasbyhjm` | `optbndbyhjm` | `optfloatbyhjm` | `optembndbyhjm` | `optemfloatbyhjm` | `rangefloatbyhjm` | `swapbyhjm` | `swaptionbyhjm` | `bdttimespec` | `bdttree` | `bdtvolspec` | `bdtprice` | `bdtsens` | `bondbybdt` | `capbybdt` | `cfbybdt` | `fixedbybdt` | `floatbybdt` | `floorbybdt` | `mmktbybdt` | `oasbybdt` | `optbndbybdt` | `optfloatbybdt` | `optembndbybdt` | `optemfloatbybdt` | `rangefloatbybdt` | `swapbybdt` | `swaptionbybdt` | `hwtimespec` | `hwtree` | `hwvolspec` | `bondbyhw` | `capbyhw` | `cfbyhw` | `fixedbyhw` | `floatbyhw` | `floorbyhw` | `hwcalbycap` | `hwcalbyfloor` | `hwprice` | `hwsens` | `oasbyhw` | `optbndbyhw` | `optfloatbyhw` | `optembndbyhw` | `optemfloatbyhw` | `rangefloatbyhw` | `swapbyhw` | `swaptionbyhw` | `bktimespec` | `bktree` | `bkvolspec` | `bkprice` | `bksens` | `bondbybk` | `capbybk` | `cfbybk` | `fixedbybk` | `floatbybk` | `floorbybk` | `oasbybk` | `optbndbybk` | `optfloatbybk` | `optembndbybk` | `optemfloatbybk` | `rangefloatbybk` | `swapbybk` | `swaptionbybk` | `capbyblk` | `floorbyblk` | `swaptionbyblk`

Related Examples

- "Pricing Using Interest-Rate Term Structure" on page 2-61
- "Understanding the Interest-Rate Term Structure" on page 2-48

More About

- "Supported Interest-Rate Instrument Functions" on page 2-3
- "Supported Equity Derivative Functions" on page 3-19
- "Supported Energy Derivative Functions" on page 3-34
- "Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects" on page 1-73

Understanding Interest-Rate Tree Models

In this section...

Introduction

Financial Instruments Toolbox supports the following interest-rate trees:

- Black-Derman-Toy (BDT)
- Black-Karasinski (BK)
- Heath-Jarrow-Morton (HJM)
- Hull-White (HW)
- Cox-Ingersoll-Ross (CIR)

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 Instruments 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 Instruments Toolbox.

Note To avoid needless repetition, this document uses the HJM and BDT models to illustrate the creation and use of interest-rate trees. The HW and BK models are similar to the BDT model. Where specific differences exist, they are documented in “HW and BK Tree Structures” on page 2-77.

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 2-68.)
- The interest-rate term structure `RateSpec`. (See “Specifying the Interest-Rate Term Structure (RateSpec)” on page 2-70.)
- The tree time layout `TimeSpec`. (See “Specifying the Time Structure (TimeSpec)” on page 2-70.)

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 2-219 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 “Modeling the Interest-Rate Term Structure” on page 2-57.)

- `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 2-68. For BDT examples, see “Creating a BDT Volatility Model” on page 2-69.

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 character vector specifying the name of the factor, followed by the pertinent parameters.

HJM Volatility Specification Example

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

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

```
HJMVolSpec =

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

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

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

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

```
HJMVolSpec =
```

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

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

Creating a BDT Volatility Model

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

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

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

The syntax used for calling `bdtvolspec` is:

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

where:

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

BDT Volatility Specification Example

Consider the following example:

```
ValuationDate = datetime(2000,1,1);
EndDates = [datetime(2001,1,1) ; datetime(2002,1,1) ; datetime(2003,1,1) ;datetime(2004,1,1) ; da
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. “Modeling the Interest-Rate Term Structure” on page 2-57 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
```

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` on page 2-70 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
```

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
```

Creating Trees

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 Instruments Toolbox uses trees to represent forward rates, prices, and so on. At the highest level, these trees have structures wrapped around them. The structures encapsulate information 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 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 Instruments Toolbox 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 tapped 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 Instruments Toolbox, such as forward-rate trees or price trees.

BDT Tree Structure

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

```
BDTTree
```

```
BDTTree =

      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1.00 2.00 3.00]
        TFwd: {[4x1 double]  [3x1 double]  [2x1 double]  [3.00]}
     CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4.00]}
    FwdTree: {[1.10]  [1.10 1.14]  [1.10 1.14 1.19]  [1.09 1.12 1.16 1.22]}
```

FwdTree contains the actual rate tree. MATLAB represents it as a cell array with each cell array element containing a tree level.

The other fields contain other information relevant to interpreting the values in FwdTree. The most important 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. So, when displaying *rates*, rates are falling on the upper branch and increasing on the lower branch.

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

```
ans =

    1.0976    1.1377    1.1942
```

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

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

```
ans =

    1.0872    1.1183    1.1606    1.2179
```

The fourth node represents the third observation time, `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 tapped 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
```

```
HWTree =

      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1.00 2.00 3.00]
        dObs: [731947.00 732313.00 732678.00 733043.00]
     CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4.00]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
     Connect: {[2.00]  [2.00 3.00 4.00]  [2.00 2.00 3.00 4.00 4.00]}
     FwdTree: {[1.03]  [1.05 1.04 1.02]  [1.08 1.07 1.05 1.03 1.01]  [1.09 1.08 1.06 1.04 1.02]
```

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

Probs Field

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

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

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

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

```
0.16666666666667
0.66666666666667
0.16666666666667
```

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

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

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

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

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

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

```
1.0000    1.0000    1.0000
```

Connect Field

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

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

```
HWTree.Connect
```

```
ans =

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

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

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

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

```
HWTree.Connect{1}

ans =

     2
```

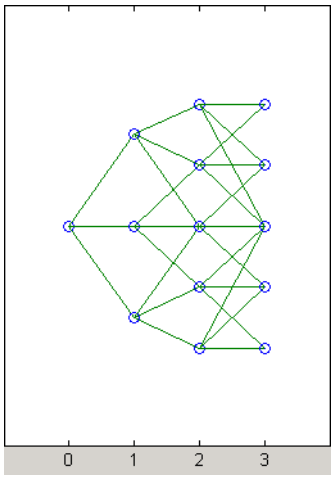
This indicates that the middle branch of the root node connects to the second (from the top) node of the next level, as expected. If you subtract `1` from this value, you obtain `1`, which tells you that the up branch goes to the top node. If you add `1`, you obtain `3`, which points to the last node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

Now consider level 3 in this example:

```
HWTree.Connect{3}

2     2     3     4     4
```

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

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



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

See Also

`instbond` | `instcap` | `instcf` | `instfixed` | `instfloat` | `instfloor` | `instoptbnd` | `instoptembnd` | `instoptfloat` | `instoptemfloat` | `instrangefloat` | `instswap` | `instswaption` | `intenvset` | `bondbyzero` | `cfbyzero` | `fixedbyzero` | `floatbyzero` | `intenvprice` | `intenvsens` | `swapbyzero` | `floatmargin` | `floatdiscmargin` | `hjmtimespec` | `hjmtree` | `hjmvolspec` | `bondbyhjm` | `capbyhjm` | `cfbyhjm` | `fixedbyhjm` | `floatbyhjm` |

floorbyhjm | hjmprice | hjmsens | mmktbyhjm | oasbyhjm | optbndbyhjm | optfloatbyhjm | optembndbyhjm | optemfloatbyhjm | rangefloatbyhjm | swapbyhjm | swaptionbyhjm | bdttimespec | bdttree | bdtvolspec | bdtprice | bdtsens | bondbybdt | capbybdt | cfbybdt | fixedbybdt | floatbybdt | floorbybdt | mmktbybdt | oasbybdt | optbndbybdt | optfloatbybdt | optembndbybdt | optemfloatbybdt | rangefloatbybdt | swapbybdt | swaptionbybdt | hwtimespec | hwtree | hwvolspec | bondbyhw | capbyhw | cfbyhw | fixedbyhw | floatbyhw | floorbyhw | hwcalbycap | hwcalbyfloor | hwprice | hwsens | oasbyhw | optbndbyhw | optfloatbyhw | optembndbyhw | optemfloatbyhw | rangefloatbyhw | swapbyhw | swaptionbyhw | bktimespec | bktree | bkvolspec | bkprice | bksens | bondbybk | capbybk | cfbybk | fixedbybk | floatbybk | floorbybk | oasbybk | optbndbybk | optfloatbybk | optembndbybk | optemfloatbybk | rangefloatbybk | swapbybk | swaptionbybk | capbyblk | floorbyblk | swaptionbyblk

Related Examples

- “Overview of Interest-Rate Tree Models” on page 2-44
- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Pricing Using Interest-Rate Tree Models” on page 2-81
- “Graphical Representation of Trees” on page 2-219
- “Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Pricing 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
- Bond with embedded options
- Arbitrary cash flows
- Fixed-rate notes
- Floating-rate notes
- Floating-rate notes with options or embedded options
- Caps
- Floors
- Range Notes
- 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 “Pricing Options Structure” on page A-2 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 2-66 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.

`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`, `rangefloatbyhjm`, `swapbyhjm`, and `swaptionbyhjm`. A similarly named set of functions exists for BDT models. You can also use these functions directly to calculate the price of sets of instruments of the same type.

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 EndMonthRule IssueDate FirstCouponDate LastCouponDate StartDate Face Name    Quanti
1     Bond 0.04       01-Jan-2000    01-Jan-2003    1      NaN   NaN          NaN       NaN             NaN            NaN       NaN  4% bond 100
2     Bond 0.04       01-Jan-2000    01-Jan-2004    2      NaN   NaN          NaN       NaN             NaN            NaN       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


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
```

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 EndMonthRule IssueDate FirstCouponDate LastCouponDate StartDate Face Name     Quant
1     Bond 0.1        01-Jan-2000    01-Jan-2003    1      NaN   NaN          NaN       NaN             NaN            NaN       NaN  10% Bond 100
2     Bond 0.1        01-Jan-2000    01-Jan-2004    2      NaN   NaN          NaN       NaN             NaN            NaN       NaN  10% Bond  50

Index Type    UnderInd OptSpec Strike ExerciseDates  AmericanOpt Name      Quantity
3     OptBond 1        call    95     01-Jan-2002    NaN         Option 95 -50

Index Type  CouponRate Settle         Maturity       FixedReset Basis Principal Name      Quantity
4     Fixed 0.1        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
```

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
```

So, 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
```

So, 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)
```

```
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In checktree (line 292)
In hjmprice (line 85)

Price =

   98.7159
   97.5280
    0.0486
```

```
   98.7159
  100.5529
    6.2831
    0.0486
    3.6923


PriceTree =

  struct with fields:

    FinObj: 'HJMPriceTree'
     PBush: {[8×1 double]  [8×1×2 double]  [8×2×2 double]  [8×4×2 double]  [8×8 double]}
    AIBush: {[8×1 double]  [8×1×2 double]  [8×2×2 double]  [8×4×2 double]  [8×8 double]}
    ExBush: {[8×1 double]  [8×1×2 double]  [8×2×2 double]  [8×4×2 double]  [8×8 double]}
      tObs: [0 1 2 3 4]
```

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 this:

```
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
```

See Also

`instbond` | `instcap` | `instcf` | `instfixed` | `instfloat` | `instfloor` | `instoptbnd` | `instoptembnd` | `instoptfloat` | `instoptemfloat` | `instrangefloat` | `instswap` | `instswaption` | `intenvset` | `bondbyzero` | `cfbyzero` | `fixedbyzero` | `floatbyzero` | `intenvprice` | `intenvsens` | `swapbyzero` | `floatmargin` | `floatdiscmargin` | `hjmtimespec` | `hjmtree` | `hjmvolspec` | `bondbyhjm` | `capbyhjm` | `cfbyhjm` | `fixedbyhjm` | `floatbyhjm` | `floorbyhjm` | `hjmprice` | `hjmsens` | `mmktbyhjm` | `oasbyhjm` | `optbndbyhjm` | `optfloatbyhjm` | `optembndbyhjm` | `optemfloatbyhjm` | `rangefloatbyhjm` | `swapbyhjm` | `swaptionbyhjm` | `bdttimespec` | `bdttree` | `bdtvolspec` | `bdtprice` | `bdtsens` | `bondbybdt` | `capbybdt` | `cfbybdt` | `fixedbybdt` | `floatbybdt` | `floorbybdt` | `mmktbybdt` | `oasbybdt` | `optbndbybdt` | `optfloatbybdt` | `optembndbybdt` | `optemfloatbybdt` | `rangefloatbybdt` | `swapbybdt` | `swaptionbybdt` | `hwtimespec` | `hwtree` | `hwvolspec` | `bondbyhw` | `capbyhw` | `cfbyhw` | `fixedbyhw` | `floatbyhw` | `floorbyhw` | `hwcalbycap` | `hwcalbyfloor` | `hwprice` | `hwsens` | `oasbyhw` | `optbndbyhw` | `optfloatbyhw` | `optembndbyhw` | `optemfloatbyhw` | `rangefloatbyhw` | `swapbyhw` | `swaptionbyhw` | `bktimespec` | `bktree` | `bkvolspec` | `bkprice` | `bksens` | `bondbybk` | `capbybk` | `cfbybk` | `fixedbybk` | `floatbybk` | `floorbybk` | `oasbybk` | `optbndbybk` | `optfloatbybk` | `optembndbybk` | `optemfloatbybk` | `rangefloatbybk` | `swapbybk` | `swaptionbybk` | `capbyblk` | `floorbyblk` | `swaptionbyblk`

Related Examples

- “Overview of Interest-Rate Tree Models” on page 2-44
- “Computing Instrument Sensitivities” on page 2-89
- “Graphical Representation of Trees” on page 2-219

- “Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
- “Understanding Interest-Rate Tree Models” on page 2-66
- “Understanding the Interest-Rate Term Structure” on page 2-48
- “Pricing Using Interest-Rate Term Structure” on page 2-61

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

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
```

See Also

instbond | instcap | instcf | instfixed | instfloat | instfloor | instoptbnd | instoptembnd | instoptfloat | instoptemfloat | instrangefloat | instswap | instswaption | intenvset | bondbyzero | cfbyzero | fixedbyzero | floatbyzero | intenvprice | intenvsens | swapbyzero | floatmargin | floatdiscmargin | hjmtimespec | hjmtree | hjmvolspec | bondbyhjm | capbyhjm | cfbyhjm | fixedbyhjm | floatbyhjm | floorbyhjm | hjmprice | hjmsens | mmktbyhjm | oasbyhjm | optbndbyhjm | optfloatbyhjm | optembndbyhjm | optemfloatbyhjm | rangefloatbyhjm | swapbyhjm | swaptionbyhjm | bdttimespec | bdttree | bdtvolspec | bdtprice | bdtsens | bondbybdt | capbybdt | cfbybdt | fixedbybdt | floatbybdt | floorbybdt | mmktbybdt | oasbybdt | optbndbybdt | optfloatbybdt | optembndbybdt | optemfloatbybdt | rangefloatbybdt | swapbybdt | swaptionbybdt | hwtimespec | hwtree | hwvolspec | bondbyhw | capbyhw | cfbyhw | fixedbyhw | floatbyhw | floorbyhw | hwcalbycap | hwcalbyfloor | hwprice | hwsens | oasbyhw | optbndbyhw | optfloatbyhw | optembndbyhw | optemfloatbyhw | rangefloatbyhw | swapbyhw | swaptionbyhw | bktimespec | bktree | bkvolspec | bkprice | bksens | bondbybk | capbybk | cfbybk | fixedbybk | floatbybk | floorbybk | oasbybk | optbndbybk | optfloatbybk | optembndbybk | optemfloatbybk | rangefloatbybk | swapbybk | swaptionbybk | capbyblk | floorbyblk | swaptionbyblk

Related Examples

- “Overview of Interest-Rate Tree Models” on page 2-44
- “Pricing Using Interest-Rate Tree Models” on page 2-81
- “Graphical Representation of Trees” on page 2-219
- “Understanding Interest-Rate Tree Models” on page 2-66
- “Understanding the Interest-Rate Term Structure” on page 2-48

- “Pricing Using Interest-Rate Term Structure” on page 2-61

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Calibrating 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. Calibration is used to determine these parameters, such that the model can reproduce, as close as possible, the prices of caps or floors observed in the market. The calibration routines find the parameters that minimize the difference between the model price predictions and the market prices for caps and floors.

For a 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's predicted prices and the observed market prices of the corresponding caplets or floorlets.

Hull-White Model Calibration Example

Use market data to identify the implied volatility (σ) and mean reversion (α) coefficients needed to build a Hull-White tree to price an instrument. The ideal case is to use the volatilities of the caps or floors used to calculate `Alpha` (α) and `Sigma` (σ). This will most likely not be the case, so market data must be interpolated to obtain the required values.

Consider a cap with these parameters:

```
Settle = ' Jan-21-2008';
Maturity = 'Mar-21-2011';
Strike = 0.0690;
Reset = 4;
Principal = 1000;
Basis = 0;
```

The caplets for this example would fall in:

```
capletDates = cfdates(Settle, Maturity, Reset, Basis);
datestr(capletDates')
```

```
ans =

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
```

In the best case, look up the market volatilities for caplets with a `Strike = 0.0690`, and maturities in each reset date listed, but the likelihood of finding these exact instruments is low. As a consequence, use data that is available in the market and interpolate to find appropriate values for the caplets.

Based on the market data, you have the cap information for different dates and strikes. Assume that instead of having the data for `Strike = 0.0690`, you have the data for `Strike1 = 0.0590` and `Strike2 = 0.0790`.

Maturity	Strike1 = 0.0590	Strike2 = 0.0790
21-Mar-2008	0.1533	0. 1526
21-Jun-2008	0.1731	0. 1730
21-Sep-2008	0. 1727	0. 1726
21-Dec-2008	0. 1752	0. 1747
21-Mar-2009	0. 1809	0. 1808
21-Jun-2009	0. 1809	0. 1792
21-Sep-2009	0. 1805	0. 1797
21-Dec-2009	0. 1802	0. 1794
21-Mar-2010	0. 1802	0. 1733
21-Jun-2010	0. 1757	0. 1751
21-Sep-2010	0. 1755	0. 1750
21-Dec-2010	0. 1755	0. 1745
21-Mar-2011	0. 1726	0. 1719

The nature of this data lends itself to matrix nomenclature, which is perfect for MATLAB. `hwcalbycap` requires that the dates, the strikes, and the actual volatility be separated into three variables: `MarketStrike`, `MarketMat`, and `MarketVol`.

```
MarketStrike = [0.0590; 0.0790];
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'};


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; % First row in table corresponding to Strike1
             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]; % Second row in table corresponding to Strike2
```

Complete the input arguments using this data for `RateSpec`:

```
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;
Basis = 0;

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

RateSpec =

           FinObj: 'RateSpec'
      Compounding: 4
             Disc: [14x1 double]
            Rates: [14x1 double]
         EndTimes: [14x1 double]
       StartTimes: [14x1 double]
         EndDates: [14x1 double]
       StartDates: 733428
    ValuationDate: 733428
            Basis: 0
     EndMonthRule: 1
```

Call the calibration routine to find values for volatility parameters Alpha and Sigma

Use `hwcalbycap` to calculate the values of `Alpha` and `Sigma` based on market data. Internally, `hwcalbycap` calls the function `lsqnonlin`. You can customize `lsqnonlin` by passing an optimization options structure created by `optimoptions` and then this can be passed to `hwcalbycap` using the name-value pair argument for `OptimOptions`. For example, `optimoptions` defines the target objective function tolerance as `100*eps` and then calls `hwcalbycap`:

```
o=optimoptions('lsqnonlin','TolFun',100*eps);

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

Local minimum possible.

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

Warning: LSQNONLIN did not converge to an optimal solution. It exited with exitflag = 2.

> In hwcalbycapfloor at 93
  In hwcalbycap at 75

Alpha =

   1.0000e-06


Sigma =

    0.0127
```

The previous warning indicates that the conversion was not optimal. The search algorithm used by the Optimization Toolbox™ function `lsqnonlin` did not find a solution that conforms to all the constraints. To discern whether the solution is acceptable, 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 is almost always some residual, so decide if it is acceptable to parameterize the market with a single value of `Alpha` and `Sigma`.

Price caplets using market data and Black's formula to obtain reference caplet values

To determine the effectiveness of the optimization, calculate reference caplet values using Black's formula and the market data. Note, you must first interpolate the market data to obtain the caplets for calculation:

```
MarketMatNum = datenum(MarketMat);
[Mats, Strikes] = meshgrid(MarketMatNum, MarketStrike);
FlatVol = interp2(Mats, Strikes, MarketVol, datenum(Maturity), Strike, 'spline');
```

Compute the price of the cap using the Black model:

```
[CapPrice, Caplets] = capbyblk(RateSpec, Strike, Settle, Maturity, FlatVol,...
'Reset', Reset, 'Basis', Basis, 'Principal', Principal);
Caplets = Caplets(2:end)';
```

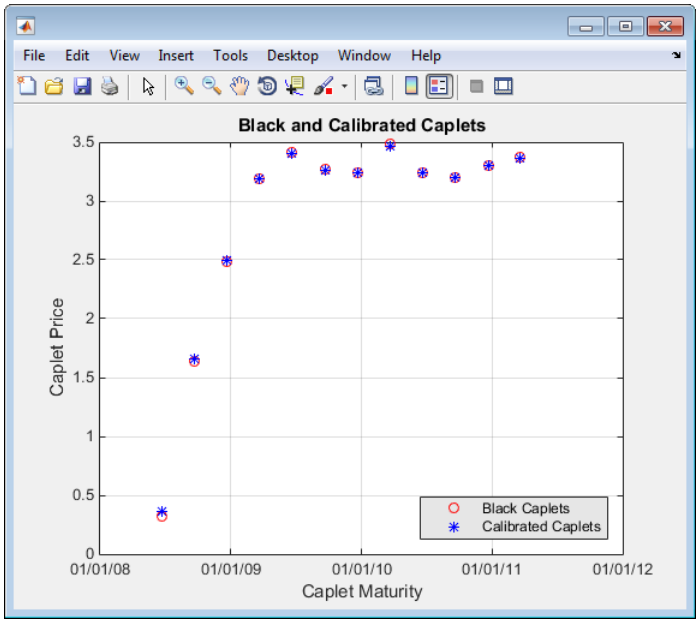
```
Caplets =

    0.3210
    1.6355
    2.4863
    3.1903
    3.4110
    3.2685
    3.2385
    3.4803
    3.2419
    3.1949
    3.2991
    3.3750
```

Compare optimized values and Black values and display graphically

After calculating the reference values for the caplets, compare the values, analytically and graphically, to determine whether the calculated single values of `Alpha` and `Sigma` provide an adequate approximation:

```
OptimCaplets = Caplets+OptimOut.residual;

disp('   ');
disp('    Black76   Calibrated Caplets');
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
```

```
Black76   Calibrated Caplets
   0.3210    0.3636
   1.6355    1.6603
   2.4863    2.4974
   3.1903    3.1874
   3.4110    3.4040
   3.2685    3.2639
   3.2385    3.2364
   3.4803    3.4683
   3.2419    3.2408
   3.1949    3.1957
   3.2991    3.2960
   3.3750    3.3663
```



Compare cap prices using the Black, HW analytical, and HW tree models

Using the calculated caplet values, compare the prices of the corresponding cap using the Black model, Hull-White analytical, and Hull-White tree models. To calculate a Hull-White tree based on `Alpha` and `Sigma`, use these calibration routines:

- Black model:

  ```
  CapPriceBLK = CapPrice;
  ```

- HW analytical model:

  ```
  CapPriceHWAnalytical = sum(OptimCaplets);
  ```

- HW tree model to price the cap derived from the calibration process:

  1. Create `VolSpec` from the calibration parameters `Alpha` and `Sigma`:

     ```
     VolDates    = EndDates;
     VolCurve    = Sigma*ones(14,1);
     AlphaDates  = EndDates;
     AlphaCurve  = Alpha*ones(14,1);
     HWVolSpec = hwvolspec(ValuationDate, VolDates, VolCurve,AlphaDates, AlphaCurve);
     ```

  2. Create the `TimeSpec`:

```
HWTimeSpec = hwtimespec(ValuationDate, EndDates, Compounding);
```

3. Build the HW tree using the `HW2000` method:

```
HWTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec, 'Method', 'HW2000');
```

4. Price the cap:

```
Price = capbyhw(HWTree, Strike, Settle, Maturity, Reset, Basis, Principal);

disp('   ');
disp(['  CapPrice Black76 ..................:  ', num2str(CapPriceBLK,'%15.5f')]);
disp(['  CapPrice HW analytical..........:  ', num2str(CapPriceHWAnalytical,'%15.5f')]);
disp(['  CapPrice HW from capbyhw ..:  ', num2str(Price,'%15.5f')]);
disp('   ');
```

```
CapPrice Black76 ...........: 34.14220
CapPrice HW analytical.....: 34.18008
CapPrice HW from capbyhw ..: 34.14192
```

Price a portfolio of instruments using the calibrated HW tree

After building a Hull-White tree, based on parameters calibrated from market data, use `HWTree` to price a portfolio of these instruments:

- Two bonds

```
CouponRate = [0.07; 0.09];
Settle = ' Jan-21-2008';
Maturity = {'Mar-21-2010';'Mar-21-2011'};
Period = 1;
Face = 1000;
Basis = 0;
```

- Bond with an embedded American call option

```
CouponRateOEB = 0.08;
SettleOEB = ' Jan-21-2008';
MaturityOEB = 'Mar-21-2011';
OptSpec = 'call';
StrikeOEB = 950;
ExerciseDatesOEB = 'Mar-21-2011';
AmericanOpt = 1;
Period = 1;
Face = 1000;
Basis = 0;
```

To price this portfolio of instruments using the calibrated `HWTree`:

1. Use `instadd` to create the portfolio `InstSet`:

```
InstSet = instadd('Bond', CouponRate, Settle,  Maturity, Period, Basis, [], [], [], [], [], Face);
InstSet = instadd(InstSet,'OptEmBond',  CouponRateOEB, SettleOEB, MaturityOEB, OptSpec,...
StrikeOEB,   ExerciseDatesOEB, 'AmericanOpt', AmericanOpt, 'Period', Period,...
'Face',Face,  'Basis', Basis);
```

2. Add the cap instrument used in the calibration:

```
SettleCap = ' Jan-21-2008';
MaturityCap = 'Mar-21-2011';
StrikeCap = 0.0690;
Reset = 4;
Principal = 1000;

InstSet = instadd(InstSet,'Cap', StrikeCap, SettleCap, MaturityCap, Reset, Basis, Principal);
```

3. Assign names to the portfolio instruments:

```
Names = {'7% Bond'; '8% Bond'; 'BondEmbCall'; '6.9% Cap'};
InstSet = instsetfield(InstSet, 'Index',1:4, 'FieldName', {'Name'}, 'Data', Names );
```

4 Examine the set of instruments contained in `InstSet`:

```
instdisp(InstSet)
```

```
IdxType CoupRate Settle Mature Period Basis EOMRule IssueDate 1stCoupDate LastCoupDate StartDate Face Name

1 Bond 0.07       21-Jan-2008    21-Mar-2010    1  0  NaN  NaN     NaN   NaN  NaN  1000    7% Bond
2 Bond 0.09       21-Jan-2008    21-Mar-2011    1  0  NaN  NaN     NaN   NaN  NaN  1000    8% Bond

IdxType CoupRate Settle Mature OptSpec Stke ExDate Per Basis EOMRule IssDate 1stCoupDate LstCoupDate StrtDate Face AmerOpt Name
3 OptEmBond 0.08 21-Jan-2008 21-Mar-2011 call 950  21-Jan-2008  21-Mar-2011  1  0  1  NaN  NaN NaN  NaN  1000 1 BondEmbCall

Index Type Strike Settle     Maturity   CapReset Basis Principal Name
4 Cap  0.069  21-Jan-2008    21-Mar-2011    4      0     1000    6.9% Cap
```

5 Use `hwprice` to price the portfolio using the calibrated `HWTree`:

```
format bank
PricePortfolio = hwprice(HWTree, InstSet)
```

```
PricePortfolio =
         980.45
        1023.05
         945.73
          34.14
```

See Also

`instbond` | `instcap` | `instoptbnd` | `instoptembnd` | `intenvset` | `hwtimespec` | `hwtree` | `hwvolspec` | `bondbyhw` | `capbyhw` | `hwcalbycap` | `hwcalbyfloor` | `hwprice` | `hwsens`

Related Examples

- “Overview of Interest-Rate Tree Models” on page 2-44
- “Pricing Using Interest-Rate Tree Models” on page 2-81
- “Graphical Representation of Trees” on page 2-219
- “Understanding Interest-Rate Tree Models” on page 2-66
- “Understanding the Interest-Rate Term Structure” on page 2-48
- “Pricing Using Interest-Rate Term Structure” on page 2-61

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

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`

See Also

`capbyblk` | `floorbyblk` | `swaptionbyblk` | `blackvolbysabr` | `optsensbysabr` | `agencyoas` | `agencyprice` | `bndfutimprepo` | `bndfutprice` | `convfactor` | `tfutbyprice` | `tfutbyyield` | `tfutimprepo` | `tfutpricebyrepo` | `tfutyieldbyrepo` | `capbylg2f` | `floorbylg2f` | `swaptionbylg2f` | `blackvolbyrebonato` | `hwcalbycap` | `hwcalbyfloor`

Related Examples

- “Calibrate the SABR Model” on page 2-33
- “Price a Swaption Using the SABR Model” on page 2-38
- “Computing the Agency OAS for Bonds” on page 6-2
- “Analysis of Bond Futures” on page 7-12
- “Managing Interest-Rate Risk with Bond Futures” on page 2-125
- “Fitting the Diebold Li Model” on page 7-15
- “Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
- “Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114

More About

- “Managing Present Value with Bond Futures” on page 7-14
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Price Swaptions with Interest-Rate Models Using Simulation

In this section...

Introduction

This example shows how to price European swaptions using interest-rate models in Financial Instruments Toolbox. Specifically, a Hull-White one factor model, a Linear Gaussian two-factor model, and a LIBOR Market Model are calibrated to market data and then used to generate interest-rate paths using Monte Carlo simulation.

The following sections set up the data that is then used with examples for “Simulate Interest-Rate Paths Using the Hull-White One-Factor Model” on page 2-103, “Simulate Interest-Rate Paths Using the Linear Gaussian Two-Factor Model” on page 2-106, and “Simulate Interest-Rate Paths Using the LIBOR Market Model” on page 2-108:

- “Construct a Zero Curve” on page 2-100
- “Define Swaption Parameters” on page 2-102
- “Compute the Black Model and the Swaption Volatility Matrix” on page 2-102
- “Select Calibration Instruments” on page 2-102
- “Compute Swaption Prices Using Black's Model” on page 2-102
- “Define Simulation Parameters” on page 2-103

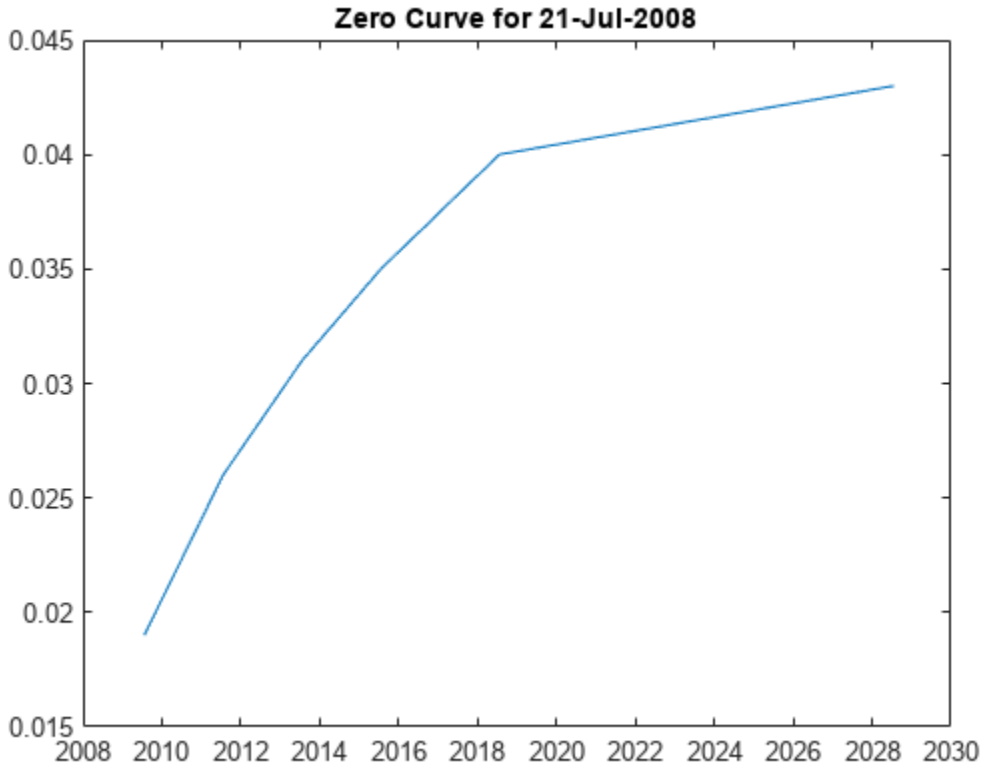
Construct a Zero Curve

This example shows how to use `ZeroRates` for a zero curve that is hard-coded. You can also create a zero curve by bootstrapping the zero curve from market data (for example, deposits, futures/forwards, and swaps)

The hard-coded data for the zero curve is defined as:

```
Settle = datetime(2008,7,21);
```

```
% Zero Curve
CurveDates = daysadd(Settle,360*([1 3 5 7 10 20]),1);
ZeroRates = [1.9 2.6 3.1 3.5 4 4.3]'/100;

plot(CurveDates,ZeroRates)
datetick
title(['Zero Curve for ' datestr(Settle)]);
```



Construct an `IRDataCurve` object.

```
irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);
```

Create the `RateSpec` using `intenvset`.

```
RateSpec = intenvset('Rates',ZeroRates,'EndDates',CurveDates,'StartDate',Settle)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [6x1 double]
            Rates: [6x1 double]
         EndTimes: [6x1 double]
       StartTimes: [6x1 double]
         EndDates: [6x1 double]
       StartDates: 733610
    ValuationDate: 733610
            Basis: 0
```

```
      EndMonthRule: 1
```

Define Swaption Parameters

While Monte Carlo simulation is typically used to value more sophisticated derivatives (for example, Bermudan swaptions), in this example, the price of a European swaption is computed with an exercise date of five years and an underlying swap of five years.

```
InstrumentExerciseDate = datenum('21-Jul-2013');
InstrumentMaturity = datenum('21-Jul-2018');
InstrumentStrike = .045;
```

Compute the Black Model and the Swaption Volatility Matrix

Black's model is often used to price and quote European exercise interest-rate options, that is, caps, floors and swaptions. In the case of swaptions, Black's model is used to imply a volatility given the current observed market price. The following matrix shows the Black implied volatility for a range of swaption exercise dates (columns) and underlying swap maturities (rows).

```
SwaptionBlackVol = [22 21 19 17 15 13 12
    21 19 17 16 15 13 11
    20 18 16 15 14 12 11
    19 17 15 14 13 12 10
    18 16 14 13 12 11 10
    15 14 13 12 12 11 10
    13 13 12 11 11 10 9]/100;
ExerciseDates = [1:5 7 10];
Tenors = [1:5 7 10];

EurExDatesFull = repmat(daysadd(Settle,ExerciseDates*360,1)',...
    length(Tenors),1);
EurMatFull = reshape(daysadd(EurExDatesFull,...
    repmat(360*Tenors,1,length(ExerciseDates)),1),size(EurExDatesFull));
```

Select Calibration Instruments

Selecting the instruments to calibrate the model to is one of the tasks in calibration. For Bermudan swaptions, it is typical to calibrate to European swaptions that are co-terminal with the Bermudan swaption to be priced. In this case, all swaptions having an underlying tenor that matures before the maturity of the swaption to be priced (21-Jul-2018) are used in the calibration.

```
% Find the swaptions that expire on or before the maturity date of the
% sample swaption
relidx = find(EurMatFull <= InstrumentMaturity);
```

Compute Swaption Prices Using Black's Model

This example shows how to compute swaption prices using Black's Model. The swaption prices are then used to compare the model’s predicted values that are obtained from the calibration process.

To compute the swaption prices using Black's model:

```
SwaptionBlackPrices = zeros(size(SwaptionBlackVol));
SwaptionStrike = zeros(size(SwaptionBlackVol));

for iSwaption=1:length(ExerciseDates)
    for iTenor=1:length(Tenors)
        [~,SwaptionStrike(iTenor,iSwaption)] = swapbyzero(RateSpec,[NaN 0], Settle, EurMatFull(iTenor,iSwaption),...
            'StartDate',EurExDatesFull(iTenor,iSwaption),'LegReset',[1 1]);
```

```
        SwaptionBlackPrices(iTenor,iSwaption) = swaptionbyblk(RateSpec, 'call', SwaptionStrike(iTenor,iSwaption),Settle, ...
            EurExDatesFull(iTenor,iSwaption), EurMatFull(iTenor,iSwaption), SwaptionBlackVol(iTenor,iSwaption));
    end
end
```

Define Simulation Parameters

This example shows how to use the `simTermStructs` method with `HullWhite1F`, `LinearGaussian2F`, and `LiborMarketModel` objects.

To demonstrate using the `simTermStructs` method with `HullWhite1F`, `LinearGaussian2F`, and `LiborMarketModel` objects, use the following simulation parameters:

```
nPeriods = 5;
DeltaTime = 1;
nTrials = 1000;

Tenor = (1:10)';

SimDates = daysadd(Settle,360*DeltaTime*(0:nPeriods),1)
SimTimes = diff(yearfrac(SimDates(1),SimDates))

% For 1 year periods and an evenly spaced tenor, the exercise row will be
% the 5th row and the swaption maturity will be the 5th column
exRow = 5;
endCol = 5;
```

```
SimDates =

      733610
      733975
      734340
      734705
      735071
      735436


SimTimes =

    1.0000
    1.0000
    1.0000
    1.0027
    1.0000
```

Simulate Interest-Rate Paths Using the Hull-White One-Factor Model

This example shows how to simulate interest-rate paths using the Hull-White one-factor model. Before beginning this example that uses a `HullWhite1F` model, make sure that you have set up the data as described in:

- “Construct a Zero Curve” on page 2-100
- “Define Swaption Parameters” on page 2-102
- “Compute the Black Model and the Swaption Volatility Matrix” on page 2-102
- “Select Calibration Instruments” on page 2-102
- “Compute Swaption Prices Using Black's Model” on page 2-102
- “Define Simulation Parameters” on page 2-103

The Hull-White one-factor model describes the evolution of the short rate and is specified using the zero curve, *alpha*, and *sigma* parameters for the equation

$$dr = [\theta(t) - a(t)r]dt + \sigma(t)dW$$

where:

dr is the change in the short-term interest rate over a small interval, dt.

r is the short-term interest rate.

$\Theta(t)$ is a function of time determining the average direction in which r moves, chosen such that movements in r are consistent with today's zero coupon yield curve.

α is the mean reversion rate.

dt is a small change in time.

σ is the annual standard deviation of the short rate.

W is the Brownian motion.

The Hull-White model is calibrated using the function `swaptionbyhw`, which constructs a trinomial tree to price the swaptions. Calibration consists of minimizing the difference between the observed market prices (computed above using the Black's implied swaption volatility matrix, see “Compute the Black Model and the Swaption Volatility Matrix” on page 2-102) and the model's predicted prices.

In this example, the Optimization Toolbox function `lsqnonlin` is used to find the parameter set that minimizes the difference between the observed and predicted values. However, other approaches (for example, simulated annealing) may be appropriate. Starting parameters and constraints for α and σ are set in the variables `x0`, `lb`, and `ub`; these could also be varied depending upon the particular calibration approach.

Calibrate the set of parameters that minimize the difference between the observed and predicted values using `swaptionbyhw` and `lsqnonlin`.

```
TimeSpec = hwtimespec(Settle,daysadd(Settle,360*(1:11),1), 2);
HW1Fobjfun = @(x) SwaptionBlackPrices(relidx) - ...
    swaptionbyhw(hwtree(hwvolspec(Settle,'11-Aug-2015',x(2),'11-Aug-2015',x(1)), RateSpec, TimeSpec), 'call', SwaptionStrike(relidx),...
    EurExDatesFull(relidx), 0, EurExDatesFull(relidx), EurMatFull(relidx));
options = optimset('disp','iter','MaxFunEvals',1000,'TolFun',1e-5);

% Find the parameters that minimize the difference between the observed and
% predicted prices
x0 = [.1 .01];
lb = [0 0];
ub = [1 1];
HW1Fparams = lsqnonlin(HW1Fobjfun,x0,lb,ub,options);

HW_alpha = HW1Fparams(1)
HW_sigma = HW1Fparams(2)
```

```
                                    Norm of      First-order
 Iteration  Func-count     f(x)          step          optimality
     0          3        0.953772                         20.5
     1          6        0.142828      0.0169199           1.53
     2          9        0.123022      0.0146705           2.31
     3         12        0.122222      0.0154098          0.482
     4         15        0.122217     0.00131297        0.00409

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the selected value of the function tolerance.

HW_alpha =

    0.0967


HW_sigma =

    0.0088
```

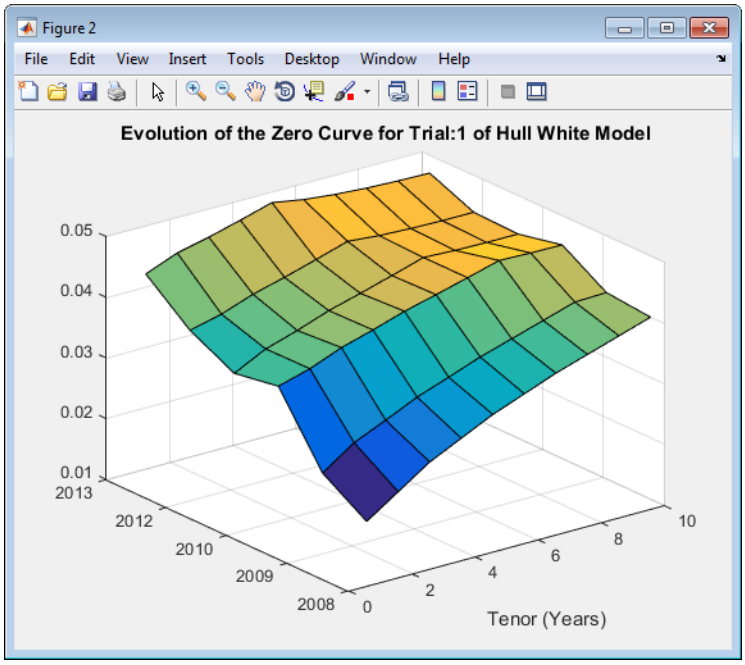
Construct the `HullWhite1F` model using the `HullWhite1F` constructor.

```
HW1F = HullWhite1F(RateSpec,HW_alpha,HW_sigma)
```

```
HW1F =

  HullWhite1F with properties:

    ZeroCurve: [1x1 IRDataCurve]
        Alpha: @(t,V)inAlpha
        Sigma: @(t,V)inSigma
```

Use Monte Carlo simulation to generate the interest-rate paths with `HullWhite1F.simTermStructs`.

```
HW1FSimPaths = HW1F.simTermStructs(nPeriods,'NTRIALS',nTrials,...
    'DeltaTime',DeltaTime,'Tenor',Tenor,'antithetic',true);
trialIdx = 1;
figure
surf(Tenor,SimDates,HW1FSimPaths(:,:,trialIdx))
datetick y keepticks keeplimits
title(['Evolution of the Zero Curve for Trial:' num2str(trialIdx) ' of Hull White Model'])
xlabel('Tenor (Years)')
```



Price the European swaption.

```
DF = exp(bsxfun(@times,-HW1FSimPaths,repmat(Tenor',[nPeriods+1 1])));
SwapRate = (1 - DF(exRow,endCol,:))./sum(bsxfun(@times,1,DF(exRow,1:endCol,:)));
PayoffValue = 100*max(SwapRate-InstrumentStrike,0).*sum(bsxfun(@times,1,DF(exRow,1:endCol,:)));
RealizedDF = prod(exp(bsxfun(@times,-HW1FSimPaths(1:exRow,1,:),SimTimes(1:exRow))),1);
HW1F_SwaptionPrice = mean(RealizedDF.*PayoffValue)
```

```
HW1F_SwaptionPrice =

    2.1839
```

Simulate Interest-Rate Paths Using the Linear Gaussian Two-Factor Model

This example shows how to simulate interest-rate paths using the Linear Gaussian two-factor model. Before beginning this example that uses a `LinearGaussian2F` model, make sure that you have set up the data as described in:

- “Construct a Zero Curve” on page 2-100
- “Define Swaption Parameters” on page 2-102
- “Compute the Black Model and the Swaption Volatility Matrix” on page 2-102
- “Select Calibration Instruments” on page 2-102
- “Compute Swaption Prices Using Black's Model” on page 2-102
- “Define Simulation Parameters” on page 2-103

The Linear Gaussian two-factor model (called the G2++ by Brigo and Mercurio, see “Interest-Rate Modeling Using Monte Carlo Simulation” on page B-4) is also a short rate model, but involves two factors. Specifically:

$$r(t) = x(t) + y(t) + \phi(t)$$

$$dx(t) = -a(t)x(t)dt + \sigma(t)dW_1(t), x(0) = 0$$

$$dy(t) = -b(t)y(t)dt + \eta(t)dW_2(t), y(0) = 0$$

where $dW_1(t)dW_2(t) = \rho dt$ is a two-dimensional Brownian motion with correlation ρ, and ϕ is a function chosen to match the initial zero curve.

The function `swaptionbylg2f` is used to compute analytic values of the swaption price for model parameters, and therefore can be used to calibrate the model. Calibration consists of minimizing the difference between the observed market prices (computed above using the Black's implied swaption volatility matrix, see “Compute the Black Model and the Swaption Volatility Matrix” on page 2-102) and the model’s predicted prices.

In this example, the approach is similar to “Simulate Interest-Rate Paths Using the Hull-White One-Factor Model” on page 2-103 and the Optimization Toolbox function `lsqnonlin` is used to minimize the difference between the observed swaption prices and the predicted swaption prices. However, other approaches (for example, simulated annealing) may also be appropriate. Starting parameters and constraints for a, b, η, ρ, and σ are set in the variables `x0`, `lb`, and `ub`; these could also be varied depending upon the particular calibration approach.

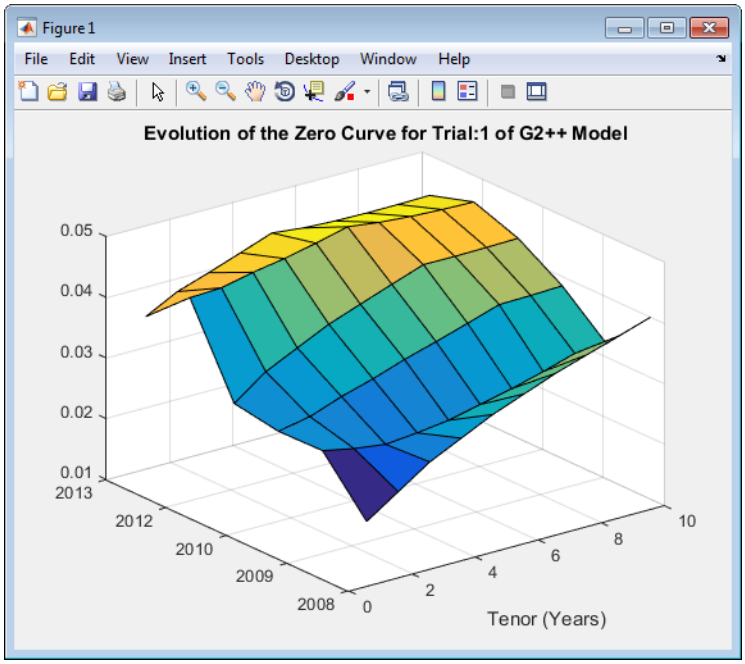
Calibrate the set of parameters that minimize the difference between the observed and predicted values using `swaptionbylg2f` and `lsqnonlin`.

```
G2PPobjfun = @(x) SwaptionBlackPrices(relidx) - swaptionbylg2f(irdc,x(1),x(2),x(3),x(4),x(5),SwaptionStrike(relidx),...
    EurExDatesFull(relidx),EurMatFull(relidx),'Reset',1);

options = optimset('disp','iter','MaxFunEvals',1000,'TolFun',1e-5);
x0 = [.2 .1 .02 .01 -.5];
lb = [0 0 0 0 -1];
ub = [1 1 1 1 1];
LG2Fparams = lsqnonlin(G2PPobjfun,x0,lb,ub,options)
```

```
                                        Norm of      First-order
 Iteration  Func-count     f(x)          step          optimality
     0          6         12.3547                         67.6
     1         12         1.37984      0.0979743           8.59
     2         18         1.37984       0.112847           8.59
     3         24        0.445202      0.0282118           1.31
     4         30        0.236746      0.0564236           3.02
     5         36        0.134678      0.0843366           7.78
     6         42       0.0398816       0.015084           6.34
     7         48       0.0287731       0.038967          0.732
     8         54       0.0273025       0.112847          0.881
     9         60       0.0241689       0.213033           1.06
    10         66       0.0241689       0.125602           1.06
    11         72       0.0239103      0.0314005           9.78
    12         78       0.0234246      0.0286685           1.21
    13         84       0.0234246      0.0491135           1.21
    14         90        0.023304      0.0122784           1.67
    15         96       0.0231931      0.0245568           5.92
    16        102       0.0230898     0.00785421          0.434
    17        108       0.0230898      0.0245568          0.434
    18        114        0.023083     0.00613919          0.255

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the selected value of the function tolerance.

LG2Fparams =

    0.5752    0.1181    0.0146    0.0119   -0.7895
```

Create the `G2PP` object using `LinearGaussian2F` and use Monte Carlo simulation to generate the interest-rate paths with `LinearGaussian2F.simTermStructs`.

```
LG2f_a = LG2Fparams(1);
LG2f_b = LG2Fparams(2);
LG2f_sigma = LG2Fparams(3);
LG2f_eta = LG2Fparams(4);
LG2f_rho = LG2Fparams(5);

G2PP = LinearGaussian2F(RateSpec,LG2f_a,LG2f_b,LG2f_sigma,LG2f_eta,LG2f_rho);

G2PPSimPaths = G2PP.simTermStructs(nPeriods,'NTRIALS',nTrials,...
    'DeltaTime',DeltaTime,'Tenor',Tenor,'antithetic',true);

trialIdx = 1;
figure
surf(Tenor,SimDates,G2PPSimPaths(:,:,trialIdx))
datetick y keepticks keeplimits
title(['Evolution of the Zero Curve for Trial:' num2str(trialIdx) ' of G2++ Model'])
xlabel('Tenor (Years)')
```



Price the European swaption.

```
DF = exp(bsxfun(@times,-G2PPSimPaths,repmat(Tenor',[nPeriods+1 1])));
SwapRate = (1 - DF(exRow,endCol,:))./sum(bsxfun(@times,1,DF(exRow,1:endCol,:)));
PayoffValue = 100*max(SwapRate-InstrumentStrike,0).*sum(bsxfun(@times,1,DF(exRow,1:endCol,:)));
RealizedDF = prod(exp(bsxfun(@times,-G2PPSimPaths(1:exRow,1,:),SimTimes(1:exRow))),1);
G2PP_SwaptionPrice = mean(RealizedDF.*PayoffValue)
```

```
G2PP_SwaptionPrice =

    2.0988
```

Simulate Interest-Rate Paths Using the LIBOR Market Model

This example shows how to simulate interest-rate paths using the LIBOR market model. Before beginning this example that uses a `LiborMarketModel`, make sure that you have set up the data as described in:

- “Construct a Zero Curve” on page 2-100
- “Define Swaption Parameters” on page 2-102
- “Compute the Black Model and the Swaption Volatility Matrix” on page 2-102
- “Select Calibration Instruments” on page 2-102
- “Compute Swaption Prices Using Black's Model” on page 2-102
- “Define Simulation Parameters” on page 2-103

The LIBOR Market Model (LMM) differs from short rate models in that it evolves a set of discrete forward rates. Specifically, the lognormal LMM specifies the following diffusion equation for each forward rate

$$\frac{dF_i(t)}{F_i} = -\mu_i dt + \sigma_i(t) dW_i$$

where:

W is an N-dimensional geometric Brownian motion with

$$dW_i(t)dW_j(t) = \rho_{ij}$$

The LMM relates the drifts of the forward rates based on no-arbitrage arguments. Specifically, under the Spot LIBOR measure, the drifts are expressed as

$$\mu_i(t) = -\sigma_i(t)\sum_{j=q(t)}^{i}\frac{\tau_j\rho_{i,j}\sigma_j(t)F_j(t)}{1+\tau_jF_j(t)}$$

where:

τ_i is the time fraction associated with the ith forward rate

$q(t)$ is an index defined by the relation

$$T_{q(t)-1} < t < T_{q(t)}$$

and the Spot LIBOR numeraire is defined as

$$B(t) = P(t,T_{q(t)})\prod_{n=0}^{q(t)-1}(1+\tau_nF_n(T_n))$$

The choice with the LMM is how to model volatility and correlation and how to estimate the parameters of these models for volatility and correlation. In practice, you may use a combination of historical data (for example, observed correlation between forward rates) and current market data. For this example, only swaption data is used. Further, many different parameterizations of the volatility and correlation exist. For this example, two relatively straightforward parameterizations are used.

One of the most popular functional forms in the literature for volatility is:

$$\sigma_i(t) = \varphi_i(a(T_i-t)+b)e^{c(T_i-t)}+d$$

where ϕ adjusts the curve to match the volatility for the i th forward rate. For this example, all of the ϕ's are taken to be 1. For the correlation, the following functional form is used:

$$\rho_{i,j} = e^{-\beta|i-j|}$$

Once the functional forms have been specified, these parameters must be estimated using market data. One useful approximation, initially developed by Rebonato, is the following, which relates the Black volatility for a European swaption, given a set of volatility functions and a correlation matrix

$$(v_{\alpha,\beta}^{LFM})^2 = \sum_{i,j=\alpha+1}^{\beta}\frac{w_i(0)w_j(0)F_i(0)F_j(0)\rho_{i,j}}{S_{\alpha,\beta}(0)^2}\int_0^{T_\alpha}\sigma_i(t)\sigma_j(t)dt$$

where:

$$w_i(t) = \frac{\tau_iP(t,T_i)}{\sum_{k=\alpha+1}^{\beta}\tau_kP(t,t_k)}$$

This calculation is done using the function `blackvolbyrebonato` to compute analytic values of the swaption price for model parameters, and therefore, is then used to calibrate the model. Calibration consists of minimizing the difference between the observed implied swaption Black volatilities and the predicted Black volatilities.

In this example, the approach is similar to “Simulate Interest-Rate Paths Using the Hull-White One-Factor Model” on page 2-103 and “Simulate Interest-Rate Paths Using the Linear Gaussian Two-Factor Model” on page 2-106 where the Optimization Toolbox function `lsqnonlin` is used to minimize the difference between the observed swaption prices and the predicted swaption prices. However, other approaches (for example, simulated annealing) may also be appropriate. Starting parameters and constraints for a, b, d, and β are set in the variables `x0`, `lb`, and `ub`; these could also be varied depending upon the particular calibration approach.

Calibrate the set of parameters that minimize the difference between the observed and predicted values using `blackvolbyrebonato` and `lsqnonlin`.

```
nRates = 10;

CorrFunc = @(i,j,Beta) exp(-Beta*abs(i-j));

objfun = @(x) SwaptionBlackVol(relidx) - blackvolbyrebonato(RateSpec,...
    repmat({@(t) ones(size(t)).*(x(1)*t + x(2)).*exp(-x(3)*t) + x(4)},nRates-1,1),...
    CorrFunc(meshgrid(1:nRates-1)',meshgrid(1:nRates-1),x(5)),...
    EurExDatesFull(relidx),EurMatFull(relidx),'Period',1);

options = optimset('disp','iter','MaxFunEvals',1000,'TolFun',1e-5);

x0 = [.2 .05 1 .05 .2];
lb = [0 0 .5 0 .01];
ub = [1 1 2 .3 1];
LMMparams = lsqnonlin(objfun,x0,lb,ub,options)
```

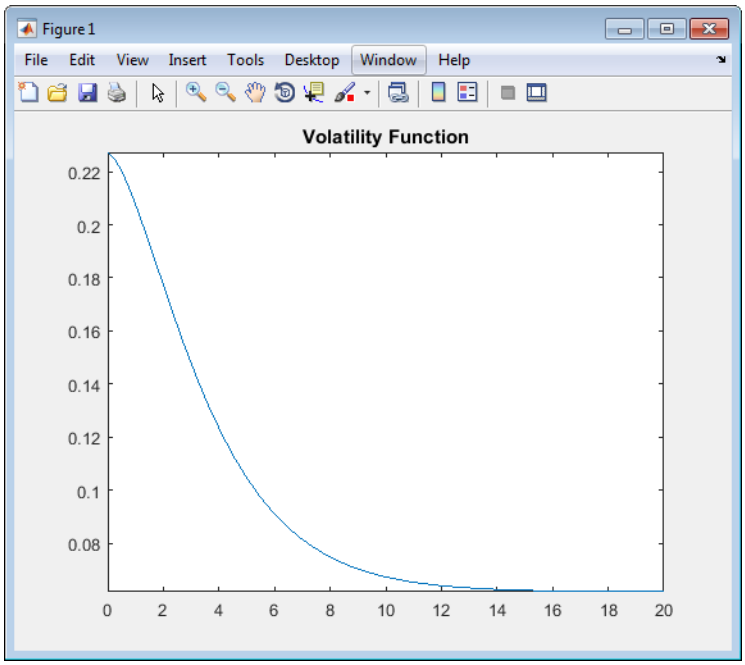
```
                                    Norm of      First-order
 Iteration  Func-count     f(x)          step          optimality
     0          6        0.156251                         0.483
     1         12      0.00870177       0.188164          0.0339
     2         18      0.00463441       0.165527         0.00095
     3         24      0.00331055       0.351017          0.0154
     4         30      0.00294775      0.0892617        7.47e-05
     5         36      0.00281565       0.385779         0.00917
     6         42      0.00278988      0.0145632        4.15e-05
     7         48      0.00278522       0.115042         0.00116
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the selected value of the function tolerance.

LMMparams =

    0.0781    0.1656    0.5121    0.0617    0.0100
```

Calculate `VolFunc` for the `LMM` object.

```
a = LMMparams(1);
b = LMMparams(2);
c = LMMparams(3);
d = LMMparams(4);

Beta = LMMparams(5);

VolFunc = repmat({@(t) ones(size(t)).*(a*t + b).*exp(-c*t) + d},nRates-1,1);
```

Plot the volatility function.

```
figure
fplot(VolFunc{1},[0 20])
title('Volatility Function')

CorrelationMatrix = CorrFunc(meshgrid(1:nRates-1)',meshgrid(1:nRates-1),Beta);
```



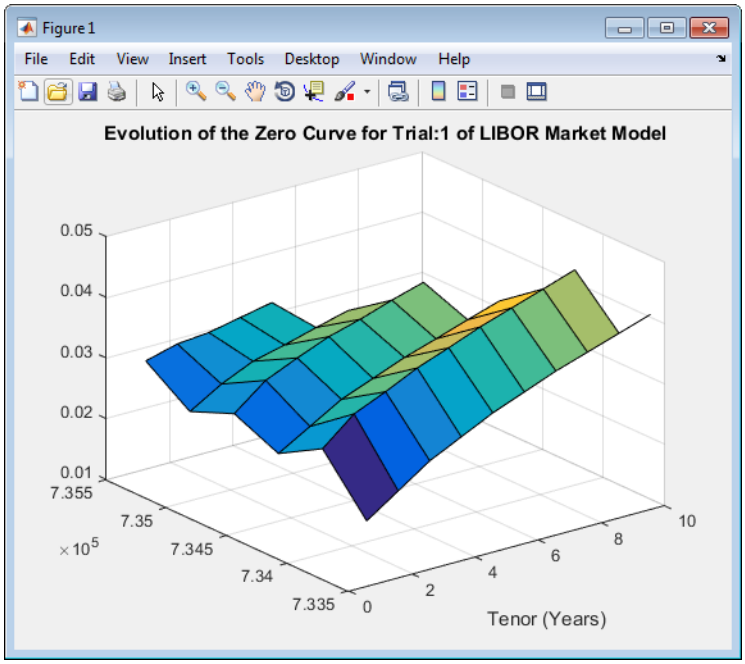
Inspect the correlation matrix.

```
disp('Correlation Matrix')
fprintf([repmat('%1.3f ',1,length(CorrelationMatrix)) ' \n'],CorrelationMatrix)
```

```
Correlation Matrix
1.000 0.990 0.980 0.970 0.961 0.951 0.942 0.932 0.923
0.990 1.000 0.990 0.980 0.970 0.961 0.951 0.942 0.932
0.980 0.990 1.000 0.990 0.980 0.970 0.961 0.951 0.942
0.970 0.980 0.990 1.000 0.990 0.980 0.970 0.961 0.951
0.961 0.970 0.980 0.990 1.000 0.990 0.980 0.970 0.961
0.951 0.961 0.970 0.980 0.990 1.000 0.990 0.980 0.970
0.942 0.951 0.961 0.970 0.980 0.990 1.000 0.990 0.980
0.932 0.942 0.951 0.961 0.970 0.980 0.990 1.000 0.990
0.923 0.932 0.942 0.951 0.961 0.970 0.980 0.990 1.000
```

Create the `LMM` object using `LiborMarketModel` and use Monte Carlo simulation to generate the interest-rate paths with `LiborMarketModel.simTermStructs`.

```
LMM = LiborMarketModel(irdc,VolFunc,CorrelationMatrix,'Period',1);

[LMMZeroRates, ForwardRates] = LMM.simTermStructs(nPeriods,'nTrials',nTrials);

trialIdx = 1;
figure
tmpPlotData = LMMZeroRates(:,:,trialIdx);
tmpPlotData(tmpPlotData == 0) = NaN;
surf(Tenor,SimDates,tmpPlotData)
title(['Evolution of the Zero Curve for Trial:' num2str(trialIdx) ' of LIBOR Market Model'])
xlabel('Tenor (Years)')
```



Price the European swaption.

```
DF = exp(bsxfun(@times,-LMMZeroRates,repmat(Tenor',[nPeriods+1 1])));
SwapRate = (1 - DF(exRow,endCol,:))./sum(bsxfun(@times,1,DF(exRow,1:endCol,:)));
PayoffValue = 100*max(SwapRate-InstrumentStrike,0).*sum(bsxfun(@times,1,DF(exRow,1:endCol,:)));
RealizedDF = prod(exp(bsxfun(@times,-LMMZeroRates(2:exRow+1,1,:),SimTimes(1:exRow))),1);
LMM_SwaptionPrice = mean(RealizedDF.*PayoffValue)
```

```
LMM_SwaptionPrice =

    1.9915
```

Compare Interest-Rate Modeling Results

This example shows how to compare the results for pricing a European swaption with different interest-rate models.

Compare the results for pricing a European swaption with interest-rate models using Monte Carlo simulation.

```
disp(' ')
fprintf('     # of Monte Carlo Trials: %8d\n'    , nTrials)
fprintf('     # of Time Periods/Trial: %8d\n\n'  , nPeriods)
fprintf('HW1F European Swaption Price: %8.4f\n', HW1F_SwaptionPrice)
fprintf('LG2F Europesn Swaption Price: %8.4f\n', G2PP_SwaptionPrice)
fprintf(' LMM European Swaption Price: %8.4f\n', LMM_SwaptionPrice)
```

```
 # of Monte Carlo Trials:     1000
     # of Time Periods/Trial:        5

HW1F European Swaption Price:   2.1839
```

```
LG2F Europesn Swaption Price:   2.0988
 LMM European Swaption Price:   1.9915
```

References

Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice with Smile, Inflation and Credit.* Springer Finance, 2006.

Andersen, L. and V. Piterbarg. *Interest Rate Modeling.* Atlantic Financial Press. 2010.

Hull, J. *Options, Futures, and Other Derivatives.* Springer Finance, 2003.

Glasserman, P. *Monte Carlo Methods in Financial Engineering.* Prentice Hall, 2008.

Rebonato, R., K. McKay, and R. White. *The Sabr/Libor Market Model: Pricing, Calibration and Hedging for Complex Interest-Rate Derivatives.* John Wiley & Sons, 2010.

See Also

`HullWhite1F` | `LinearGaussian2F` | `LiborMarketModel` | `simTermStructs` | `simTermStructs` | `simTermStructs` | `capbylg2f` | `floorbylg2f` | `swaptionbylg2f` | `swaptionbyhw` | `blackvolbyrebonato` | `lsqnonlin`

Related Examples

- “Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114

More About

- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

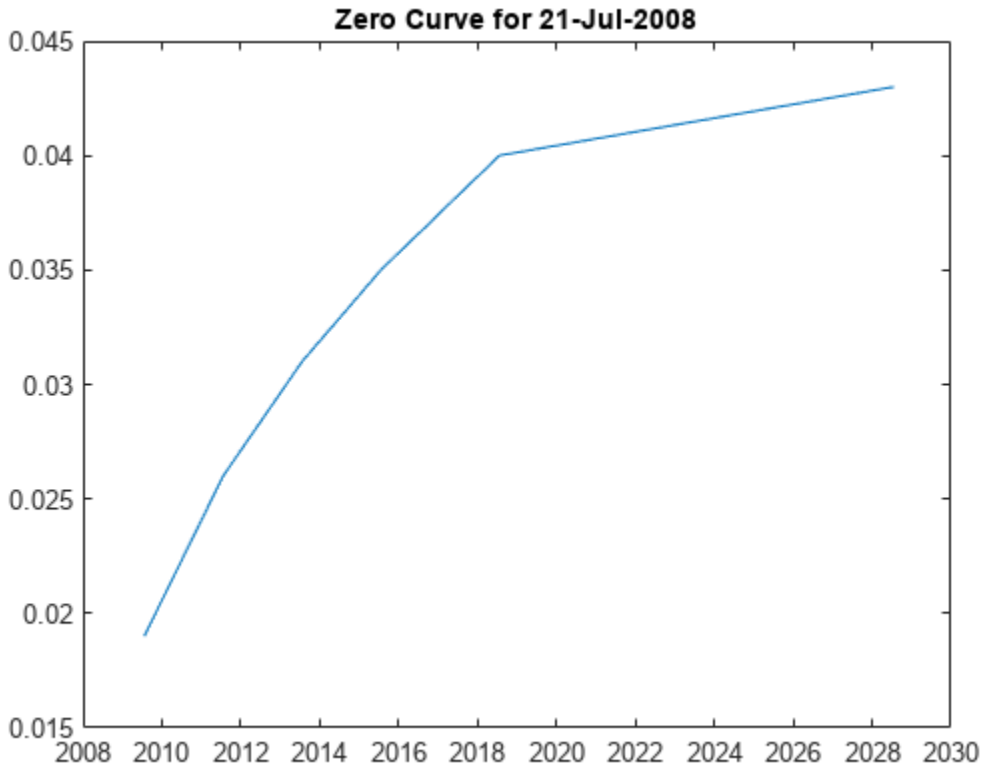
Pricing Bermudan Swaptions with Monte Carlo Simulation

This example shows how to price Bermudan swaptions using interest-rate models in Financial Instruments Toolbox™. Specifically, a Hull-White one factor model, a Linear Gaussian two-factor model, and a LIBOR Market Model are calibrated to market data and then used to generate interest-rate paths using Monte Carlo simulation.

Zero Curve

In this example, the `ZeroRates` for a zero curve is hard-coded. You can also create a zero curve by bootstrapping the zero curve from market data (for example, deposits, futures/forwards, and swaps). The hard-coded data for the zero curve is defined as:

```
Settle = datetime(2008,7,21);

% Zero Curve
CurveDates = daysadd(Settle,360*([1 3 5 7 10 20]),1);
ZeroRates = [1.9 2.6 3.1 3.5 4 4.3]'/100;

plot(CurveDates,ZeroRates)
datetick
title(['Zero Curve for ' datestr(Settle)]);
```




```
RateSpec = intenvset('Rates',ZeroRates,'EndDates',CurveDates,'StartDate',Settle);
```

Define Swaption Parameters

For this example, you compute the price of a 10-no-call-1 Bermudan swaption.

```
BermudanExerciseDates = daysadd(Settle,360*(1:9),1);
BermudanMaturity = datetime(2018,7,21);
BermudanStrike = .045;
```

Black's Model and the Swaption Volatility Matrix

Black's model is often used to price and quote European exercise interest-rate options, that is, caps, floors, and swaptions. In the case of swaptions, Black's model is used to imply a volatility given the current observed market price. The following matrix shows the Black implied volatility for a range of swaption exercise dates (columns) and underlying swap maturities (rows).

```
SwaptionBlackVol = [22 21 19 17 15 13 12
    21 19 17 16 15 13 11
    20 18 16 15 14 12 11
    19 17 15 14 13 12 10
    18 16 14 13 12 11 10
    15 14 13 12 12 11 10
    13 13 12 11 11 10 9]/100;
ExerciseDates = [1:5 7 10];
Tenors = [1:5 7 10];

EurExDatesFull = repmat(daysadd(Settle,ExerciseDates*360,1)',...
    length(Tenors),1);
EurMatFull = reshape(daysadd(EurExDatesFull,...
    repmat(360*Tenors,1,length(ExerciseDates)),1),size(EurExDatesFull));
```

Selecting the Calibration Instruments

Selecting the instruments to calibrate the model to is one of the tasks in calibration. For Bermudan swaptions, it is typical to calibrate to European swaptions that are co-terminal with the Bermudan swaption that you want to price. In this case, all swaptions having an underlying tenor that matures before the maturity of the swaption to be priced are used in the calibration.

```
% Find the swaptions that expire on or before the maturity date of the
% sample swaption
relidx = find(EurMatFull <= BermudanMaturity);
```

Compute Swaption Prices Using Black's Model

Swaption prices are computed using Black's Model. You can then use the swaption prices to compare the model's predicted values. To compute the swaption prices using Black's model:

```
% Compute Swaption Prices using Black's model
SwaptionBlackPrices = zeros(size(SwaptionBlackVol));
SwaptionStrike = zeros(size(SwaptionBlackVol));

for iSwaption=1:length(ExerciseDates)
    for iTenor=1:length(Tenors)
        [~,SwaptionStrike(iTenor,iSwaption)] = swapbyzero(RateSpec,[NaN 0], Settle, EurMatFull(i
            'StartDate',EurExDatesFull(iTenor,iSwaption),'LegReset',[1 1]);
        SwaptionBlackPrices(iTenor,iSwaption) = swaptionbyblk(RateSpec, 'call', SwaptionStrike(i
            EurExDatesFull(iTenor,iSwaption), EurMatFull(iTenor,iSwaption), SwaptionBlackVol(iTen
    end
end
```

Simulation Parameters

The following parameters are used where each exercise date is a simulation date.

```
nPeriods = 9;
DeltaTime = 1;
nTrials = 1000;

Tenor = (1:10)';

SimDates = daysadd(Settle,360*DeltaTime*(0:nPeriods),1);
SimTimes = diff(yearfrac(SimDates(1),SimDates));
```

Hull White 1 Factor Model

The Hull-White one-factor model describes the evolution of the short rate and is specified by the following:

$$dr = [\theta(t) - \alpha r]dt + \sigma dW$$

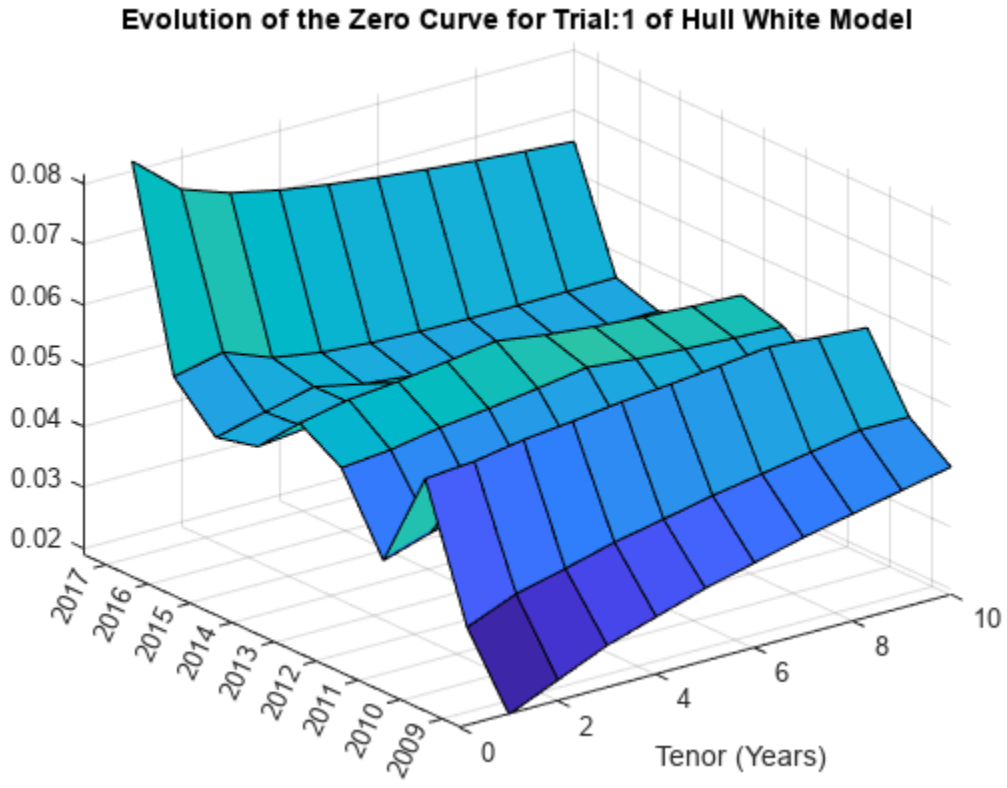
The Hull-White model is calibrated using the function `swaptionbyhw`, which constructs a trinomial tree to price the swaptions. Calibration consists of minimizing the difference between the observed market prices (computed above using the Black's implied swaption volatility matrix) and the model's predicted prices.

This example uses the Optimization Toolbox™ function `lsqnonlin` to find the parameter set that minimizes the difference between the observed and predicted values. However, other approaches (for example, simulated annealing) may be appropriate. Starting parameters and constraints for α and σ are set in the variables `x0` , `lb`, and `ub`; these could also be varied depending upon the particular calibration approach.

```
warnId = 'fininst:swaptionbyirtree:IgnoredSettle';
warnStruct = warning('off',warnId); % Turn warning off

TimeSpec = hwtimespec(Settle,daysadd(Settle,360*(1:11),1), 2);
HW1Fobjfun = @(x) SwaptionBlackPrices(relidx) - ...
    swaptionbyhw(hwtree(hwvolspec(Settle,datetime(2015,8,11),x(2),datetime(2015,8,11),x(1)), Rate
    EurExDatesFull(relidx), 0, EurExDatesFull(relidx), EurMatFull(relidx));
options = optimset('disp','iter','MaxFunEvals',1000,'TolFun',1e-5);

% Find the parameters that minimize the difference between the observed and
% predicted prices
x0 = [.1 .01];
lb = [0 0];
ub = [1 1];

HW1Fparams = lsqnonlin(HW1Fobjfun,x0,lb,ub,options);
```

```
                                        Norm of      First-order
 Iteration  Func-count     f(x)          step          optimality
     0          3        0.953772                         20.5
     1          6        0.142828      0.0169199          1.53
     2          9        0.123022      0.0146705          2.31
     3         12        0.122222      0.0154097         0.482
     4         15        0.122217     0.00131299       0.00409

Local minimum possible.
```

```
lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

warning(warnStruct); % Turn warnings on

HW_alpha = HW1Fparams(1);
HW_sigma = HW1Fparams(2);

% Construct the HullWhite1F model using the HullWhite1F constructor.
HW1F = HullWhite1F(RateSpec,HW_alpha,HW_sigma);

% Use Monte Carlo simulation to generate the interest-rate paths with
% HullWhite1F.simTermStructs.
HW1FSimPaths = HW1F.simTermStructs(nPeriods,'NTRIALS',nTrials,...
    'DeltaTime',DeltaTime,'Tenor',Tenor,'antithetic',true);

% Examine one simulation
trialIdx = 1;
figure
surf(Tenor,SimDates,HW1FSimPaths(:,:,trialIdx))
datetick y keepticks keeplimits
title(['Evolution of the Zero Curve for Trial:' num2str(trialIdx) ' of Hull White Model'])
xlabel('Tenor (Years)')
```




```
% Price the swaption using the helper function hBermudanSwaption
HW1FBermPrice = hBermudanSwaption(HW1FSimPaths,SimDates,Tenor,BermudanStrike,...
    BermudanExerciseDates,BermudanMaturity);
```

Linear Gaussian 2 Factor Model

The Linear Gaussian two-factor model (called the G2++ by Brigo and Mercurio) is also a short rate model, but involves two factors. Specifically:

$$r(t) = x(t) + y(t) + \varphi(t)$$

$$dx(t) = -ax(t)dt + \sigma dW_1(t)$$

$$dy(t) = -by(t)dt + \eta dW_2(t)$$

where $dW_1(t)dW_2(t)$ is a two-dimensional Brownian motion with correlation ρ

$$dW_1(t)dW_2(t) = \rho$$

and φ is a function chosen to match the initial zero curve.

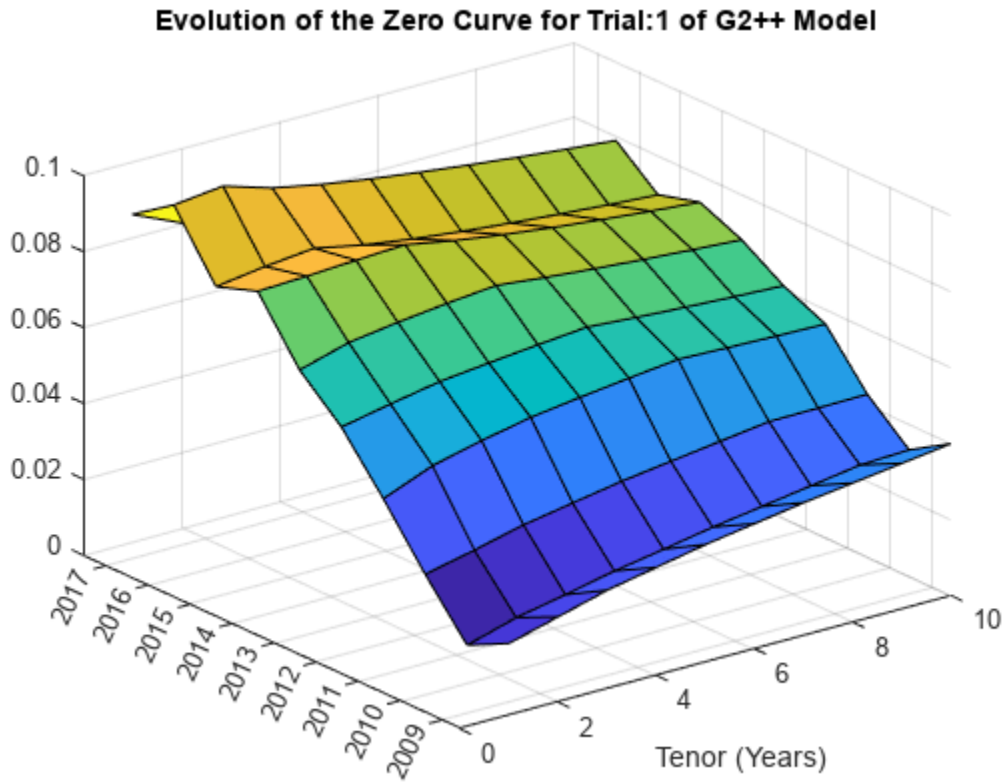
You can use the function `swaptionbylg2f` to compute analytic values of the swaption price for model parameters and to calibrate the model. Calibration consists of minimizing the difference between the observed market prices and the model's predicted prices.

```
% Calibrate the set of parameters that minimize the difference between the
% observed and predicted values using swaptionbylg2f and lsqnonlin.
G2PPobjfun = @(x) SwaptionBlackPrices(relidx) - ...
    swaptionbylg2f(RateSpec,x(1),x(2),x(3),x(4),x(5),SwaptionStrike(relidx),...
    EurExDatesFull(relidx),EurMatFull(relidx),'Reset',1);
x0 = [.2 .1 .02 .01 -.5];
lb = [0 0 0 0 -1];
ub = [1 1 1 1 1];
LG2Fparams = lsqnonlin(G2PPobjfun,x0,lb,ub,options);
```

```
                                        Norm of      First-order
 Iteration  Func-count     f(x)          step          optimality
     0          6         11.9838                         66.5
     1         12         1.35076      0.0967223           8.47
     2         18         1.35076       0.111744           8.47
     3         24        0.439341      0.0279361           1.29
     4         30        0.237917      0.0558721           3.49
     5         36        0.126732      0.0836846           7.51
     6         42       0.0395759      0.0137735           7.43
     7         48       0.0265828      0.0355772          0.787
     8         54       0.0252764       0.111744            0.5
     9         60       0.0228937       0.196793          0.338
    10         66       0.0222739       0.106678         0.0946
    11         72       0.0221799      0.0380101          0.912
    12         78       0.0221726      0.0163245           1.37

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.
```

```
LG2f_a = LG2Fparams(1);
LG2f_b = LG2Fparams(2);
LG2f_sigma = LG2Fparams(3);
LG2f_eta = LG2Fparams(4);
LG2f_rho = LG2Fparams(5);
```

```
% Create the G2PP object and use Monte Carlo simulation to generate the
% interest-rate paths with LinearGaussian2F.simTermStructs.
G2PP = LinearGaussian2F(RateSpec,LG2f_a,LG2f_b,LG2f_sigma,LG2f_eta,LG2f_rho);

G2PPSimPaths = G2PP.simTermStructs(nPeriods,'NTRIALS',nTrials,...
    'DeltaTime',DeltaTime,'Tenor',Tenor,'antithetic',true);

% Examine one simulation
trialIdx = 1;
figure
surf(Tenor,SimDates,G2PPSimPaths(:,:,trialIdx))
datetick y keepticks keeplimits
title(['Evolution of the Zero Curve for Trial:' num2str(trialIdx) ' of G2++ Model'])
xlabel('Tenor (Years)')
```




```
% Price the swaption using the helper function hBermudanSwaption
LG2FBermPrice = hBermudanSwaption(G2PPSimPaths,SimDates,Tenor,BermudanStrike,BermudanExerciseDate
```

LIBOR Market Model

The LIBOR Market Model (LMM) differs from short rate models in that it evolves a set of discrete forward rates. Specifically, the lognormal LMM specifies the following diffusion equation for each forward rate

$$\frac{dF_i(t)}{F_i} = -\mu_i dt + \sigma_i(t) dW_i$$

where

σ_i is the volatility function for each rate and dW is an `N` dimensional geometric Brownian motion with:

$$dW_i(t)dW_j(t) = \rho_{ij}$$

The LMM relates the drifts of the forward rates based on no-arbitrage arguments.

The choice with the LMM is how to model volatility and correlation and how to estimate the parameters of these models for volatility and correlation. In practice, you might use a combination of historical data (for example, observed correlation between forward rates) and current market data. For this example, only swaption data is used. Furthermore, many different parameterizations of the volatility and correlation exist. This example uses two relatively straightforward parameterizations.

One of the most popular functional forms in the literature for volatility is:

$$\sigma_i(t) = \phi_i(a(T_i - t) + b)e^{c(T_i - t)} + d$$

where ϕ adjusts the curve to match the volatility for the i^{th} forward rate. For this example, all of the Phi's will be taken to be `1`.

For the correlation, the following functional form is used:

$$\rho_{i,\,j} = e^{-\beta|i - j|}$$

Once the functional forms are specified, the parameters need to be estimated using market data. One useful approximation, initially developed by Rebonato, is the following, which computes the Black volatility for a European swaption, given a LMM with a set of volatility functions and a correlation matrix.

$$(v_{\alpha,\beta}^{LFM})^2 = \sum_{i,\,j\,=\,\alpha\,+\,1}^{\beta} \frac{w_i(0)w_j(0)F_i(0)F_j(0)\rho_{i,\,j}}{S_{\alpha,\beta}(0)^2}\int_0^{T\alpha}\sigma_i(t)\sigma_j(t)dt$$

where

$$w_i(t) = \frac{\tau_i P(t, T_i)}{\sum_{k\,=\,\alpha\,+\,1}^{\beta}\tau_k P(t, t_k)}$$

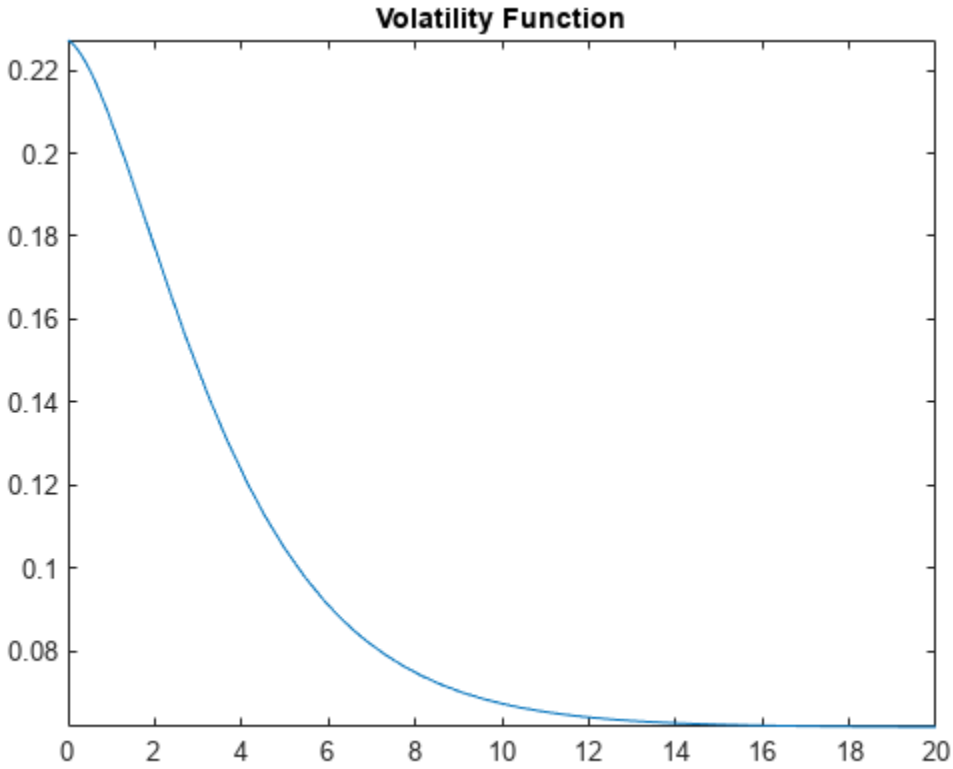
This calculation is done using `blackvolbyrebonato` to compute analytic values of the swaption price for model parameters and also to calibrate the model. Calibration consists of minimizing the difference between the observed implied swaption Black volatilities and the predicted Black volatilities.

```
nRates = 10;

CorrFunc = @(i,j,Beta) exp(-Beta*abs(i-j));

objfun = @(x) SwaptionBlackVol(relidx) - blackvolbyrebonato(RateSpec,...
    repmat({@(t) ones(size(t)).*(x(1)*t + x(2)).*exp(-x(3)*t) + x(4)},nRates-1,1),...
    CorrFunc(meshgrid(1:nRates-1)',meshgrid(1:nRates-1),x(5)),...
    EurExDatesFull(relidx),EurMatFull(relidx),'Period',1);

x0 = [.2 .05 1 .05 .2];
```

```
lb = [0 0 .5 0 .01];
ub = [1 1 2 .3 1];
LMMparams = lsqnonlin(objfun,x0,lb,ub,options);
```

```
                                         Norm of      First-order
 Iteration  Func-count     f(x)            step          optimality
     0          6        0.156251                         0.483
     1         12      0.00870177        0.188164         0.0339
     2         18      0.00463441        0.165527        0.00095
     3         24      0.00331055        0.351017         0.0154
     4         30      0.00294775       0.0892616       7.47e-05
     5         36      0.00281565        0.385779        0.00917
     6         42      0.00278988       0.0145632       4.15e-05
     7         48      0.00278522        0.115043        0.00116

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.
```

```
% Calculate VolFunc for the LMM object.
a = LMMparams(1);
b = LMMparams(2);
c = LMMparams(3);
d = LMMparams(4);

Beta = LMMparams(5);

VolFunc = repmat({@(t) ones(size(t)).*(a*t + b).*exp(-c*t) + d},nRates-1,1);

% Plot the volatility function
figure
fplot(VolFunc{1},[0 20])
title('Volatility Function')
```

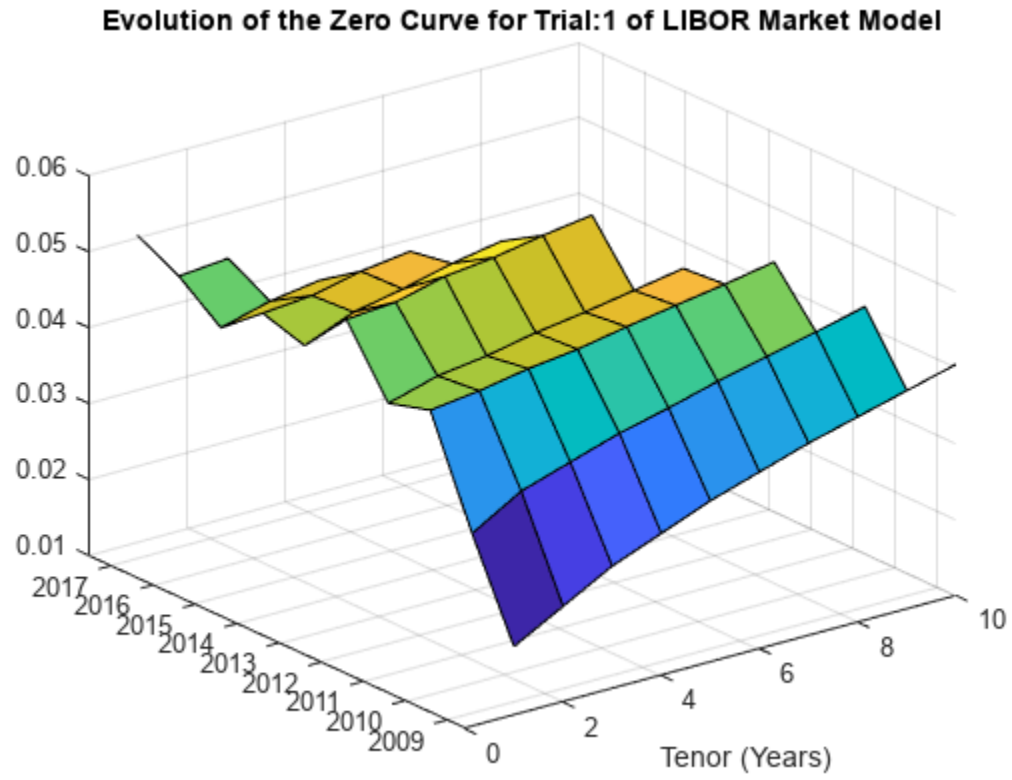



```
% Inspect the correlation matrix
CorrelationMatrix = CorrFunc(meshgrid(1:nRates-1)',meshgrid(1:nRates-1),Beta);
displayCorrelationMatrix(CorrelationMatrix);
```

```
Correlation Matrix
1.000 0.990 0.980 0.970 0.961 0.951 0.942 0.932 0.923
0.990 1.000 0.990 0.980 0.970 0.961 0.951 0.942 0.932
0.980 0.990 1.000 0.990 0.980 0.970 0.961 0.951 0.942
0.970 0.980 0.990 1.000 0.990 0.980 0.970 0.961 0.951
0.961 0.970 0.980 0.990 1.000 0.990 0.980 0.970 0.961
0.951 0.961 0.970 0.980 0.990 1.000 0.990 0.980 0.970
0.942 0.951 0.961 0.970 0.980 0.990 1.000 0.990 0.980
0.932 0.942 0.951 0.961 0.970 0.980 0.990 1.000 0.990
0.923 0.932 0.942 0.951 0.961 0.970 0.980 0.990 1.000
```

```
% Create the LMM object and use Monte Carlo simulation to generate the
% interest-rate paths with LiborMarketModel.simTermStructs.
LMM = LiborMarketModel(RateSpec,VolFunc,CorrelationMatrix,'Period',1);

[LMMZeroRates, ForwardRates] = LMM.simTermStructs(nPeriods,'nTrials',nTrials);

% Examine one simulation
trialIdx = 1;
figure
tmpPlotData = LMMZeroRates(:,:,trialIdx);
tmpPlotData(tmpPlotData == 0) = NaN;
surf(Tenor,SimDates,tmpPlotData)
```

```
title(['Evolution of the Zero Curve for Trial:' num2str(trialIdx) ' of LIBOR Market Model'])
xlabel('Tenor (Years)')
```




```
% Price the swaption using the helper function hBermudanSwaption
LMMTenor = 1:10;
LMMBermPrice = hBermudanSwaption(LMMZeroRates,SimDates,LMMTenor,.045,BermudanExerciseDates,Bermu
```

Results

```
displayResults(nTrials, nPeriods, HW1FBermPrice, LG2FBermPrice, LMMBermPrice);
```

```
     # of Monte Carlo Trials:     1000
     # of Time Periods/Trial:        9

HW1F Bermudan Swaption Price:   3.7577
LG2F Bermudan Swaption Price:   3.5576
 LMM Bermudan Swaption Price:   3.4911
```

Bibliography

This example is based on the following books, papers and journal articles:

1. Andersen, L. and V. Piterbarg. *Interest Rate Modeling*. Atlantic Financial Press, 2010.
2. Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice with Smile, Inflation and Credit*. Springer Verlag, 2007.
3. Glasserman, P. *Monte Carlo Methods in Financial Engineering*. Springer, 2003.

4 Hull, J. *Options, Futures, and Other Derivatives*. Prentice Hall, 2008.

5 Rebonato, R., K. McKay, and R. White. *The Sabr/Libor Market Model: Pricing, Calibration and Hedging for Complex Interest-Rate Derivatives*. John Wiley & Sons, 2010.

Utility Functions

```
function displayCorrelationMatrix(CorrelationMatrix)
fprintf('Correlation Matrix\n');
fprintf([repmat('%1.3f ',1,length(CorrelationMatrix)) ' \n'],CorrelationMatrix);
end

function displayResults(nTrials, nPeriods, HW1FBermPrice, LG2FBermPrice, LMMBermPrice)
fprintf('     # of Monte Carlo Trials: %8d\n'    , nTrials);
fprintf('     # of Time Periods/Trial: %8d\n\n'  , nPeriods);
fprintf('HW1F Bermudan Swaption Price: %8.4f\n', HW1FBermPrice);
fprintf('LG2F Bermudan Swaption Price: %8.4f\n', LG2FBermPrice);
fprintf(' LMM Bermudan Swaption Price: %8.4f\n', LMMBermPrice);
end
```

See Also

`capbyblk` | `floorbyblk` | `swaptionbyblk` | `blackvolbysabr` | `optsensbysabr` | `agencyoas` | `agencyprice` | `bndfutimprepo` | `bndfutprice` | `convfactor` | `tfutbyprice` | `tfutbyyield` | `tfutimprepo` | `tfutpricebyrepo` | `tfutyieldbyrepo` | `capbylg2f` | `floorbylg2f` | `swaptionbylg2f` | `blackvolbyrebonato` | `hwcalbycap` | `hwcalbyfloor`

Related Examples

- “Calibrate the SABR Model” on page 2-33
- “Price a Swaption Using the SABR Model” on page 2-38
- “Computing the Agency OAS for Bonds” on page 6-2
- “Analysis of Bond Futures” on page 7-12
- “Managing Interest-Rate Risk with Bond Futures” on page 2-125
- “Fitting the Diebold Li Model” on page 7-15
- “Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100

More About

- “Managing Present Value with Bond Futures” on page 7-14
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Managing Interest-Rate Risk with Bond Futures

This example shows how to hedge the interest-rate risk of a portfolio using bond futures.

Modifying the Duration of a Portfolio with Bond Futures

In managing a bond portfolio, you can use a benchmark portfolio to evaluate performance. Sometimes a manager is constrained to keep the portfolio's duration within a particular band of the duration of the benchmark. One way to modify the duration of the portfolio is to buy and sell bonds, however, there may be reasons why a portfolio manager wishes to maintain the existing composition of the portfolio (for example, the current holdings reflect fundamental research/views about future returns). Therefore, another option for modifying the duration is to buy and sell bond futures.

Bond futures are futures contracts where the commodity to be delivered is a government bond that meets the standard outlined in the futures contract (for example, the bond has a specified remaining time to maturity). Since often many bonds are available, and each bond may have a different coupon, you can use a conversion factor to normalize the payment by the long to the short.

There exist well developed markets for government bond futures. Specifically, the Chicago Board of Trade offers futures on the following:

- 2 Year Note
- 3 Year Note
- 5 Year Note
- 10 Year Note
- 30 Year Bond

https://www.cmegroup.com/trading/interest-rates/

Eurex offers futures on the following:

- Euro-Schatz Futures 1.75 to 2.25
- Euro-Bobl Futures 4.5 to 5.5
- Euro-Bund Futures 8.5 to 10.5
- Euro-Buxl Futures 24.0 to 35

https://www.eurex.com/ex-en/

Bond futures can be used to modify the duration of a portfolio. Since bond futures derive their value from the underlying instrument, the duration of a bond futures contract is related to the duration of the underlying bond.

There are two challenges in computing this duration:

- Since there are many available bonds for delivery, the short in the contract has a choice in which bond to deliver.
- Some contracts allow the short flexibility in choosing the delivery date.

Typically, the bond used for analysis is the bond that is cheapest for the short to deliver (CTD). One approach is to compute duration measures using the CTD's duration and the conversion factor.

For example, the Present Value of a Basis Point (PVBP) can be computed from the following:

$$PVBP_{Futures} = \frac{PVBP_{CTD}}{ConversionFactor_{CTD}}$$

$$PVBP_{CTD} = \frac{Duration_{CTD} * Price_{CTD}}{100}$$

Note that these definitions of duration for the futures contract are approximate, and do not account for the value of the delivery options for the short.

If the goal is to modify the duration of a portfolio, use the following:

$$NumContracts = \frac{(Dur_{Target} - Dur_{Initial}) * Value_{Portfolio}}{Dur_{CTD} * Price_{CTD} * ContractSize} * ConvFactor_{CTD}$$

Note that the contract size is typically for 100,000 face value of a bond -- so the contract size is typically 1000, as the bond face value is 100.

The following example assumes an initial duration, portfolio value, and target duration for a portfolio with exposure to the Euro interest rate. The June Euro-Bund Futures contract is used to modify the duration of the portfolio. Note that typically futures contracts are offered for March, June, September and December.

```
% Assume the following for the portfolio and target
PortfolioDuration = 6.4;
PortfolioValue = 100000000;
BenchmarkDuration = 4.8;

% Deliverable Bunds -- note that these conversion factors may also be
% computed with the MATLAB(R) function convfactor
BondPrice = [106.46;108.67;104.30];
BondMaturity = datenum({'04-Jan-2018','04-Jul-2018','04-Jan-2019'});
BondCoupon = [.04;.0425;.0375];
ConversionFactor = [.868688;.880218;.839275];

% Futures data -- found from http://www.eurex.com
FuturesPrice = 122.17;
FuturesSettle = '23-Apr-2009';
FuturesDelivery = '10-Jun-2009';

% To find the CTD bond we can compute the implied repo rate
ImpliedRepo = bndfutimprepo(BondPrice,FuturesPrice,FuturesSettle,...
    FuturesDelivery,ConversionFactor,BondCoupon,BondMaturity);

% Note that the bond with the highest implied repo rate is the CTD
[CTDImpRepo,CTDIndex] = max(ImpliedRepo);

% Compute the CTD's Duration -- note the period and basis for German Bunds
Duration = bnddurp(BondPrice,BondCoupon,FuturesSettle,BondMaturity,1,8);

ContractSize = 1000;

% Use the formula above to compute the number of contracts to sell
NumContracts = (BenchmarkDuration - PortfolioDuration)*PortfolioValue./...
    (BondPrice(CTDIndex)*ContractSize*Duration(CTDIndex))*ConversionFactor(CTDIndex);
```

```
disp(['To achieve the target duration, ' num2str(abs(round(NumContracts))) ...
    ' Euro-Bund Futures must be sold.'])
```

```
To achieve the target duration, 180 Euro-Bund Futures must be sold.
```

Modifying the Key Rate Durations of a Portfolio with Bond Futures

One of the shortcomings of using duration as a risk measure is that it assumes parallel shifts in the yield curve. While many studies have shown that this explains roughly 85% of the movement in the yield curve, changes in the slope or shape of the yield curve are not captured by duration, and therefore, hedging strategies are not successful at addressing these dynamics. One approach is to use key rate duration -- this is particularly relevant when using bond futures with multiple maturities, like Treasury futures.

The following example uses 2, 5, 10 and 30 year Treasury Bond futures to hedge the key rate duration of a portfolio. Computing key rate durations requires a zero curve. This example uses the zero curve published by the Treasury and found at the following location:

https://www.treasury.gov/resource-center/data-chart-center/interest-rates/Pages/TextView.aspx?data=yield

Note that this zero curve could also be derived using the Interest-Rate Curve functionality found in `IRDataCurve` and `IRFunctionCurve`.

```
% Assume the following for the portfolio and target, where the duration
% vectors are key rate durations at 2, 5, 10, and 30 years.
PortfolioDuration = [.5 1 2 6];
PortfolioValue = 100000000;
BenchmarkDuration = [.4 .8 1.6 5];

% The following are the CTD Bonds for the 30, 10, 5 and 2 year futures
% contracts -- these were determined using the procedure outlined in the
% previous section.
CTDCoupon = [4.75 3.125 5.125 7.5]'/100;
CTDMaturity = datenum({'3/31/2011','08/31/2013','05/15/2016','11/15/2024'});
CTDConversion = [0.9794 0.8953 0.9519 1.1484]';
CTDPrice = [107.34 105.91 117.00 144.18]';

ZeroRates = [0.07 0.10 0.31 0.50 0.99 1.38 1.96 2.56 3.03 3.99 3.89]'/100;
ZeroDates = daysadd(FuturesSettle,[30 360 360*2 360*3 360*5 ...
    360*7 360*10 360*15 360*20 360*25 360*30],1);

% Compute the key rate durations for each of the CTD bonds.
CTDKRD = bndkrdur([ZeroDates ZeroRates], CTDCoupon,FuturesSettle,...
    CTDMaturity,'KeyRates',[2 5 10 30]);

% Note that the contract size for the 2 Year Note Future is $200,000
ContractSize = [2000;1000;1000;1000];

NumContracts = (bsxfun(@times,CTDPrice.*ContractSize./CTDConversion,CTDKRD))\...
    (BenchmarkDuration - PortfolioDuration)'*PortfolioValue;

sprintf(['To achieve the target duration, \n' ...
    num2str(-round(NumContracts(1))) ' 2 Year Treasury Note Futures must be sold, \n' ...
    num2str(-round(NumContracts(2))) ' 5 Year Treasury Note Futures must be sold, \n' ...
    num2str(-round(NumContracts(3))) ' 10 Year Treasury Note Futures must be sold, \n' ...
    num2str(-round(NumContracts(4))) ' Treasury Bond Futures must be sold, \n'])
```

```
ans =
    'To achieve the target duration,
     24 2 Year Treasury Note Futures must be sold,
     47 5 Year Treasury Note Futures must be sold,
     68 10 Year Treasury Note Futures must be sold,
     120 Treasury Bond Futures must be sold,
     '
```

Improving the Performance of a Hedge with Regression

An additional component to consider in hedging interest-rate risk with bond futures, again related to movements in the yield curve, is that typically the yield curve moves more at the short end than at the long end.

Therefore, if a position is hedged with a future where the CTD bond has a maturity that is different than the portfolio this could lead to a situation where the hedge under- or over-compensates for the actual interest-rate risk of the portfolio One approach is to perform a regression on historical yields at different maturities to determine a Yield Beta, which is a value that represents how much more the yield changes for different maturities.

This example shows how to use this approach with UK Long Gilt futures and historical data on Gilt Yields.

Market data on Gilt futures is found at the following:

https://www.euronext.com

Historical data on gilts is found at the following;

https://www.dmo.gov.uk

Note that while this approach does offer the possibility of improving the performance of a hedge, any analysis using historical data depends on historical relationships remaining consistent.

Also note that an additional enhancement takes into consideration the correlation between different maturities. While this approach is outside the scope of this example, you can use this to implement a minimum variance hedge.

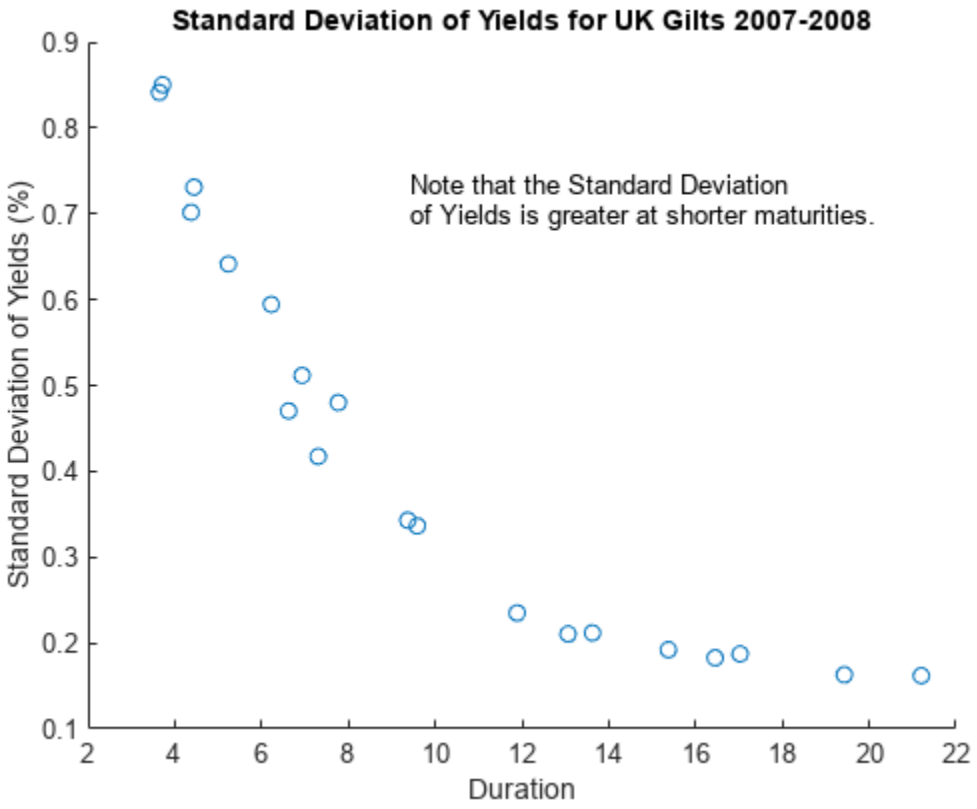
```
% Assume the following for the portfolio and target
PortfolioDuration = 6.4;
PortfolioValue = 100000000;
BenchmarkDuration = 4.8;

% This is the CTD Bond for the Long Gilt Futures contract
CTDBondPrice = 113.40;
CTDBondMaturity = datenum('7-Mar-2018');
CTDBondCoupon = .05;
CTDConversionFactor = 0.9325024;

% Market data for the Long Gilt Futures contract
FuturesPrice = 120.80;
FuturesSettle = '23-Apr-2009';
FuturesDelivery = '10-Jun-2009';

CTDDuration = bnddurp(CTDBondPrice,CTDBondCoupon,FuturesSettle,CTDBondMaturity);
```

```
ContractSize = 1000;

NumContracts = (BenchmarkDuration - PortfolioDuration)*PortfolioValue./...
    (CTDBondPrice*ContractSize*CTDDuration)*CTDConversionFactor;

disp(['To achieve the target duration with a conventional hedge ' ...
    num2str(-round(NumContracts)) ...
    ' Long Gilt Futures must be sold.'])
```

```
To achieve the target duration with a conventional hedge 182 Long Gilt Futures must be sold.
```

To improve the accuracy of this hedge, historical data is used to determine a relationship between the standard deviation of the yields. Specifically, standard deviation of yields is plotted and regressed vs bond duration. This relationship is then used to compute a Yield Beta for the hedge.

```
% Load data from XLS spreadsheet
load ukbonddata_20072008

Duration = bnddury(Yield(1,:)',Coupon,Dates(1,:),Maturity);

scatter(Duration,100*std(Yield))
title('Standard Deviation of Yields for UK Gilts 2007-2008')
ylabel('Standard Deviation of Yields (%)')
xlabel('Duration')
annotation(gcf,'textbox',[0.4067 0.685 0.4801 0.0989],...
    'String',{'Note that the Standard Deviation',...
    'of Yields is greater at shorter maturities.'},...
    'FitBoxToText','off',...
    'EdgeColor','none');
```




```
stats = regstats(std(Yield),Duration);
YieldBeta = (stats.beta'*[1 PortfolioDuration]')./(stats.beta'*[1 CTDDuration]');
```

Now the Yield Beta is used to compute a new value for the number of contracts to be sold. Note that since the duration of the portfolio was less than the duration of the CTD Gilt, the number of futures to sell is actually greater than in the first case.

```
NumContracts = (BenchmarkDuration - PortfolioDuration)*PortfolioValue./...
    (CTDBondPrice*ContractSize*CTDDuration)*CTDConversionFactor*YieldBeta;

disp(['To achieve the target duration using a Yield Beta-modified hedge, ' ...
    num2str(abs(round(NumContracts))) ...
    ' Long Gilt Futures must be sold.'])
```

```
To achieve the target duration using a Yield Beta-modified hedge, 193 Long Gilt Futures must be s
```

Bibliography

This example is based on the following books and papers:

[1] Burghardt, G., T. Belton, M. Lane and J. Papa. *The Treasury Bond Basis*. New York, NY: McGraw-Hill, 2005.

[2] Krgin, D. *Handbook of Global Fixed Income Calculations*. New York, NY: John Wiley & Sons, 2002.

[3] CFA Program Curriculum, Level III, Volume 4, Reading 31. CFA Institute, 2009.

See Also

`capbyblk` | `floorbyblk` | `swaptionbyblk` | `blackvolbysabr` | `optsensbysabr` | `agencyoas` | `agencyprice` | `bndfutimprepo` | `bndfutprice` | `convfactor` | `tfutbyprice` | `tfutbyyield` | `tfutimprepo` | `tfutpricebyrepo` | `tfutyieldbyrepo` | `capbylg2f` | `floorbylg2f` | `swaptionbylg2f` | `blackvolbyrebonato` | `hwcalbycap` | `hwcalbyfloor`

Related Examples

- “Calibrate the SABR Model” on page 2-33
- “Price a Swaption Using the SABR Model” on page 2-38
- “Computing the Agency OAS for Bonds” on page 6-2
- “Analysis of Bond Futures” on page 7-12
- “Managing Interest-Rate Risk with Bond Futures” on page 2-125
- “Fitting the Diebold Li Model” on page 7-15
- “Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100

More About

- “Managing Present Value with Bond Futures” on page 7-14
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Analyze Inflation-Indexed Instruments

This example shows how to analyze inflation-indexed instruments using Financial Toolbox™ and Financial Instruments Toolbox™.

Compute Real Prices and Yields for Inflation-Indexed Bonds

While inflation-indexed bonds have a great deal of variation in the design, for example, the length of the indexation lag, the majority of inflation-indexed bonds now have a three month lag. They are also capital-indexed, that is, the principal of the bond is indexed to inflation. Therefore, the coupon rate of the bond is constant, but the actual coupon payments vary as the principal of the bond is indexed to inflation.

Specifically, the indexation is done with the following ratio:

$$IndexRatio = \frac{CPI_{Ref}}{CPI_{Base}}$$

where CPI_{Base} is the level of the consumer price index (or equivalent price measure) at the time of the bond's issue and CPI_{Ref} is the reference CPI.

Typically, you compute the CPI_{Ref} by interpolating between the index data of a known inflation-index curve. To compute the cash flows for an inflation-indexed bond, you simply compute the appropriate reference CPI and Index Ratio.

The market convention for inflation-indexed bonds is to quote the price and yield using the actual (that is, unadjusted) coupon, which means that your quote is a real price and yield. To get a real price and yield, you can use the Financial Toolbox™ functions `bndprice` and `bndyield`. For example:

```
Price = 124 + 9/32;
Settle = datetime(2009,9,28);
Coupon = .03375;
Maturity = datetime(2032,4,15);

RealYield = bndyield(Price,Coupon,Settle,Maturity);
disp(['Real Yield: ', num2str(RealYield*100) '%'])
```

```
Real Yield: 2.0278%
```

Construct Nominal, Real, and Inflation Curves

With the advent of the inflation-indexed bond market, real curves can be constructed in a similar fashion to nominal curves. Using the available market data, you can construct the real curve and compare it to the nominal curve.

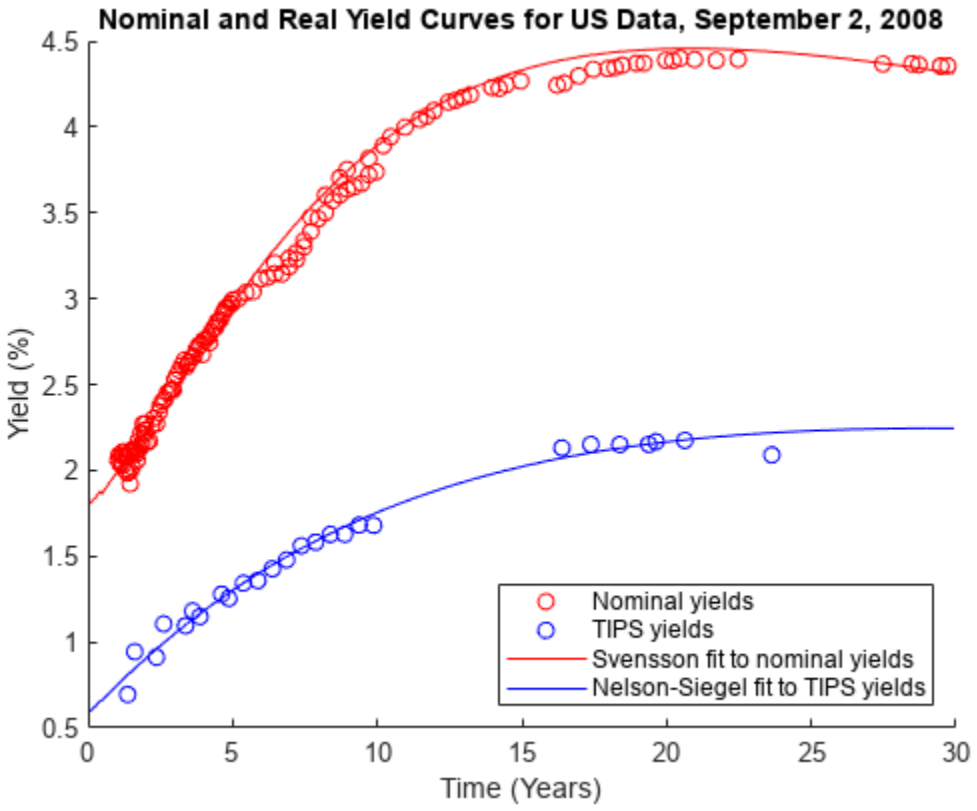
Note that one issue relates to the indexation lag of the bonds. As stated previously, typically the indexation lag is three months, which means that the inflation compensation is not actually matched up with the maturity or the coupon payments of the bond. While Anderson and Sleath [1] discuss an approach to resolving this discrepancy, for this example, the lag is simply noted.

You can use the `fitNelsonSiegel` and `fitSvensson` functions in the Financial Instruments Toolbox™ to create `parametercurve` objects that fit Nelson-Siegel and Svensson models to real and nominal yield curves in the US. The Nelson-Siegel model typically places restrictions on the model

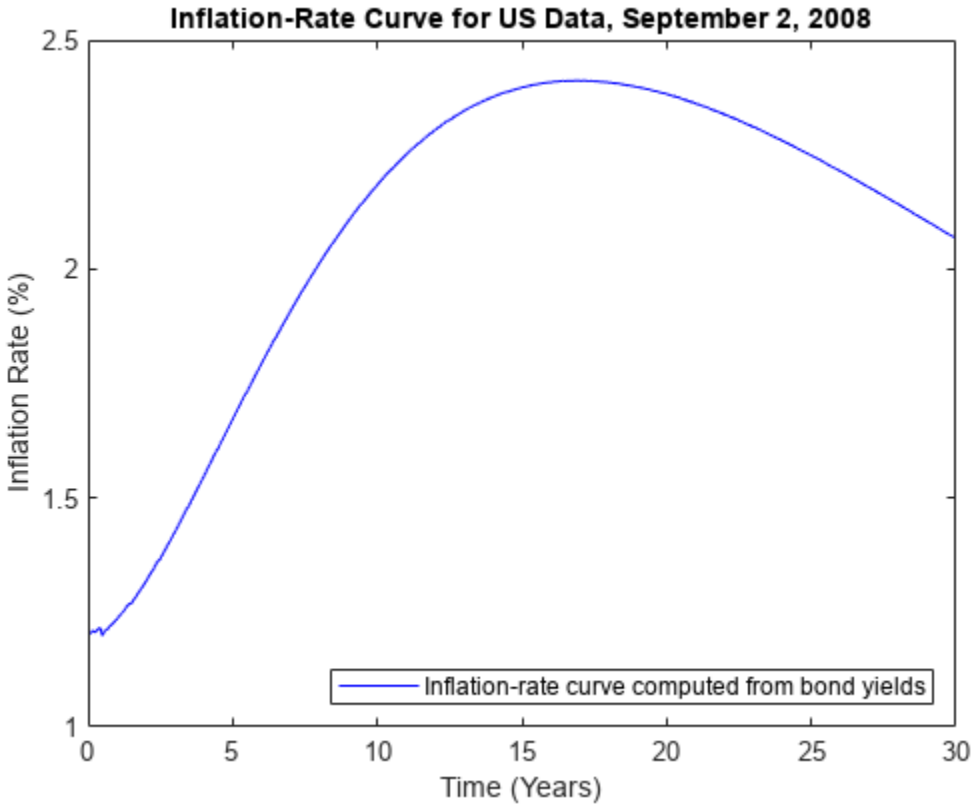
parameters to ensure that the interest rates are always positive. However, real interest rates can be negative, which means that these Nelson-Siegel restrictions are not used in the case below.

```
% Load the data.
load usbond_02Sep2008
Settle = datetime(2008, 9, 2);
NominalTimeToMaturity = yearfrac(Settle,NominalMaturity);
TIPSTimeToMaturity = yearfrac(Settle,TIPSMaturity);

% Compute the yields.
NominalYield = bndyield(NominalPrice,NominalCoupon,Settle,NominalMaturity);
TIPSYield = bndyield(TIPSPrice,TIPSCoupon,Settle,TIPSMaturity);

% Plot the yields.
scatter(NominalTimeToMaturity,NominalYield*100,'r');
hold on;
scatter(TIPSTimeToMaturity,TIPSYield*100,'b');

% Fit the real yield curve using fitNelsonSiegel.
nInst = numel(TIPSCoupon);
TIPSBonds(nInst,1) = fininstrument.FinInstrument;
for ii=1:nInst
    TIPSBonds(ii) = fininstrument("FixedBond",'Maturity',TIPSMaturity(ii),...
        'CouponRate',TIPSCoupon(ii));
end

TIPSNelsonSiegel = fitNelsonSiegel(Settle,TIPSBonds,TIPSPrice);

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

% Fit the nominal yield curve using fitSvensson.
nInst = numel(NominalCoupon);
NominalBonds(nInst,1) = fininstrument.FinInstrument;
for ii=1:nInst
    NominalBonds(ii) = fininstrument("FixedBond",'Maturity',NominalMaturity(ii),...
        'CouponRate',NominalCoupon(ii));
end

NominalSvensson = fitSvensson(Settle,NominalBonds,NominalPrice);

Solver stopped prematurely.

lsqnonlin stopped because it exceeded the function evaluation limit,
options.MaxFunctionEvaluations = 6.000000e+02.

% Plot the nominal and real yield curves.
PlotDates = (Settle+calmonths(1):calmonths(1):Settle+calyears(30)-1)';
PlotTimeToMaturity = yearfrac(Settle,PlotDates);

TIPSNelsonSiegelZeroRates = zerorates(TIPSNelsonSiegel,PlotDates);
TIPSNelsonSiegelParYields = zero2pyld(TIPSNelsonSiegelZeroRates,PlotDates,Settle, ...
    'InputCompounding', -1, 'OutputCompounding', 2);

NominalSvenssonZeroRates = zerorates(NominalSvensson,PlotDates);
NominalSvenssonParYields = zero2pyld(NominalSvenssonZeroRates,PlotDates,Settle, ...
    'InputCompounding', -1, 'OutputCompounding', 2);
```

```
plot(PlotTimeToMaturity,NominalSvenssonParYields*100,'r')
plot(PlotTimeToMaturity,TIPSNelsonSiegelParYields*100,'b')
hold off;

title('Nominal and Real Yield Curves for US Data, September 2, 2008')
xlabel('Time (Years)')
ylabel('Yield (%)')
legend({'Nominal yields','TIPS yields','Svensson fit to nominal yields',...
    'Nelson-Siegel fit to TIPS yields'},'location','southeast')
```




```
% Create an inflation-rate curve by subtracting the real curve from the
% nominal curve.
InflationRateCurve = ratecurve("zero", Settle, PlotDates, ...
    NominalSvenssonZeroRates - TIPSNelsonSiegelZeroRates);

figure
plot(PlotTimeToMaturity, zero2pyld(...
    zerorates(InflationRateCurve, PlotDates), PlotDates, Settle, ...
    'InputCompounding', -1, 'OutputCompounding', 2)*100,'b');

title('Inflation-Rate Curve for US Data, September 2, 2008')
xlabel('Time (Years)')
ylabel('Inflation Rate (%)')
legend({'Inflation-rate curve computed from bond yields'},'location','southeast')
```



Constructing Inflation Curves from Zero-Coupon Inflation Swaps

Inflation-linked derivatives have also experienced growth in the market. Some of the most liquidly traded inflation derivatives are zero coupon inflation swaps (`ZeroCouponInflationSwap`) and year-on-year inflation swaps (`YearYearInflationSwap`).

In a zero-coupon inflation swap, the inflation payer agrees to pay the rate of inflation at maturity (lagged by a certain amount) compounded by the number of years. The inflation receiver typically pays a fixed rate, again compounded by the tenor of the instrument. At the inception of the zero-coupon inflation swap, the fixed rate is set to the projected inflation rate for the life of the swap. This rate is called the "breakeven inflation swap rate" and it is quoted in the market [6].

Using the notation from Hurd and Relleen, you compute the rate as:

$$(1 + Rate_{swap})^T = (1 + Inflation_{t-L,\,t+T-L})^T$$

where t is the current time, T is the tenor, and L is the lag. [5]

At maturity, the actual cash flows of the zero-coupon inflation swap are:

$$FixedLeg = N \times [(1 + k)^M - 1]$$

$$InflationLeg = N \times \left[\frac{I(T_M)}{I_0} - 1\right]$$

where

- $N$is the reference notional of the swap.
- k is the fixed inflation rate.
- $M$is the number of years for the life of the swap.
- $I(T_M)$is the inflation index at the maturity date with some lag (for example, three months).
- I_0 is the inflation index at the start date with some lag (for example, three months).

While the fixed-leg cash flow might be different from the actual inflation-leg cash flow at maturity, the fixed breakeven inflation swap rate of the zero-coupon inflation swap represents the projected inflation rate for the tenor of the swap at inception. You can build an inflation curve from a series of breakeven zero-coupon inflation swap rates starting on the same date and maturing on different dates. Here, the dates are already adjusted with the appropriate indexation lag to simplify the notation:

$$I(0,T_{1Y}) = I(T_0)(1 + b(0;T_0,T_{1Y}))^{T_{1Y} - T_0}$$

$$I(0,T_{2Y}) = I(T_0)(1 + b(0;T_0,T_{2Y}))^{T_{2Y} - T_0}$$

$$I(0,T_{3Y}) = I(T_0)(1 + b(0;T_0,T_{3Y}))^{T_{3Y} - T_0}$$

. . .

$$I(0,T_i) = I(T_0)(1 + b(0;T_0,T_i))^{T_i - T_0}$$

where

- $I(0,T_i)$ is the breakeven inflation index reference number for maturity date T_i.
- $I(T_0)$ is the base inflation index value for the starting date T_0.
- $b(0;T_0,T_i)$ is the breakeven inflation rate for the zero-coupon inflation swap maturing on T_i.

You can get your inflation curve this by using the `inflationbuild` function to create an `inflationcurve` object. To build an `inflationcurve` from zero-coupon inflation swap rates, first define the base inflation date and the corresponding base inflation-index value.

```
% Define the base inflation date and index value for the inflation-index
% curve.
BaseDate = datetime(2020,6,1);
BaseIndexValue = 100;
```

Then, define the zero-coupon inflation swap rates and the corresponding maturity dates already adjusted with the appropriate indexation lag.

```
% Define the zero-coupon inflation swap rates and maturity dates.
ZCISTimes = (calyears([1 2 3 4 5 7 10 20 30]))';
ZCISRates = [0.42 0.54 0.76 0.87 0.92 1.39 1.71 2.01 2.46]'./100
```

```
ZCISRates = 9×1

    0.0042
    0.0054
    0.0076
    0.0087
```

```
    0.0092
    0.0139
    0.0171
    0.0201
    0.0246
```

```
ZCISDates = BaseDate + ZCISTimes
```

```
ZCISDates = 9x1 datetime
   01-Jun-2021
   01-Jun-2022
   01-Jun-2023
   01-Jun-2024
   01-Jun-2025
   01-Jun-2027
   01-Jun-2030
   01-Jun-2040
   01-Jun-2050
```

In pricing inflation derivatives and building inflation curves, incorporating seasonality can be a critical factor. The zero-coupon inflation swap rates typically have maturities that increase in whole number of years. As a result, the inflation curve is typically built from zero-coupon inflation swap rates on an annual basis. However, when computing inflation-index values for monthly periods that are not whole number of years, you can make seasonal adjustments to reflect the seasonal patterns of inflation within the year. These 12 monthly seasonal rates are annualized and they add up to zero to ensure that the cumulative seasonal adjustments are reset to zero every year. In the `inflationbuild` function and the `inflationcurve` object, you define these seasonal rates using the `'Seasonality'` name-value pair argument and they are internally corrected to ensure that they add to zero.

```
% Define the 12 monthly seasonal rates.
%
% Months:
%    Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
%     1     2     3     4     5     6     7     8     9    10    11    12
% Seasonal Rates (percent):
%   -6.34 -3.00 -1.34  3.34  5.34  3.66  8.66  5.66 -2.34 -2.66 -4.66 -6.32
SeasonalRates = [-6.34 -3.00 -1.34 3.34 5.34 3.66 8.66 5.66 -2.34 -2.66 -4.66 -6.32]./100
```

```
SeasonalRates = 1×12

   -0.0634   -0.0300   -0.0134    0.0334    0.0534    0.0366    0.0866    0.0566   -0.0234   -0.0
```

```
% Build an inflation-index curve from zero-coupon inflation swap rates.
myInflationCurve = inflationbuild(BaseDate, BaseIndexValue, ...
    ZCISDates, ZCISRates, 'Seasonality', SeasonalRates)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
     InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
```

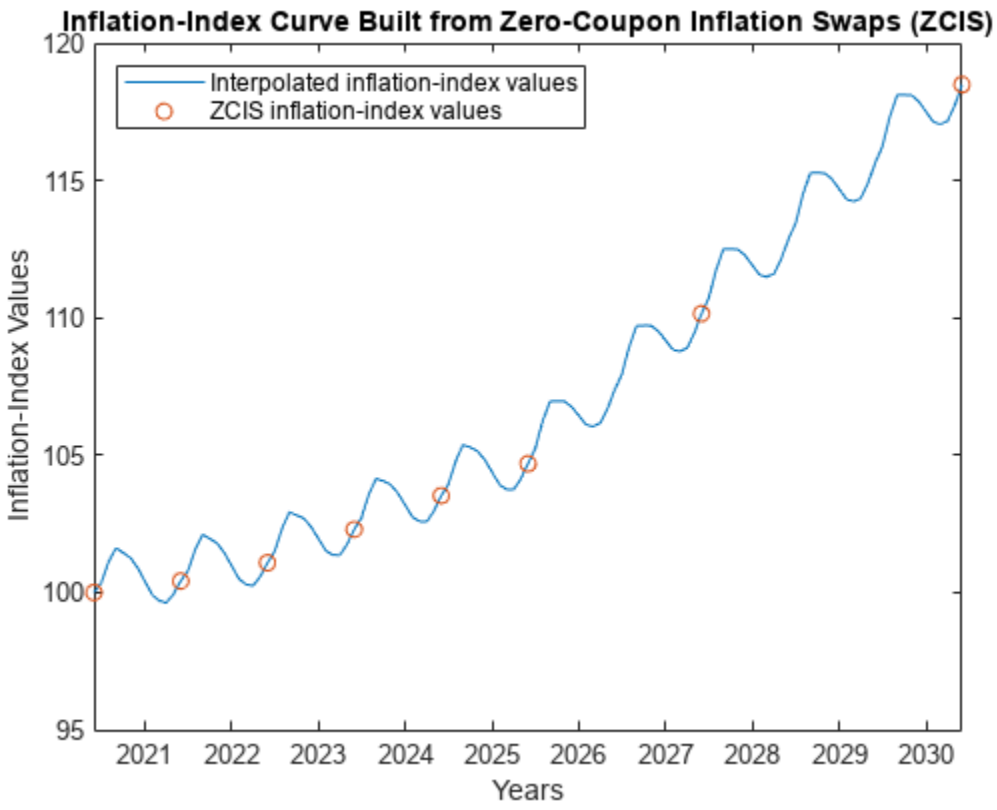
```
            Seasonality: [12x1 double]
```

Once you have created the `inflationcurve` object, compute the inflation-index values for each month using `indexvalues`.

```
% Compute the inflation-index values.
IndexPlotDates = (BaseDate:calmonths(1):BaseDate+calyears(10))';
IndexPlotValues = indexvalues(myInflationCurve, IndexPlotDates);
```

To visualize the seasonal patterns of inflation that occur within each year, plot the computed inflation-index values.

```
% Plot the inflation-index curve.
figure; plot(IndexPlotDates, IndexPlotValues)
hold on;
plot(myInflationCurve.Dates(1:8), myInflationCurve.InflationIndexValues(1:8), 'o')
hold off;

title('Inflation-Index Curve Built from Zero-Coupon Inflation Swaps (ZCIS)')
xlabel('Years')
ylabel('Inflation-Index Values')
legend({'Interpolated inflation-index values','ZCIS inflation-index values'},'location','northwes
```



Price Inflation-Indexed Instruments Using an Inflation Curve

With the `inflationcurve` object created, you can price inflation-indexed instruments such as zero-coupon inflation swaps (`ZeroCouponInflationSwap`), year-on-year inflation swaps (`YearYearInflationSwap`), and inflation-indexed bonds (`InflationBond`).

First, create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,25);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0043 0.0051 0.0062 0.0072 0.0096 0.0121 0.0172 0.0241 0.0302 0.0308]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 25-Sep-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Using the `ratecurve` and `inflationcurve` objects as inputs, create an `Inflation` pricer object using `finpricer`.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Create an `InflationBond` instrument using `fininstrument`.

```
IssueDate = datetime(2020,9,20);
Maturity = datetime(2025,9,20);
CouponRate = 0.023;

InflationBond = fininstrument("InflationBond",'IssueDate',IssueDate,'Maturity',Maturity,'CouponRa
```

```
InflationBond =
  InflationBond with properties:

                  CouponRate: 0.0230
                      Period: 2
                       Basis: 0
                   Principal: 100
    DaycountAdjustedCashFlow: 0
                         Lag: 3
```

```
BusinessDayConvention: "actual"
              Holidays: NaT
          EndMonthRule: 1
             IssueDate: 20-Sep-2020
       FirstCouponDate: NaT
        LastCouponDate: NaT
              Maturity: 20-Sep-2025
                  Name: ""
```

Here, the default indexation lag is three months and the bond issue date is `20-Sep-2020`. The first date on the inflation curve of the pricer must be on or before `20-Jun-2020` to price this instrument. In this example, the first date on the inflation curve of the pricer is `01-Jun-2020`.

Price the `InflationBond` instrument by using the `price` function for the `Inflation` pricer.

```
InflationBondPrice = price(outPricer, InflationBond)
```

```
InflationBondPrice = 110.1314
```

References

This example is based on the following papers and journal articles:

[1] Anderson N. and J. Sleath. "New Estimates of the UK Real and Nominal Yield Curves." Bank of England, working paper 126, 2001.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice: With Smile, Inflation and Credit*. Springer, 2006.

[3] Deacon, M., A. Derry, and D. Mirfendereski. *Inflation-Indexed Securities: Bonds, Swaps, and Other Derivatives*. Wiley Finance, 2004.

[4] Gurkaynak, R. S., B.P. Sack, and J.H. Wright. "The TIPS Yield Curve and Inflation Compensation." FEDS Working Paper No. 2008-05, October 2008.

[5] Hurd, M. and J. Relleen. "New Information from Inflation Swaps and Index-linked Bonds." Quarterly Bulletin, Spring 2006.

[6] Kerkhof, J. "Inflation Derivatives Explained." Lehman Brothers, 2005.

See Also

`fininstrument` | `finpricer` | `ratecurve` | `inflationbuild` | `indexvalues` | `inflationcurve` | `InflationBond` | `YearYearInflationSwap` | `ZeroCouponInflationSwap` | `Inflation`

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53

Bootstrapping a Swap Curve

This example shows how to bootstrap an interest-rate curve, often referred to as a swap curve, using the `IRDataCurve` object. The static `bootstrap` method takes as inputs a cell array of market instruments (which can be deposits, interest-rate futures, swaps, and bonds) and bootstraps an interest-rate curve of either the forward or the zero curve. It is also possible to specify multiple interpolation methods, including piecewise constant, linear, and Piecewise Cubic Hermite Interpolating Polynomial (PCHIP).

Obtain Data

A curve is bootstrapped from market data. In this example, you bootstrap a swap curve from deposits, Eurodollar Futures, and swaps.

For this example, the input market data is hard-coded and specified as 2 cell arrays of data, one which indicates the type of instrument and a second cell array containing the `Settle`, `Maturity`, and Market Quote for the instrument. For deposits and swaps, the quote is a rate, and for the EuroDollar Futures, the quote is a price. Although bonds are not used in this example, a bond would be quoted with a price.

```
InstrumentTypes = {'Deposit';'Deposit';'Deposit';'Deposit';'Deposit'; ...
    'Futures';'Futures'; ...
    'Futures';'Futures';'Futures'; ...
    'Futures';'Futures';'Futures'; ...
    'Futures';'Futures';'Futures'; ...
    'Futures';'Futures';'Futures'; ...
    'Futures';'Futures';'Futures'; ...
    'Swap';'Swap';'Swap';'Swap';'Swap';'Swap';'Swap'};

Instruments = [datenum('08/10/2007'),datenum('08/17/2007'),.0532063; ...
    datenum('08/10/2007'),datenum('08/24/2007'),.0532000; ...
    datenum('08/10/2007'),datenum('09/17/2007'),.0532000; ...
    datenum('08/10/2007'),datenum('10/17/2007'),.0534000; ...
    datenum('08/10/2007'),datenum('11/17/2007'),.0535866; ...
    datenum('08/08/2007'),datenum('19-Dec-2007'),9485; ...
    datenum('08/08/2007'),datenum('19-Mar-2008'),9502; ...
    datenum('08/08/2007'),datenum('18-Jun-2008'),9509.5; ...
    datenum('08/08/2007'),datenum('17-Sep-2008'),9509; ...
    datenum('08/08/2007'),datenum('17-Dec-2008'),9505.5; ...
    datenum('08/08/2007'),datenum('18-Mar-2009'),9501; ...
    datenum('08/08/2007'),datenum('17-Jun-2009'),9494.5; ...
    datenum('08/08/2007'),datenum('16-Sep-2009'),9489; ...
    datenum('08/08/2007'),datenum('16-Dec-2009'),9481.5; ...
    datenum('08/08/2007'),datenum('17-Mar-2010'),9478; ...
    datenum('08/08/2007'),datenum('16-Jun-2010'),9474; ...
    datenum('08/08/2007'),datenum('15-Sep-2010'),9469.5; ...
    datenum('08/08/2007'),datenum('15-Dec-2010'),9464.5; ...
    datenum('08/08/2007'),datenum('16-Mar-2011'),9462.5; ...
    datenum('08/08/2007'),datenum('15-Jun-2011'),9456.5; ...
    datenum('08/08/2007'),datenum('21-Sep-2011'),9454; ...
    datenum('08/08/2007'),datenum('21-Dec-2011'),9449.5; ...
    datenum('08/08/2007'),datenum('08/08/2014'),.0530; ...
    datenum('08/08/2007'),datenum('08/08/2017'),.0545; ...
    datenum('08/08/2007'),datenum('08/08/2019'),.0551; ...
    datenum('08/08/2007'),datenum('08/08/2022'),.0559; ...
```

```
    datenum('08/08/2007'),datenum('08/08/2027'),.0565; ...
    datenum('08/08/2007'),datenum('08/08/2032'),.0566; ...
    datenum('08/08/2007'),datenum('08/08/2037'),.0566];
```

Construct the Curve Using Bootstrapping

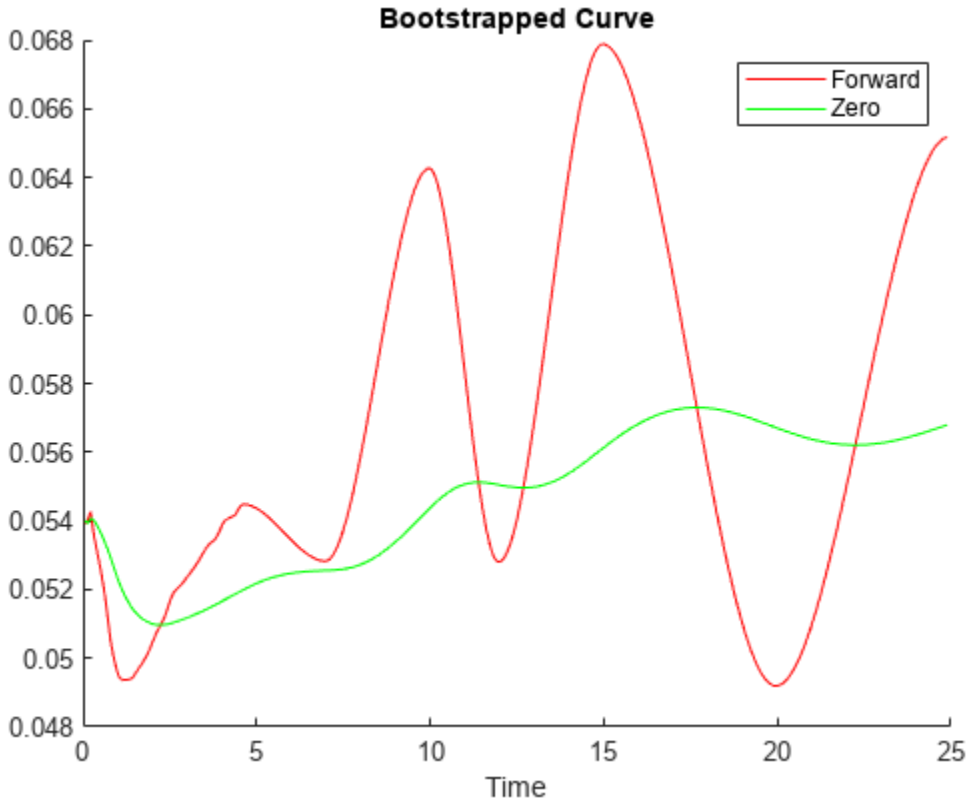
The `bootstrap` method is called as a static method of the `IRDataCurve` class. Inputs to this method include the curve type (Zero or Forward), settle date, instrument types, instrument data, and optional arguments including an interpolation method, compounding, and an options structure for bootstrapping. Note that in this example, you pass in an `IRBootstrapOptions` object which includes information for the convexity adjustment to forward rates.

```
IRsigma = .01;
CurveSettle = datenum('08/10/2007');
bootModel = IRDataCurve.bootstrap('Forward', CurveSettle, ...
    InstrumentTypes, Instruments,'InterpMethod','pchip',...
    'Compounding',-1,'IRBootstrapOptions',...
    IRBootstrapOptions('ConvexityAdjustment',@(t) .5*IRsigma^2.*t.^2));
```

Plot Curves

Plot both the forward and zero curves.

```
PlottingDates = (CurveSettle+20:30:CurveSettle+365*25)';
TimeToMaturity = yearfrac(CurveSettle,PlottingDates);
BootstrappedForwardRates = bootModel.getForwardRates(PlottingDates);
BootstrappedZeroRates = bootModel.getZeroRates(PlottingDates);

figure
hold on
plot(TimeToMaturity,BootstrappedForwardRates,'r')
plot(TimeToMaturity,BootstrappedZeroRates,'g')
title('Bootstrapped Curve')
xlabel('Time')
legend({'Forward','Zero'})
```



Bibliography

This example draws from the following papers and journal articles:

[1] Hagan, P., West, G. (2006). "Interpolation Methods for Curve Construction." *Applied Mathematical Finance*, Vol 13, No. 2

[2] Ron, Uri (2000). "A Practical Guide to Swap Curve Construction." Working Papers 00-17, Bank of Canada.

Fitting Interest-Rate Curve Functions

This example shows how to use `IRFunctionCurve` objects to model the term structure of interest rates (also referred to as the yield curve). This can be contrasted with modeling the term structure with vectors of dates and data and interpolating between the points (which can currently be done with the function `prbyzero`). The term structure can refer to at least three different curves: the discount curve, zero curve, or forward curve.

The `IRFunctionCurve` object allows you to model an interest-rate curve as a function.

This example explores using an `IRFunctionCurve` object to model the default-free term structure of interest rates in the United Kingdom. Three different forms for the term structure are implemented and are discussed in more detail later:

- Nelson-Siegel
- Svensson
- Smoothing Cubic Spline with a so-called Variable Roughness Penalty (VRP)

Choosing the Data

The first question in modeling the yield curve is what data should be used. To model a default-free yield curve, default-free, option-free market instruments must be used. The most significant component of the data is UK Government Bonds (known as Gilts). Historical data is retrieved from the following site:

https://www.dmo.gov.uk

Repo data is used to construct the short end of the yield curve. Repo data is retrieved from the following site:

https://www.ukfinance.org.uk/

Note also that the data must be specified as a matrix where the columns are `Settle`, `Maturity`, `CleanPrice`, and `CouponRate` and that instruments must be bonds or synthetically converted to bonds.

Market data for a close date of April 30, 2008, has been downloaded and saved to the following data file (`ukdata20080430`), which is loaded into MATLAB® with the following command:

```
% Load the data
load ukdata20080430

% Convert repo rates to be equivalent zero coupon bonds
RepoCouponRate = repmat(0,size(RepoRates));
RepoPrice = bndprice(RepoRates, RepoCouponRate, RepoSettle, RepoMaturity);

% Aggregate the data
Settle = [RepoSettle;BondSettle];
Maturity = [RepoMaturity;BondMaturity];
CleanPrice = [RepoPrice;BondCleanPrice];
CouponRate = [RepoCouponRate;BondCouponRate];
Instruments = [Settle Maturity CleanPrice CouponRate];
InstrumentPeriod = [repmat(0,6,1);repmat(2,31,1)];
```

```
CurveSettle = datenum('30-Apr-2008');
```

Fit Nelson-Siegel Model to Market Data

The Nelson-Siegel model proposes that the instantaneous forward curve can be modeled with the following:

$$f = \beta_0 + \beta_1 e^{\frac{-m}{\tau}} + \beta_2 e^{\frac{-m}{\tau}} \frac{m}{\tau}$$

This can be integrated to derive an equation for the zero curve (see [6] for more information on the equations and the derivation):

$$s = \beta_0 + (\beta_1 + \beta_2)\frac{\tau}{m}(1 - e^{\frac{-m}{\tau}}) - \beta_2 e^{\frac{-m}{\tau}}$$

See [1 on page 2-148] for more information.

The `IRFunctionCurve` object provides the capability to fit a Nelson Siegel curve to observed market data with the `fitNelsonSiegel` method. The fitting is done by calling the Optimization Toolbox™ function `lsqnonlin`.

The `fitNelsonSiegel` function has required inputs for Curve `Type`, Curve `Settle`, and a matrix of instrument data.

Optional input arguments, specified in name-value pair argument, are:

- `IRFitOptions` structure: Provides the capability to choose which quantity to be minimized (price, yield, or duration weighted price) and other optimization parameters (for example, upper and lower bounds for parameters).
- Curve `Compounding` and `Basis` (day-count convention)
- Additional instrument parameters, `Period`, `Basis`, `FirstCouponDate`, and so on.

```
NSModel = IRFunctionCurve.fitNelsonSiegel('Zero',CurveSettle,...
    Instruments,'InstrumentPeriod',InstrumentPeriod);
```

Fit Svensson Model

A very similar model to the Nelson-Siegel model is the Svensson model, which adds two additional parameters to account for greater flexibility in the term structure. This model proposes that the forward rate can be modeled with the following form:

$$f = \beta_0 + \beta_1 e^{\frac{-m}{\tau_1}} + \beta_2 e^{\frac{-m}{\tau_1}} \frac{m}{\tau_1} + \beta_3 e^{\frac{-m}{\tau_2}} \frac{m}{\tau_2}$$

As above, this can be integrated to derive an equation for the zero curve:

$$s = \beta_0 + \beta_1(1 - e^{\frac{-m}{\tau_1}})(-\frac{\tau_1}{m}) + \beta_2((1 - e^{\frac{-m}{\tau_1}})\frac{\tau_1}{m} - e^{\frac{m}{\tau_1}}) + \beta_3((1 - e^{\frac{-m}{\tau_2}})\frac{\tau_2}{m} - e^{\frac{m}{\tau_2}})$$

See [2 on page 2-148] for more information.

Fitting the parameters to this model proceeds in a similar fashion to the Nelson-Siegel model using the `fitSvensson` function.

```
SvenssonModel = IRFunctionCurve.fitSvensson('Zero',CurveSettle,...
    Instruments,'InstrumentPeriod',InstrumentPeriod);
```

Fit Smoothing Spline

The term structure can also be modeled with a spline, specifically, one way to model the term structure is by representing the forward curve with a cubic spline. To ensure that the spline is sufficiently smooth, a penalty is imposed relating to the curvature (second derivative) of the spline:

$$\sum_{i=1}^{N}[\frac{P_i - \widehat{P}_i(f)}{D_i}]^2 + \int_0^M \lambda_t(m)[f''(m)]^2 dm$$

where the first term is the difference between the observed price *P* and the predicted price, *P_hat*, (weighted by the bond's duration, *D*) summed over all bonds in the data set, and the second term is the penalty term (where *lambda* is a penalty function and *f* is the spline).

See [3 on page 2-148], [4 on page 2-148], [5 on page 2-149] below.

There have been different proposals for the specification of the penalty function *lambda*. One approach, advocated by [4 on page 2-148], and currently used by the UK Debt Management Office, is a penalty function of the following form:

$$log(\lambda(m)) = L - (L - S)e^{\frac{-m}{\mu}}$$

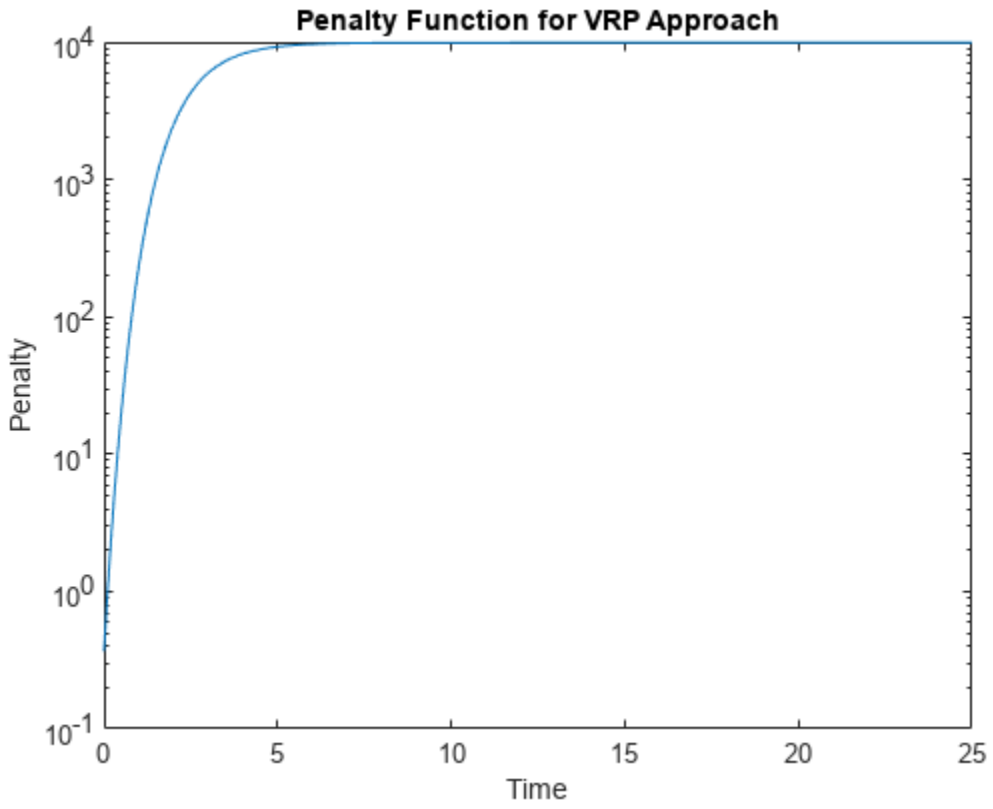
The parameters *L*, *S*, and *mu* are typically estimated from historical data.

The `IRFunctionCurve` object can be used to fit a smoothing spline representation of the forward curve with a penalty function using the function `fitSmoothingSpline`.

Required inputs, like for the functions above, are a `CurveType`, Curve`Settle`, `Instruments` matrix, and a function handle (`Lambdafun`) containing the penalty function.

The optional parameters are similar to `fitNelsonSiegel` and `fitSvensson`.

```
% Parameters chosen to be roughly similar to [4] below.
L = 9.2;
S = -1;
mu = 1;

lambdafun = @(t) exp(L - (L-S)*exp(-t/mu)); % Construct penalty function
t = 0:.1:25; % Construct data to plot penalty function
y = lambdafun(t);
figure
semilogy(t,y);
title('Penalty Function for VRP Approach')
ylabel('Penalty')
xlabel('Time')
```

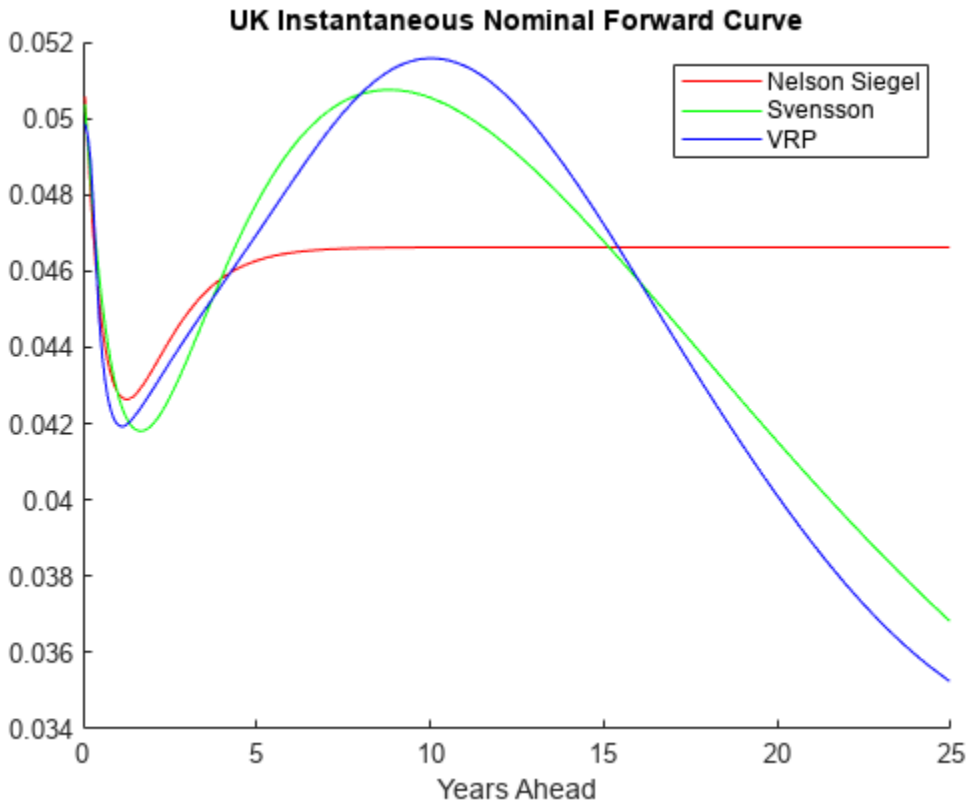



```
VRPModel = IRFunctionCurve.fitSmoothingSpline('Forward',CurveSettle,...
    Instruments,lambdafun,'Compounding',-1,...
    'InstrumentPeriod',InstrumentPeriod);
```

Use Fitted Curves and Plot Results

Once a curve is created, functions are used to extract the Forward and Zero Rates and the Discount Factors. This curve can also be converted into a `RateSpec` structure using the `toRateSpec` function. The `RateSpec` can then be used with many other functions in the Financial Instruments Toolbox™

```
PlottingDates = CurveSettle+20:30:CurveSettle+365*25;
TimeToMaturity = yearfrac(CurveSettle,PlottingDates);

NSForwardRates = NSModel.getForwardRates(PlottingDates);
SvenssonForwardRates = SvenssonModel.getForwardRates(PlottingDates);
VRPForwardRates = VRPModel.getForwardRates(PlottingDates);

figure
hold on
plot(TimeToMaturity,NSForwardRates,'r')
plot(TimeToMaturity,SvenssonForwardRates,'g')
plot(TimeToMaturity,VRPForwardRates,'b')
title('UK Instantaneous Nominal Forward Curve')
xlabel('Years Ahead')
legend({'Nelson Siegel','Svensson','VRP'})
```



Compare with this Link

This link provides a live look at the derived yield curve published by the UK

https://www.bankofengland.co.uk

Bibliography

This example is based on the following papers and journal articles:

[1] Nelson, C.R., Siegel, A.F. "Parsimonious Modelling of Yield Curves." *Journal of Business.* 60, pp 473-89, 1987.

[2] Svensson, L.E.O. "Estimating and Interpreting Forward Interest Rates: Sweden 1992-4." International Monetary Fund, IMF Working Paper, 1994/114, 1994.

[3] Fisher, M., Nychka, D., Zervos, D. "Fitting the Term Structure of Interest Rates with Smoothing Splines." Board of Governors of the Federal Reserve System, Federal Reserve Board Working Paper, 95-1, 1995.

[4] Anderson, N., Sleath, J. "New Estimates of the UK Real and Nominal Yield Curves." *Bank of England Quarterly Bulletin.* November, pp 384-92, 1999.

[5] Waggoner, D. "Spline Methods for Extracting Interest Rate Curves from Coupon Bond Prices." Federal Reserve Board Working Paper, 97-10, 1997.

[6] "Zero-Coupon Yield Curves: Technical Documentation." BIS Papers No. 25, October 2005.

[7] Bolder, D.J., Gusba,S. "Exponentials, Polynomials, and Fourier Series: More Yield Curve Modelling at the Bank of Canada." Working Papers 02-29, Bank of Canada, 2002.

[8] Bolder, D.J., Streliski, D. "Yield Curve Modelling at the Bank of Canada." Technical Reports 84, Bank of Canada, 1999.

See Also

More About

- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Fitting the Diebold Li Model

This example shows how to construct a Diebold Li model of the US yield curve for each month from 1990 to 2010. This example also demonstrates how to forecast future yield curves by fitting an autoregressive model to the time series of each parameter.

The paper can be found here:

https://www.nber.org/papers/w10048

Load the Data

The data used are monthly Treasury yields from 1990 through 2010 for tenors of 1 Mo, 3 Mo, 6 Mo, 1 Yr, 2 Yr, 3 Yr, 5 Yr, 7 Yr, 10 Yr, 20 Yr, 30 Yr.

Daily data can be found here:

https://www.treasury.gov/resource-center/data-chart-center/interest-rates/Pages/TextView.aspx?data=yieldAll

Data is stored in a MATLAB® data file as a MATLAB `dataset` object.

```
load Data_USYieldCurve

% Extract data for the last day of each month
MonthYearMat = repmat((1990:2010)',1,12)';
EOMDates = lbusdate(MonthYearMat(:),repmat((1:12)',21,1));
MonthlyIndex = find(ismember(Dataset.Properties.ObsNames,datestr(EOMDates)));
Estimationdataset = Dataset(MonthlyIndex,:);
EstimationData = double(Estimationdataset);
```

Diebold Li Model

Diebold and Li start with the Nelson Siegel model

$$y = \beta_0 + (\beta_1 + \beta_2)\frac{\tau}{m}(1 - e^{\frac{-m}{\tau}}) - \beta_2 e^{\frac{-m}{\tau}}$$

and rewrite it to be the following:

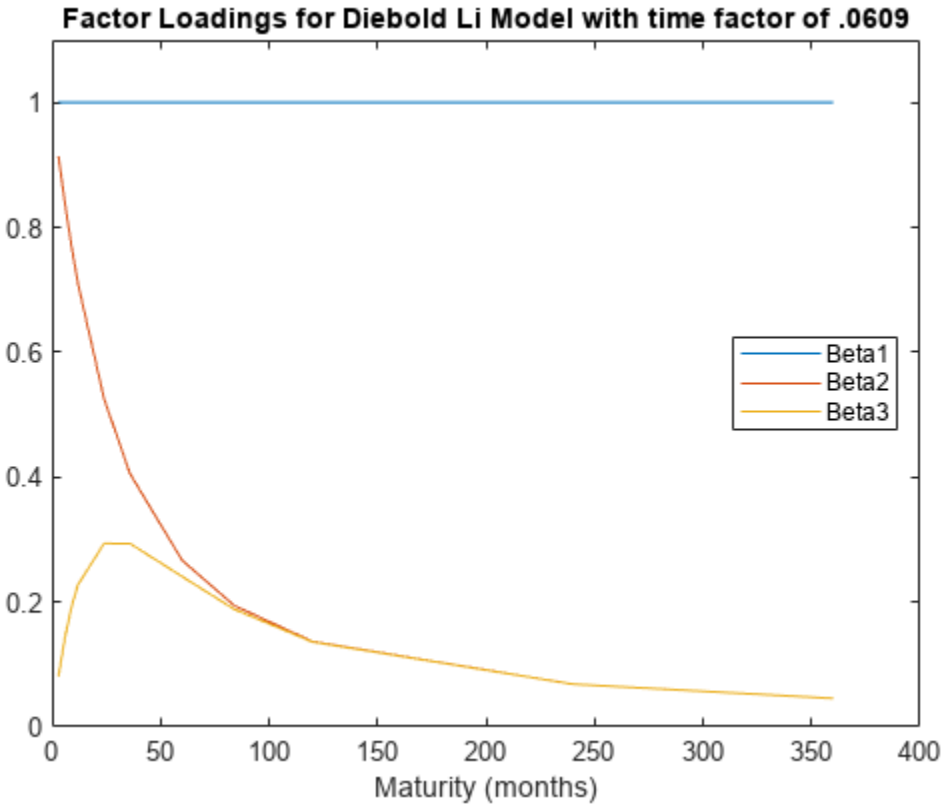
$$y_t(\tau) = \beta_{1t} + \beta_{2t}\left(\frac{1 - e^{-\lambda_t\tau}}{\lambda_t\tau}\right) + \beta_{3t}\left(\frac{1 - e^{-\lambda_t\tau}}{\lambda_t\tau} - e^{-\lambda_t\tau}\right)$$

The above model allows the factors to be interpreted in the following way: Beta1 corresponds to the long term/level of the yield curve, Beta2 corresponds to the short term/slope, and Beta3 corresponds to the medium term/curvature. λ determines the maturity at which the loading on the curvature is maximized, and governs the exponential decay rate of the model.

Diebold and Li advocate setting λ to maximize the loading on the medium term factor, Beta3, at 30 months. This also transforms the problem from a nonlinear fitting to a simple linear regression.

```
% Explicitly set the time factor lambda
lambda_t = .0609;
```

```
% Construct a matrix of the factor loadings
% Tenors associated with data
TimeToMat = [3 6 9 12 24 36 60 84 120 240 360]';
X = [ones(size(TimeToMat)) (1 - exp(-lambda_t*TimeToMat))./(lambda_t*TimeToMat) ...
    ((1 - exp(-lambda_t*TimeToMat))./(lambda_t*TimeToMat) - exp(-lambda_t*TimeToMat))];

% Plot the factor loadings
plot(TimeToMat,X)
title('Factor Loadings for Diebold Li Model with time factor of .0609')
xlabel('Maturity (months)')
ylim([0 1.1])
legend({'Beta1','Beta2','Beta3'},'location','east')
```



Fit the Model

A `DieboldLi` object is developed to facilitate fitting the model from yield data. The `DieboldLi` object inherits from the `IRCurve` object, so the `getZeroRates`, `getDiscountFactors`, `getParYields`, `getForwardRates`, and `toRateSpec` methods are all implemented. Additionally, the method `fitYieldsFromBetas` is implemented to estimate the Beta parameters given a lambda parameter for observed market yields.

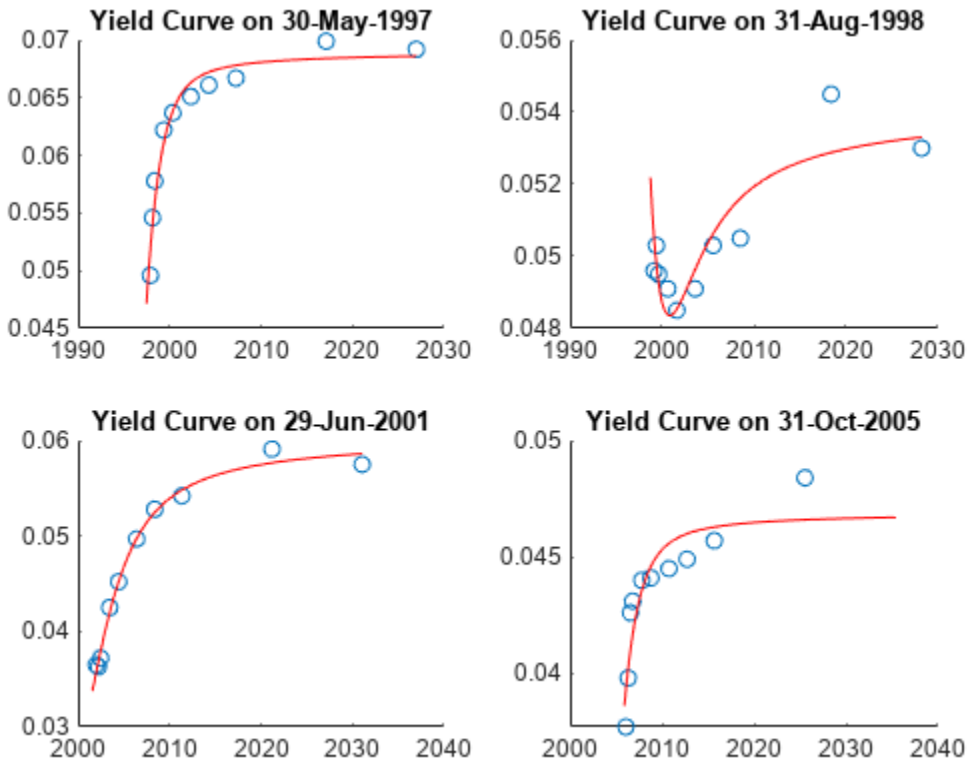
The `DieboldLi` object is used to fit a Diebold Li model for each month from 1990 through 2010.

```
% Preallocate the Betas
Beta = zeros(size(EstimationData,1),3);

% Loop through and fit each end of month yield curve
```

```
for jdx = 1:size(EstimationData,1)
    tmpCurveModel = DieboldLi.fitBetasFromYields(EOMDates(jdx),lambda_t*12,daysadd(EOMDates(jdx)
    Beta(jdx,:) = [tmpCurveModel.Beta1 tmpCurveModel.Beta2 tmpCurveModel.Beta3];
end
```

The Diebold Li fits on selected dates are included here

```
PlotSettles = [datetime(1997,5,30) , datetime(1998,8,31) , datetime(2001,6,29) , datetime(2005,10
figure
for jdx = 1:length(PlotSettles)
    subplot(2,2,jdx)
    tmpIdx = find(strcmpi(Estimationdataset.Properties.ObsNames,datestr(PlotSettles(jdx))));
    tmpCurveModel = DieboldLi.fitBetasFromYields(PlotSettles(jdx),lambda_t*12,...
        daysadd(PlotSettles(jdx),30*TimeToMat),EstimationData(tmpIdx,:)');
    scatter(daysadd(PlotSettles(jdx),30*TimeToMat),EstimationData(tmpIdx,:))
    hold on
    PlottingDates = (PlotSettles(jdx)+30:30:PlotSettles(jdx)+30*360)';
    plot(PlottingDates,tmpCurveModel.getParYields(PlottingDates),'r-')
    title(['Yield Curve on ' datestr(PlotSettles(jdx))])
    datetick
end
```



Forecasting

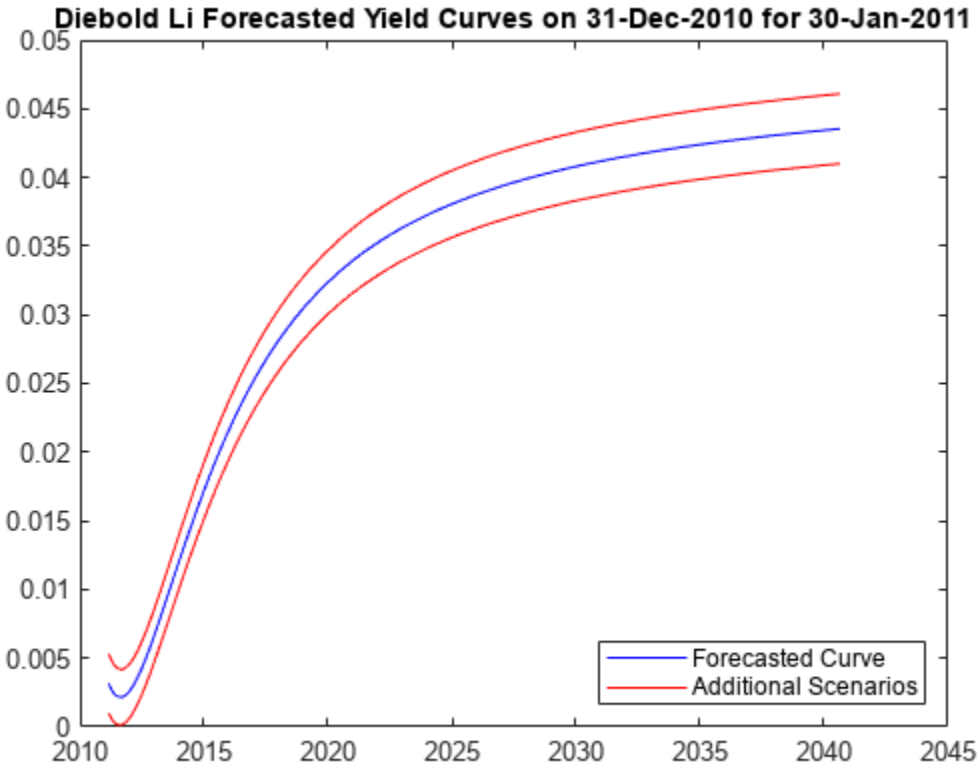
The Diebold Li model can be used to forecast future yield curves. Diebold and Li propose fitting an AR(1) model to the time series of each Beta parameter. This fitted model can then be used to forecast future values of each parameter, and by extension, future yield curves.

For this example the MATLAB function `regress` is used to estimate the parameters for an AR(1) model for each Beta.

The confidence intervals for the regression fit are also used to generate two additional yield curve forecasts that serve as additional possible scenarios for the yield curve.

The `MonthsLag` variable can be adjusted to make different period ahead forecasts. For example, changing the value from `1` to `6` would change the forecast from a 1 month ahead to 6 month ahead forecast.

```
MonthsLag = 1;

[tmpBeta,bint] = regress(Beta(MonthsLag+1:end,1),[ones(size(Beta(MonthsLag+1:end,1))) Beta(1:end
ForecastBeta(1,1) = [1 Beta(end,1)]*tmpBeta;
ForecastBeta_Down(1,1) = [1 Beta(end,1)]*bint(:,1);
ForecastBeta_Up(1,1) = [1 Beta(end,1)]*bint(:,2);
[tmpBeta,bint]  = regress(Beta(MonthsLag+1:end,2),[ones(size(Beta(MonthsLag+1:end,2))) Beta(1:en
ForecastBeta(1,2) = [1 Beta(end,2)]*tmpBeta;
ForecastBeta_Down(1,2) = [1 Beta(end,2)]*bint(:,1);
ForecastBeta_Up(1,2) = [1 Beta(end,2)]*bint(:,2);
[tmpBeta,bint]  = regress(Beta(MonthsLag+1:end,3),[ones(size(Beta(MonthsLag+1:end,3))) Beta(1:en
ForecastBeta(1,3) = [1 Beta(end,3)]*tmpBeta;
ForecastBeta_Down(1,3) = [1 Beta(end,3)]*bint(:,1);
ForecastBeta_Up(1,3) = [1 Beta(end,3)]*bint(:,2);

% Forecasted yield curve
figure
Settle = daysadd(EOMDates(end),30*MonthsLag);
DieboldLi_Forecast = DieboldLi('ParYield',Settle,[ForecastBeta lambda_t*12]);
DieboldLi_Forecast_Up = DieboldLi('ParYield',Settle,[ForecastBeta_Up lambda_t*12]);
DieboldLi_Forecast_Down = DieboldLi('ParYield',Settle,[ForecastBeta_Down lambda_t*12]);
PlottingDates = (Settle+30:30:Settle+30*360)';
plot(PlottingDates,DieboldLi_Forecast.getParYields(PlottingDates),'b-')
hold on
plot(PlottingDates,DieboldLi_Forecast_Up.getParYields(PlottingDates),'r-')
plot(PlottingDates,DieboldLi_Forecast_Down.getParYields(PlottingDates),'r-')
title(['Diebold Li Forecasted Yield Curves on ' datestr(EOMDates(end)) ' for '  datestr(Settle)]
legend({'Forecasted Curve','Additional Scenarios'},'location','southeast')
datetick
```



Bibliography

This example is based on the following paper:

[1] Francis X. Diebold, Canlin Li. "Forecasting the Term Structure of Government Bond Yields." *Journal of Econometrics*, Volume 130, Issue 2, February 2006, pp. 337-364.

See Also

More About

- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Calibrating Caplets Using the Normal (Bachelier) Model

This example shows how to use `hwcalbycap` to calibrate market data with the Normal (Bachelier) model to price caplets. Use the Normal (Bachelier) model to perform calibrations when working with negative interest rates, strikes, and normal implied volatilities.

Consider a cap with these parameters:

```
Settle = datetime(2016,12,30);
Maturity = datetime(2019,12,30);
Strike = -0.001075;
Reset = 2;
Principal = 100;
Basis = 0;
```

The caplets and market data for this example are defined as:

```
capletDates = cfdates(Settle, Maturity, Reset, Basis);
datestr(capletDates')
```

```
ans = 6x11 char array
    '30-Jun-2017'
    '30-Dec-2017'
    '30-Jun-2018'
    '30-Dec-2018'
    '30-Jun-2019'
    '30-Dec-2019'
```

```
% Market data information
MarketStrike = [-0.0013; 0];
MarketMat =  [datetime(2017,6,30) ; datetime(2017,12,30) ; datetime(2018,6,30) ; datetime(2018,12
MarketVol = [0.184 0.2329 0.2398 0.2467 0.2906 0.3348;   % First row in table corresponding to St
             0.217 0.2707 0.2760 0.2814 0.3160 0.3508];  % Second row in table corresponding to S
```

Define the `RateSpec` using `intenvset`.

```
Rates= [-0.002210;-0.002020;-0.00182;-0.001343;-0.001075];
ValuationDate = datetime(2016,12,30);
EndDates =  [datetime(2017,6,30) ; datetime(2017,12,30) ; datetime(2018,6,30) ; datetime(2018,12
Compounding = 2;
Basis = 0;

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

Use `hwcalbycap` to find values for the volatility parameters `Alpha` and `Sigma` using the Normal (Bachelier) model.

```
format short
o=optimoptions('lsqnonlin','TolFun',100*eps);
warning ('off','fininst:hwcalbycapfloor:NoConverge')
[Alpha, Sigma, OptimOut] = hwcalbycap(RateSpec, MarketStrike, MarketMat,...
MarketVol, Strike, Settle, Maturity, 'Reset', Reset, 'Principal', Principal,...
'Basis', Basis, 'OptimOptions', o, 'model', 'normal')
```

```
Local minimum possible.
lsqnonlin stopped because the size of the current step is less than
the value of the step size tolerance.

Alpha = 1.0000e-06

Sigma = 0.3384

OptimOut = struct with fields:
     resnorm: 1.5181e-04
    residual: [5x1 double]
    exitflag: 2
      output: [1x1 struct]
      lambda: [1x1 struct]
    jacobian: [5x2 double]
```

The `OptimOut.residual` field of the `OptimOut` structure is the optimization residual. This value contains the difference between the Normal (Bachelier) caplets and those calculated during the optimization. Use the `OptimOut.residual` value to calculate the percentual difference (error) compared to Normal (Bachelier) caplet prices, and then decide whether the residual is acceptable. There is almost always some residual, so decide if it is acceptable to parameterize the market with a single value of `Alpha` and `Sigma`.

Price the caplets using the market data and Normal (Bachelier) model to obtain the reference caplet values. To determine the effectiveness of the optimization, calculate reference caplet values using the Normal (Bachelier) formula and the market data. Note, you must first interpolate the market data to obtain the caplets for calculation.

```
MarketMatNum = datenum(MarketMat);
[Mats, Strikes] = meshgrid(MarketMatNum, MarketStrike);
FlatVol = interp2(Mats, Strikes, MarketVol, datenum(Maturity), Strike, 'spline');

[CapPrice, Caplets] = capbynormal(RateSpec, Strike, Settle, Maturity, FlatVol,...
'Reset', Reset, 'Basis', Basis, 'Principal', Principal);
Caplets = Caplets(2:end)'
```

```
Caplets = 5×1

    4.7392
    6.7799
    8.2609
    9.6136
   10.6455
```

Compare the optimized values and Normal (Bachelier) values, and display the results graphically. After calculating the reference values for the caplets, compare the values analytically and graphically to determine whether the calculated single values of `Alpha` and `Sigma` provide an adequate approximation.

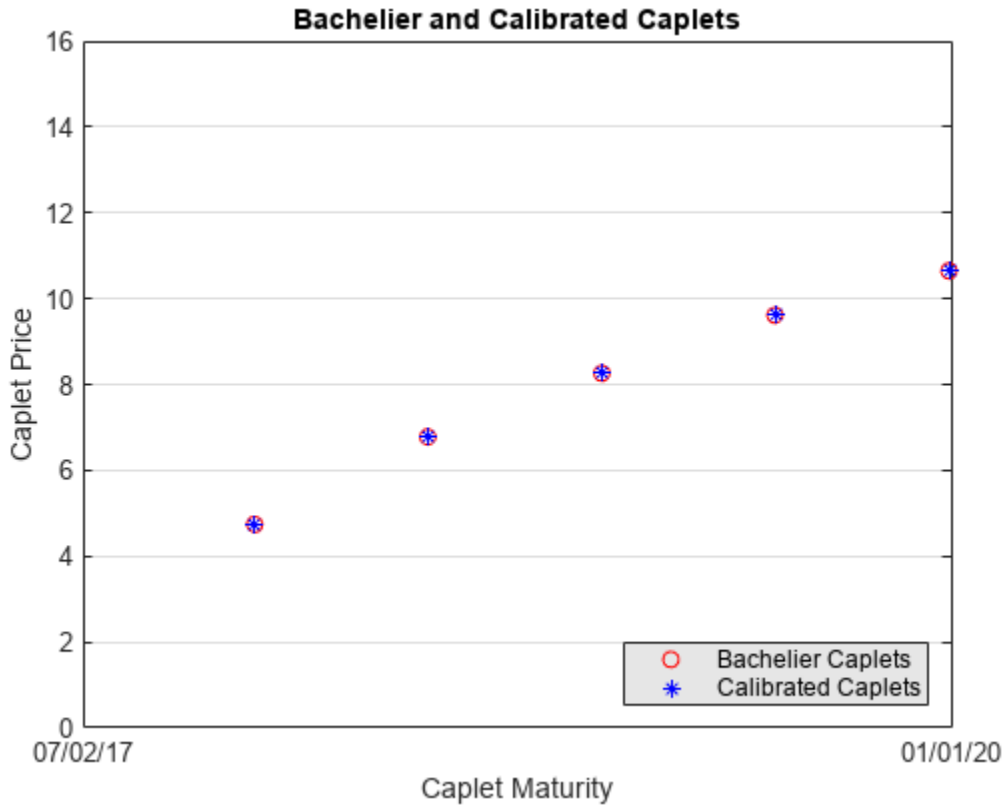
```
OptimCaplets = Caplets+OptimOut.residual;

disp('   ');


disp(' Bachelier   Calibrated Caplets');
```

```
 Bachelier   Calibrated Caplets
```

```
disp([Caplets        OptimCaplets])
```

```
    4.7392    4.7453
    6.7799    6.7851
    8.2609    8.2657
    9.6136    9.6112
   10.6455   10.6379
```

```
plot(MarketMatNum(2:end), Caplets, 'or', MarketMatNum(2:end), OptimCaplets, '*b');
datetick('x', 2)
xlabel('Caplet Maturity');
ylabel('Caplet Price');
ylim ([0 16]);
title('Bachelier and Calibrated Caplets');
h = legend('Bachelier Caplets', 'Calibrated Caplets');
set(h, 'color', [0.9 0.9 0.9]);
set(h, 'Location', 'SouthEast');
set(gcf, 'NumberTitle', 'off')
grid on
```



See Also

More About

- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Calibrating Floorlets Using the Normal (Bachelier) Model

This example shows how to use `hwcalbyfloor` to calibrate market data with the Normal (Bachelier) model to price floorlets. Use the Normal (Bachelier) model to perform calibrations when working with negative interest rates, strikes, and normal implied volatilities.

Consider a floor with these parameters:

```
Settle = datetime(2016,12,30);
Maturity = datetime(2019,12,30);
Strike = -0.004075;
Reset = 2;
Principal = 100;
Basis = 0;
```

The floorlets and market data for this example are defined as:

```
floorletDates = cfdates(Settle, Maturity, Reset, Basis);
datestr(floorletDates')
```

```
ans = 6x11 char array
    '30-Jun-2017'
    '30-Dec-2017'
    '30-Jun-2018'
    '30-Dec-2018'
    '30-Jun-2019'
    '30-Dec-2019'
```

```
% Market data information
MarketStrike = [-0.00595; 0];
MarketMat = [datetime(2017,6,30) ; datetime(2017,12,30) ; datetime(2018,6,30) ; datetime(2018,12
MarketVol = [0.184 0.2329 0.2398 0.2467 0.2906 0.3348;   % First row in table corresponding to St
             0.217 0.2707 0.2760 0.2814 0.3160 0.3508];  % Second row in table corresponding to S
```

Define the `RateSpec` using `intenvset`.

```
Rates= [-0.003210;-0.003020;-0.00182;-0.001343;-0.001075];
ValuationDate = datetime(2016,12,30);
EndDates = [datetime(2017,6,30) ; datetime(2017,12,30) ; datetime(2018,6,30) ; datetime(2018,12,3
Compounding = 2;
Basis = 0;

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

Use `hwcalbyfloor` to find values for the volatility parameters `Alpha` and `Sigma` using the Normal (Bachelier) model.

```
format short
o=optimoptions('lsqnonlin','TolFun',100*eps);
warning ('off','fininst:hwcalbycapfloor:NoConverge')
[Alpha, Sigma, OptimOut] = hwcalbyfloor(RateSpec, MarketStrike, MarketMat,...
MarketVol, Strike, Settle, Maturity, 'Reset', Reset, 'Principal', Principal,...
'Basis', Basis, 'OptimOptions', o, 'model', 'normal')
```

```
Local minimum possible.
lsqnonlin stopped because the size of the current step is less than
the value of the step size tolerance.

Alpha = 1.0000e-06

Sigma = 0.3410

OptimOut = struct with fields:
     resnorm: 1.9233e-04
    residual: [5x1 double]
    exitflag: 2
      output: [1x1 struct]
      lambda: [1x1 struct]
    jacobian: [5x2 double]
```

The `OptimOut.residual` field of the `OptimOut` structure is the optimization residual. This value contains the difference between the Normal (Bachelier) floorlets and those calculated during the optimization. Use the `OptimOut.residual` value to calculate the percentual difference (error) compared to Normal (Bachelier) floorlet prices, and then decide whether the residual is acceptable. There is almost always some residual, so decide if it is acceptable to parameterize the market with a single value of `Alpha` and `Sigma`.

Price the floorlets using the market data and Normal (Bachelier) model to obtain the reference floorlet values. To determine the effectiveness of the optimization, calculate reference floorlet values using the Normal (Bachelier) formula and the market data. Note, you must first interpolate the market data to obtain the floorlets for calculation.

```
MarketMatNum = datenum(MarketMat);
[Mats, Strikes] = meshgrid(MarketMatNum, MarketStrike);
FlatVol = interp2(Mats, Strikes, MarketVol, datenum(Maturity), Strike, 'spline');

[FloorPrice, Floorlets] = floorbynormal(RateSpec, Strike, Settle, Maturity, FlatVol,...
'Reset', Reset, 'Basis', Basis, 'Principal', Principal);
Floorlets = Floorlets(2:end)'
```

```
Floorlets = 5×1

    4.7637
    6.7180
    8.1833
    9.5825
   10.6090
```

Compare the optimized values and Normal (Bachelier) values, and display the results graphically. After calculating the reference values for the floorlets, compare the values analytically and graphically to determine whether the calculated single values of `Alpha` and `Sigma` provide an adequate approximation.

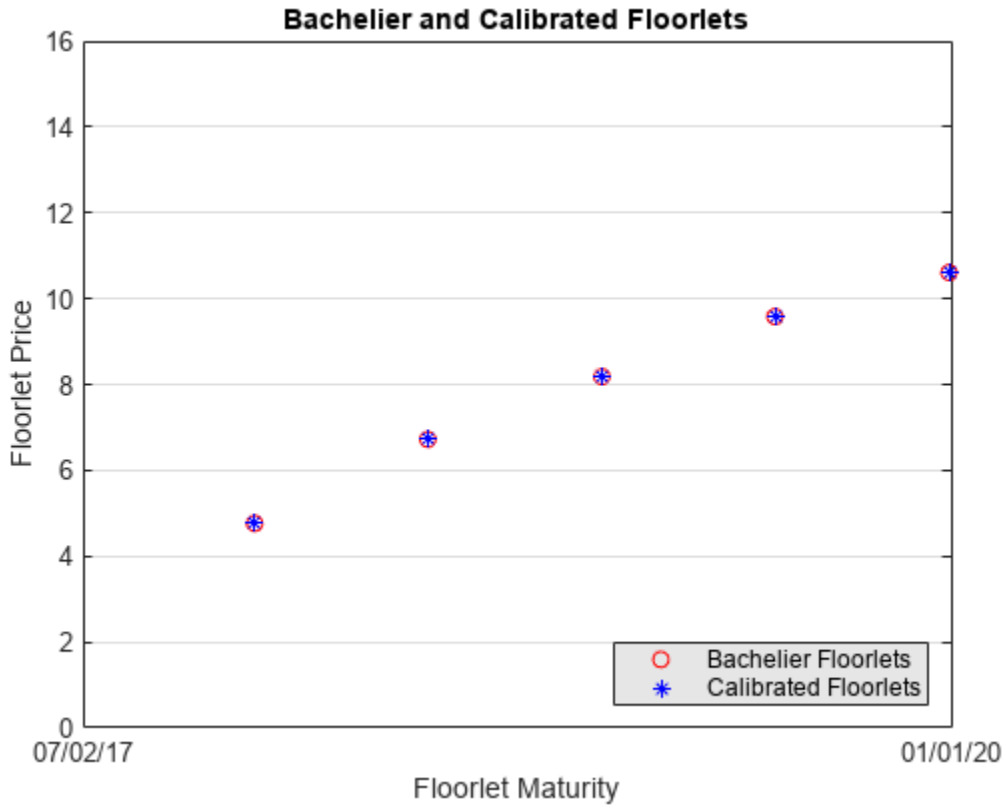
```
OptimFloorlets = Floorlets+OptimOut.residual;

disp('   ');


disp(' Bachelier   Calibrated Floorlets');
```

```
 Bachelier   Calibrated Floorlets
```

```
disp([Floorlets        OptimFloorlets])
```

```
    4.7637    4.7685
    6.7180    6.7263
    8.1833    8.1878
    9.5825    9.5795
   10.6090   10.6007
```

```
plot(MarketMatNum(2:end), Floorlets, 'or', MarketMatNum(2:end), OptimFloorlets, '*b');
datetick('x', 2)
xlabel('Floorlet Maturity');
ylabel('Floorlet Price');
ylim ([0 16]);
title('Bachelier and Calibrated Floorlets');
h = legend('Bachelier Floorlets', 'Calibrated Floorlets');
set(h, 'color', [0.9 0.9 0.9]);
set(h, 'Location', 'SouthEast');
set(gcf, 'NumberTitle', 'off')
grid on
```



See Also

More About

- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Calibrate the SABR Model Using Normal (Bachelier) Volatilities with Negative Strikes

This example shows how to use two different methods to calibrate the SABR stochastic volatility model from market implied Normal (Bachelier) volatilities with negative strikes. Both approaches use `normalvolbysabr`, which computes the implied Normal volatilities by using the SABR model. When the `Beta` parameter of the SABR model is set to zero, the model is a Normal SABR model, which allows computing the implied Normal volatilities for negative strikes.

Load the Market Implied Normal (Bachelier) Volatility Data

Set up hypothetical market implied Normal volatilities for European swaptions over a range of strikes before calibration. The swaptions expire in one year from the `Settle` date and have two-year swaps as the underlying instrument. The rates are expressed in decimals. The market implied Normal volatilities are converted from basis points to decimals. (Changing the units affects the numerical value and interpretation of the `Alpha` parameter input to the function `normalvolbysabr`.)

```
% Load the market implied Normal volatility data for swaptions expiring in one year.
Settle = '20-Sep-2017';
ExerciseDate = '20-Sep-2018';
Basis = 1;

ATMStrike = -0.174/100;
MarketStrikes = ATMStrike + ((-0.5:0.25:1.5)')./100;
MarketVolatilities = [20.58 17.64 16.93 18.01 20.46 22.90 26.11 28.89 31.91]'/10000;

% At the time of Settle, define the underlying forward rate and the at-the-money volatility.
CurrentForwardValue = MarketStrikes(3)
```

```
CurrentForwardValue = -0.0017
```

```
ATMVolatility = MarketVolatilities(3)
```

```
ATMVolatility = 0.0017
```

Method 1: Calibrate `Alpha`, `Rho`, and `Nu` Directly

This section demonstrates how to calibrate the `Alpha`, `Rho`, and `Nu` parameters directly. The value of the `Beta` parameter is set to zero in order to allow negative rates in the SABR model (Normal SABR). After fixing the value of β(`Beta`), the parameters α (`Alpha`), ρ (`Rho`), and ν (`Nu`) are all fitted directly. The Optimization Toolbox™ function `lsqnonlin` generates the parameter values that minimize the squared error between the market volatilities and the volatilities computed by `normalvolbysabr`.

```
% Define the predetermined Beta
Beta1 = 0; % Setting Beta to zero allows negative rates for Normal volatilities

% Calibrate Alpha, Rho, and Nu
objFun = @(X) MarketVolatilities - ...
    normalvolbysabr(X(1), Beta1, X(2), X(3), Settle, ...
    ExerciseDate, CurrentForwardValue, MarketStrikes, 'Basis', Basis);

% If necessary, tolerances and stopping criteria can be adjusted for lsqnonlin
X = lsqnonlin(objFun, [ATMVolatility 0 0.5], [0 -1 0], [Inf 1 Inf]);
```

```
Local minimum found.
```

```
Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.
```

```
Alpha1 = X(1);
Rho1 = X(2);
Nu1 = X(3);
```

Method 2: Calibrate Rho and Nu by Implying Alpha from At-The-Money Volatility

This section demonstrates how to use an alternative calibration method where the value of β is again predetermined to be zero in order to allow negative rates. However, after fixing the value of β (`Beta`), the parameters ρ (`Rho`), and ν (`Nu`) are fitted directly while α (`Alpha`) is implied from the market at-the-money volatility. Models calibrated using this method produce at-the-money volatilities that are equal to market quotes. This approach can be useful when at-the-money volatilities are quoted most frequently and are important to match. In order to imply α (`Alpha`) from market at-the-money Normal volatility ($\sigma_{Normal,\,ATM}$), the following cubic polynomial is solved for α (`Alpha`), and the smallest positive real root is selected. This is similar to the approach used for implying α (`Alpha`) from market at-the-money Black volatility [2]. However, note that the following expression that is used for Normal volatilities is different from another expression that is used for Black volatilities.

$$\frac{\beta(\beta-2)T}{24F^{(2-2\beta)}}\alpha^3+\frac{\rho\beta\nu T}{4F^{(1-\beta)}}\alpha^2+\left(1+\frac{2-3\rho^2}{24}\nu^2T\right)\alpha-\sigma_{Normal,\,ATM}F^{-\beta}=0$$

```
% Define the predetermined Beta
Beta2 = 0; % Setting Beta to zero allows negative rates for Normal volatilities

% Year fraction from Settle to option maturity
T = yearfrac(Settle, ExerciseDate, Basis);

% This function solves the SABR at-the-money volatility equation as a
% polynomial of Alpha
alpharootsNormal = @(Rho,Nu) roots([...
    Beta2.*(Beta2 - 2)*T/24/CurrentForwardValue^(2 - 2*Beta2) ...
    Rho*Beta2*Nu*T/4/CurrentForwardValue^(1 - Beta2) ...
    (1 + (2 - 3*Rho^2)*Nu^2*T/24) ...
    -ATMVolatility*CurrentForwardValue^(-Beta2)]);

% This function converts at-the-money volatility into Alpha by picking the
% smallest positive real root
atmNormalVol2SabrAlpha = @(Rho,Nu) min(real(arrayfun(@(x) ...
    x*(x>0) + realmax*(x<0 || abs(imag(x))>1e-6), alpharootsNormal(Rho,Nu))));

% Calibrate Rho and Nu (while converting at-the-money volatility into Alpha
% using atmVol2NormalSabrAlpha)
objFun = @(X) MarketVolatilities - ...
    normalvolbysabr(atmNormalVol2SabrAlpha(X(1), X(2)), ...
    Beta2, X(1), X(2), Settle, ExerciseDate, CurrentForwardValue, ...
    MarketStrikes, 'Basis', Basis);

% If necessary, tolerances and stopping criteria can be adjusted for lsqnonlin
X = lsqnonlin(objFun, [0 0.5], [-1 0], [1 Inf]);
```

```
Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.
```

```
Rho2 = X(1);
Nu2 = X(2);

% Obtain final Alpha from at-the-money volatility using calibrated parameters
Alpha2 = atmNormalVol2SabrAlpha(Rho2, Nu2);

% Display calibrated parameters
C = {Alpha1 Beta1 Rho1 Nu1;Alpha2 Beta2 Rho2 Nu2};
format;
CalibratedPrameters = cell2table(C,...
    'VariableNames',{'Alpha' 'Beta' 'Rho' 'Nu'},...
    'RowNames',{'Method 1';'Method 2'})
```

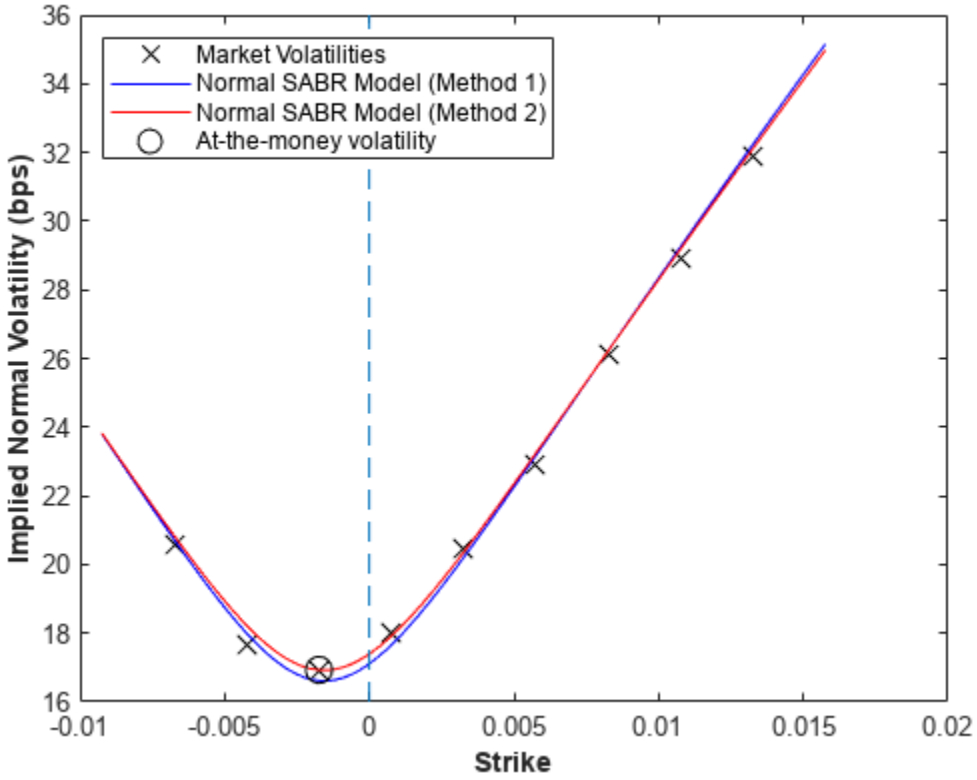
```
CalibratedPrameters=2×4 table
                Alpha      Beta       Rho         Nu
              _________    ____    _________    _______

    Method 1    0.0016332     0      -0.034233    0.45877
    Method 2    0.0016652     0        -0.0318    0.44812
```

Use the Calibrated Models

Use the calibrated models to compute new volatilities at any strike value, including negative strikes.

Compute volatilities for models calibrated using Method 1 and Method 2, then plot the results. The model calibrated using Method 2 reproduces the market at-the-money volatility (marked with a circle) exactly.

```
PlottingStrikes = (min(MarketStrikes)-0.0025:0.0001:max(MarketStrikes)+0.0025)';

% Compute volatilities for model calibrated by Method 1
ComputedVols1 = normalvolbysabr(Alpha1, Beta1, Rho1, Nu1, Settle, ...
    ExerciseDate, CurrentForwardValue, PlottingStrikes, 'Basis', Basis);

% Compute volatilities for model calibrated by Method 2
ComputedVols2 = normalvolbysabr(Alpha2, Beta2, Rho2, Nu2, Settle, ...
    ExerciseDate, CurrentForwardValue, PlottingStrikes, 'Basis', Basis);

figure;
plot(MarketStrikes,MarketVolatilities*10000,'xk',...
    PlottingStrikes,ComputedVols1*10000,'b', ...
    PlottingStrikes,ComputedVols2*10000,'r', ...
    CurrentForwardValue,ATMVolatility*10000,'ok',...
    'MarkerSize',10);

h = gca;
line([0,0],[min(h.YLim),max(h.YLim)],'LineStyle','--');

xlabel('Strike', 'FontWeight', 'bold');
ylabel('Implied Normal Volatility (bps)', 'FontWeight', 'bold');
legend('Market Volatilities', 'Normal SABR Model (Method 1)', ...
    'Normal SABR Model (Method 2)', 'At-the-money volatility', ...
    'Location', 'northwest');
```



References

[1] Hagan, P. S., Kumar, D., Lesniewski, A. S. and Woodward, D. E. "Managing smile risk." *Wilmott Magazine*, 2002.

[2] West, G. "Calibration of the SABR Model in Illiquid Markets." *Applied Mathematical Finance*, 12(4), pp. 371–385, 2004.

See Also

More About

- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Calibrate Shifted SABR Model Parameters for Swaption Instrument

Calibrate model parameters for a `Swaption` instrument when you use a `SABR` pricing method.

Load Market Data

```
% Zero curve
ValuationDate = datetime("5-Mar-2016", 'Locale', 'en_US');
ZeroDates = datemnth(ValuationDate,[1 2 3 6 9 12*[1 2 3 4 5 6 7 8 9 10 12]])';
ZeroRates = [-0.33 -0.28 -0.24 -0.12 -0.08 -0.03 0.015 0.028 ...
    0.033 0.042 0.056 0.095 0.194 0.299 0.415 0.525]'/100;
Compounding = 1;
ZeroCurve = ratecurve("zero",ValuationDate,ZeroDates,ZeroRates,'Compounding',Compounding)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: 1
               Basis: 0
               Dates: [16x1 datetime]
               Rates: [16x1 double]
              Settle: 05-Mar-2016
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

```
% Define the swaptions
SwaptionSettle = datetime("5-Mar-2016", 'Locale', 'en_US');
SwaptionExerciseDate = datetime("5-Mar-2017", 'Locale', 'en_US');
SwaptionStrikes = (-0.6:0.01:1.6)'/100; % Include negative strikes
SwapMaturity = datetime("5-Mar-2022", 'Locale', 'en_US'); % Maturity of underlying swap
OptSpec = 'call';
```

Compute Forward Swap Rate by Creating Swap Instrument

Use `fininstrument` to create a `Swap` instrument object.

```
LegRate = [0 0];
Swap = fininstrument("Swap", 'Maturity', SwapMaturity, 'LegRate', LegRate, "LegType",["fixed" "f
    "ProjectionCurve", ZeroCurve, "StartDate", SwaptionExerciseDate)
```

```
Swap =
  Swap with properties:

                   LegRate: [0 0]
                   LegType: ["fixed"    "float"]
                     Reset: [2 2]
                     Basis: [0 0]
                  Notional: 100
         LatestFloatingRate: [NaN NaN]
                ResetOffset: [0 0]
    DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [1x2 ratecurve]
```

```
       BusinessDayConvention: ["actual"    "actual"]
                    Holidays: NaT
                EndMonthRule: [1 1]
                   StartDate: 05-Mar-2017
                    Maturity: 05-Mar-2022
                        Name: ""
```

```
ForwardValue = parswaprate(Swap,ZeroCurve)
```

```
ForwardValue = 7.3271e-04
```

Load the Market Implied Volatility Data

The market swaption volatilities are quoted in terms of shifted Black volatilities with a `0.8` percent shift.

```
StrikeGrid = [-0.5; -0.25; -0.125; 0; 0.125; 0.25; 0.5; 1.0; 1.5]/100;
MarketStrikes = ForwardValue + StrikeGrid;
Shift = 0.008;  % 0.8 percent shift
MarketShiftedBlackVolatilities = [21.1; 15.3; 14.0; 14.6; 16.0; 17.7; 19.8; 23.9; 26.2]/100;
ATMShiftedBlackVolatility = MarketShiftedBlackVolatilities(StrikeGrid==0);
```

Calibrate Shifted SABR Model Parameters

The `Beta` parameter is predetermined at `0.5`. Use `volatilities` to compute the implied volatility.

```
Beta = 0.5;

% Calibrate Alpha, Rho, and Nu
objFun = @(X) MarketShiftedBlackVolatilities - volatilities(finpricer("Analytic", 'Model', ...
    finmodel("SABR", 'Alpha', X(1), 'Beta', Beta, 'Rho', X(2), 'Nu', X(3), 'Shift', Shift), ...
    'DiscountCurve', ZeroCurve), SwaptionExerciseDate, ForwardValue, MarketStrikes);

X = lsqnonlin(objFun, [0.5 0 0.5], [0 -1 0], [Inf 1 Inf]);
```

```
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.
```

```
Alpha = X(1);
Rho = X(2);
Nu = X(3);
```

Create SABR Model Using the Calibrated Parameters

Use `finmodel` to create a SABR model object.

```
SABRModel = finmodel("SABR",'Alpha',Alpha,'Beta',Beta,'Rho',Rho,'Nu',Nu,'Shift',Shift)
```

```
SABRModel =
  SABR with properties:

             Alpha: 0.0135
              Beta: 0.5000
               Rho: 0.4654
                Nu: 0.4957
```

```
             Shift: 0.0080
    VolatilityType: "black"
```

Create SABR Pricer Using Calibrated SABR Model and Compute Volatilities

Use `finpricer` to create a SABR pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

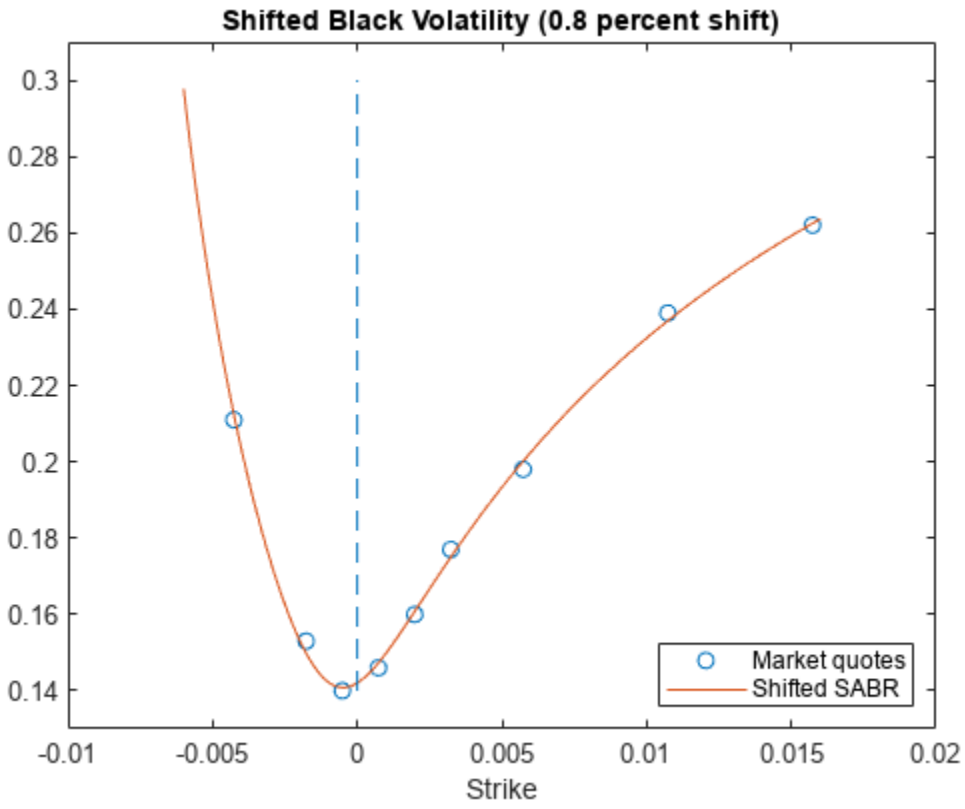
```
SABRPricer = finpricer("Analytic", 'Model', SABRModel, 'DiscountCurve', ZeroCurve)
```

```
SABRPricer =
  SABR with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.SABR]
```

```
SABRShiftedBlackVolatilities = volatilities(SABRPricer, SwaptionExerciseDate, ForwardValue, Swapt
```

```
SABRShiftedBlackVolatilities = 221×1

    0.2978
    0.2911
    0.2848
    0.2787
    0.2729
    0.2673
    0.2620
    0.2568
    0.2518
    0.2470
      ⋮
```

```
figure;
plot(MarketStrikes, MarketShiftedBlackVolatilities, 'o', ...
    SwaptionStrikes, SABRShiftedBlackVolatilities);
h = gca;
line([0,0],[min(h.YLim),max(h.YLim)],'LineStyle','--');
ylim([0.13 0.31])
xlabel('Strike');
legend('Market quotes','Shifted SABR', 'location', 'southeast');
title (['Shifted Black Volatility (',num2str(Shift*100),' percent shift)']);
```



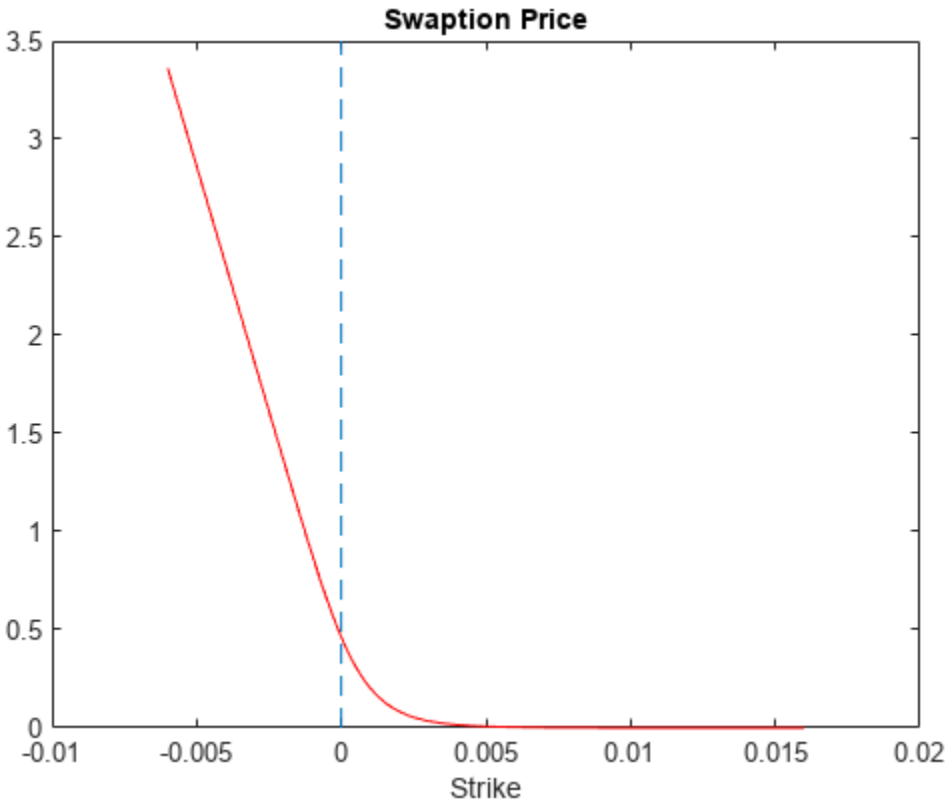
Price Swaption Instruments Using Calibrated SABR Model and SABR Pricer

```
% Create swaption instruments
NumInst = length(SwaptionStrikes);
Swaptions(NumInst, 1) = fininstrument("Swaption", ...
    'Strike', SwaptionStrikes(1), 'ExerciseDate', SwaptionExerciseDate(1), 'Swap', Swap);
for k = 1:NumInst
    Swaptions(k) = fininstrument("Swaption", 'Strike', SwaptionStrikes(k), ...
        'ExerciseDate', SwaptionExerciseDate, 'Swap', Swap, 'OptionType', OptSpec);
end
Swaptions
```

```
Swaptions=221×1 object
  16x1 Swaption array with properties:

    OptionType
    ExerciseStyle
    ExerciseDate
    Strike
    Swap
    Name
      ⋮
```

```
% Price swaptions using the SABR pricer
SwaptionPrices = price(SABRPricer,Swaptions);

figure;
```

```
plot(SwaptionStrikes, SwaptionPrices, 'r');
h = gca;
line([0,0],[min(h.YLim),max(h.YLim)],'LineStyle','--');
xlabel('Strike');
title ('Swaption Price');
```



Price Portfolio of Bond and Bond Option Instruments

This example shows the workflow to create and price a portfolio of bond and bond option instruments. You can use `finportfolio` and `pricePortfolio` to price `FixedBond`, `FixedBondOption`, `OptionEmbeddedFixedBond`, and `FloatBond` instruments using an `IRTree` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018, 1, 1);
 ZeroTimes = calyears(1:4)';
 ZeroRates = [0.035; 0.042147; 0.047345; 0.052707];
 ZeroDates = Settle + ZeroTimes;
 Compounding = 1;
 ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: 1
               Basis: 0
               Dates: [4x1 datetime]
               Rates: [4x1 double]
              Settle: 01-Jan-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create Bond and Option Instruments

Use `fininstrument` to create a `FixedBond`, `FixedBondOption`, `OptionEmbeddedFixedBond`, and `FloatBond` instrument objects.

```
CDates = datetime([2020,1,1 ; 2022,1,1]);
CRates = [.0425; .0750];
CouponRate = timetable(CDates,CRates);
Maturity = datetime(2022,1,1);
Period = 1;

% Vanilla FixedBond
VBond = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',0.0425,'Period',Period,'Name'
```

```
VBond =
  FixedBond with properties:

                 CouponRate: 0.0425
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
```

```
                    Holidays: NaT
                   IssueDate: NaT
              FirstCouponDate: NaT
               LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                        Name: "vanilla_fixed"
```

```
 % Stepped coupon bond
SBond = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',CouponRate,'Period',Period,'Na
```

```
SBond =
  FixedBond with properties:

                CouponRate: [2x1 timetable]
                    Period: 1
                     Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
              FirstCouponDate: NaT
               LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                        Name: "stepped_coupon_bond"
```

```
% FloatBond
Spread = 0;
Reset = 1;
Float = fininstrument("FloatBond",'Maturity',Maturity,'Spread',Spread,'Reset', Reset,...
                      'ProjectionCurve',ZeroCurve,'Name',"floatbond")
```

```
Float =
  FloatBond with properties:

                     Spread: 0
              ProjectionCurve: [1x1 ratecurve]
                  ResetOffset: 0
                        Reset: 1
                        Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
          LatestFloatingRate: NaN
                    Holidays: NaT
                   IssueDate: NaT
              FirstCouponDate: NaT
               LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                        Name: "floatbond"
```

```
% Call option
Strike = 100;
ExerciseDates = datetime(2020,1,1);
OptionType ='call';
Period = 1;
CallOption = fininstrument("FixedBondOption",'Strike',Strike,'ExerciseDate',ExerciseDates,...
                  'OptionType',OptionType,'ExerciseStyle',"american",'Bond', VBond,'Name',"fixed
```

```
CallOption =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2020
           Strike: 100
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option"
```

```
% Option for embedded bond (callable bond)
CDates = datetime([2020,1,1 ; 2022,1,1]);
CRates = [.0425; .0750];
CouponRate = timetable(CDates,CRates);
StrikeOE = [100; 100];
ExerciseDatesOE = [datetime(2020,1,1); datetime(2021,1,1)];
CallSchedule =  timetable(ExerciseDatesOE,StrikeOE,'VariableNames',{'Strike Schedule'});
CallableBond = fininstrument("OptionEmbeddedFixedBond", 'Maturity',Maturity,...
                             'CouponRate',CouponRate,'Period', Period, ...
                             'CallSchedule',CallSchedule,'Name',"option_embedded_fixedbond")
```

```
CallableBond =
  OptionEmbeddedFixedBond with properties:

                 CouponRate: [2x1 timetable]
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                   CallDates: [2x1 datetime]
                    PutDates: [0x1 datetime]
                CallSchedule: [2x1 timetable]
                 PutSchedule: [0x0 timetable]
           CallExerciseStyle: "american"
            PutExerciseStyle: [0x0 string]
                        Name: "option_embedded_fixedbond"
```

Create HullWhite Model

Use `finmodel` to create a `HullWhite` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;

HWModel = finmodel("hullwhite",'alpha',AlphaCurve,'sigma',VolCurve)
```

```
HWModel =
  HullWhite with properties:

    Alpha: 0.1000
    Sigma: 0.0100
```

Create IRTree Pricer for HullWhite Model

Use `finpricer` to create an `IRTree` pricer object for a `HullWhite` model and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [4x1 datetime]
           Model: [1x1 finmodel.HullWhite]
   DiscountCurve: [1x1 ratecurve]
```

Create finportfolio Object and Add Callable Bond Instrument

Create a `finportfolio` object with the vanilla bond, stepped coupon bond, float bond, and the call option.

```
myportfolio = finportfolio([VBond,SBond,Float,CallOption],HWTreePricer, [1,2,2,1])
```

```
myportfolio =
  finportfolio with properties:

    Instruments: [4x1 fininstrument.FinInstrument]
        Pricers: [1x1 finpricer.irtree.HWBKTree]
    PricerIndex: [4x1 double]
       Quantity: [4x1 double]
```

Use `addInstrument` to add the callable bond instrument to the existing portfolio.

```
myportfolio = addInstrument(myportfolio,CallableBond,HWTreePricer,1)
```

```
myportfolio =
  finportfolio with properties:

    Instruments: [5x1 fininstrument.FinInstrument]
        Pricers: [1x1 finpricer.irtree.HWBKTree]
    PricerIndex: [5x1 double]
       Quantity: [5x1 double]
```

```
myportfolio.PricerIndex
```

```
ans = 5×1

     1
     1
     1
     1
     1
```

The `PricerIndex` property has a length equal to the length of instrument objects in the `finportfolio` object and stores the index of which pricer is used for each instrument object. In this case, because there is only one pricer, each instrument must use that pricer.

Price Portfolio

Use `pricePortfolio` to compute the price and sensitivities for the portfolio and the bond and option instruments in the portfolio.

```
format bank
[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(myportfolio)
```

```
PortPrice =
        600.55


InstPrice = 5×1

         96.59
        204.14
        200.00
          0.05
         99.77


PortSens=1×4 table
    Price      Delta       Gamma      Vega
    ______    ________    _______    ______

    600.55    -1297.48    5759.65    -63.40


InstSens=5×4 table
                                Price      Delta      Gamma      Vega
                                ______    _______    _______    ______

    vanilla_fixed                96.59    -344.81    1603.49     -0.00
    stepped_coupon_bond         204.14    -725.96    3364.60      0.00
    floatbond                   200.00       0.00      -0.00     -0.00
    fixed_bond_option             0.05      -3.69      24.15     12.48
    option_embedded_fixedbond    99.77    -223.03     767.41    -75.88
```

Calibrate SABR Model Using Normal (Bachelier) Volatilities with Analytic Pricer

This example shows how to use two different methods to calibrate the SABR stochastic volatility model from market implied Normal (Bachelier) volatilities with negative strikes. Both approaches use the SABR analytic pricer. When the Beta parameter of the SABR model is set to zero, the model is a Normal SABR model, which allows computing the implied Normal volatilities for negative strikes.

Load Market Implied Normal (Bachelier) Volatility Data

Set up hypothetical market implied Normal volatilities for European swaptions over a range of strikes before calibration. The swaptions expire in one year from the Settle date and have two-year swaps as the underlying instrument. The rates are expressed in decimals. The market implied Normal volatilities are converted from basis points to decimals. (Changing the units affects the numerical value and interpretation of the Alpha parameter in the SABR model.)

```
% Load the market implied Normal volatility data for swaptions expiring in one year.
Settle = datetime(2020, 4, 24);
ExerciseDate = datetime(2021, 4, 24);
Basis = 1;
ZeroDates = Settle + [calmonths([3 6 9]) calyears([1 2 3 4 5 ...
    6 7 10 15 20])]';
ZeroRates = [-.54 -.57 -.60 -.62 -.67 -.67 -.65 -.61 ...
    -.56 -.51 -.36 -.19 -.10]'/100;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Compounding',Compounding);

ATMStrike = -0.70/100;
MarketStrikes = ATMStrike + ((-0.5:0.25:1.5)')./100;
MarketVolatilities = [29.89 25.47 23.21 26.17 29.59 33.12 37.81 41.88 46.24]'/10000;

% At the time of Settle, define the underlying forward rate and the at-the-money volatility.
CurrentForwardValue = MarketStrikes(3)
```

```
CurrentForwardValue = -0.0070
```

```
ATMVolatility = MarketVolatilities(3)
```

```
ATMVolatility = 0.0023
```

Method 1: Calibrate Alpha, Rho, and Nu Directly

You can calibrate the Alpha, Rho, and Nu parameters directly. Set the value of the Beta parameter to zero in order to allow negative rates in the SABR model (Normal SABR). After you fix the value of β (Beta), you fit the parameters α (Alpha), ρ (Rho), and ν (Nu) directly. The Optimization Toolbox™ function lsqnonlin generates the parameter values that minimize the squared error between the market volatilities and the volatilities computed by the SABR analytic pricer.

```
% Define the predetermined Beta
Beta1 = 0; % Setting Beta to zero allows negative rates for Normal volatilities

% Calibrate Alpha, Rho, and Nu
objFun = @(X) MarketVolatilities - ...
    volatilities(finpricer("Analytic", 'Model', ...
    finmodel("SABR", 'Alpha', X(1), 'Beta', Beta1, 'Rho', X(2), ...
```

```
    'Nu', X(3), 'VolatilityType', 'Normal'), 'DiscountCurve', ZeroCurve), ...
    ExerciseDate, CurrentForwardValue, MarketStrikes);

% If necessary, adjust the tolerances and stopping criteria for lsqnonlin
X = lsqnonlin(objFun, [ATMVolatility 0 0.5], [0 -1 0], [Inf 1 Inf]);

Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.

Alpha1 = X(1);
Rho1 = X(2);
Nu1 = X(3);
```

Method 2: Calibrate Rho and Nu by Implying Alpha from At-The-Money Volatility

Another method is to use an alternative calibration method. As in the first method, you set the value of β (`Beta`) to zero to allow negative rates. However, after fixing the value of β (`Beta`), you fit the parameters ρ (`Rho`) and ν (`Nu`) directly while α (`Alpha`) is implied from the market at-the-money volatility. Models calibrated using this method produce at-the-money volatilities that are equal to market quotes. This approach can be useful when at-the-money volatilities are quoted most frequently and are important to match. To imply α (`Alpha`) from market at-the-money Normal volatility ($\sigma_{Normal,\,ATM}$), solve the following cubic polynomial for α (`Alpha`), and select the smallest positive real root. This is similar to the approach used for implying α (`Alpha`) from market at-the-money Black volatility [2 on page 2-180]. However, note that the following expression that is used for Normal volatilities is different from the expression that is used for Black volatilities.

$$\frac{\beta(\beta-2)T}{24F^{(2-2\beta)}}\alpha^3+\frac{\rho\beta\nu T}{4F^{(1-\beta)}}\alpha^2+\left(1+\frac{2-3\rho^2}{24}\nu^2T\right)\alpha-\sigma_{Normal,\,ATM}F^{-\beta}=0$$

```
% Define the predetermined Beta
Beta2 = 0; % Setting Beta to zero allows negative rates for Normal volatilities

% Year fraction from Settle date to option maturity
T = yearfrac(Settle, ExerciseDate, Basis);

% This function solves the SABR at-the-money volatility equation as a
% polynomial of Alpha
alpharootsNormal = @(Rho,Nu) roots([...
    Beta2.*(Beta2 - 2)*T/24/CurrentForwardValue^(2 - 2*Beta2) ...
    Rho*Beta2*Nu*T/4/CurrentForwardValue^(1 - Beta2) ...
    (1 + (2 - 3*Rho^2)*Nu^2*T/24) ...
    -ATMVolatility*CurrentForwardValue^(-Beta2)]);

% This function converts at-the-money volatility into Alpha by picking the
% smallest positive real root
atmNormalVol2SabrAlpha = @(Rho,Nu) min(real(arrayfun(@(x) ...
    x*(x>0) + realmax*(x<0 || abs(imag(x))>1e-6), alpharootsNormal(Rho,Nu))));

% Calibrate Rho and Nu (while converting at-the-money volatility into Alpha
% using atmNormalVol2SabrAlpha)
objFun = @(X) MarketVolatilities - ...
    volatilities(finpricer("Analytic", 'Model', ...
    finmodel("SABR", 'Alpha', atmNormalVol2SabrAlpha(X(1), X(2)), ...
    'Beta', Beta2, 'Rho', X(1), 'Nu', X(2), 'VolatilityType', 'Normal'), ...
    'DiscountCurve', ZeroCurve), ...
```

```
    ExerciseDate, CurrentForwardValue, MarketStrikes);

% If necessary, adjust the tolerances and stopping criteria for lsqnonlin
X = lsqnonlin(objFun, [0 0.5], [-1 0], [1 Inf]);

Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.

Rho2 = X(1);
Nu2 = X(2);

% Obtain final Alpha from at-the-money volatility using calibrated parameters
Alpha2 = atmNormalVol2SabrAlpha(Rho2, Nu2);

% Display calibrated parameters
C = {Alpha1 Beta1 Rho1 Nu1;Alpha2 Beta2 Rho2 Nu2};
format;
CalibratedPrameters = cell2table(C,...
    'VariableNames',{'Alpha' 'Beta' 'Rho' 'Nu'},...
    'RowNames',{'Method 1';'Method 2'})

CalibratedPrameters=2×4 table
                 Alpha      Beta       Rho         Nu
               _________    ____    _________    _______

    Method 1    0.0023279     0      -0.010078    0.63538
    Method 2    0.0022389     0      -0.019029    0.66368
```

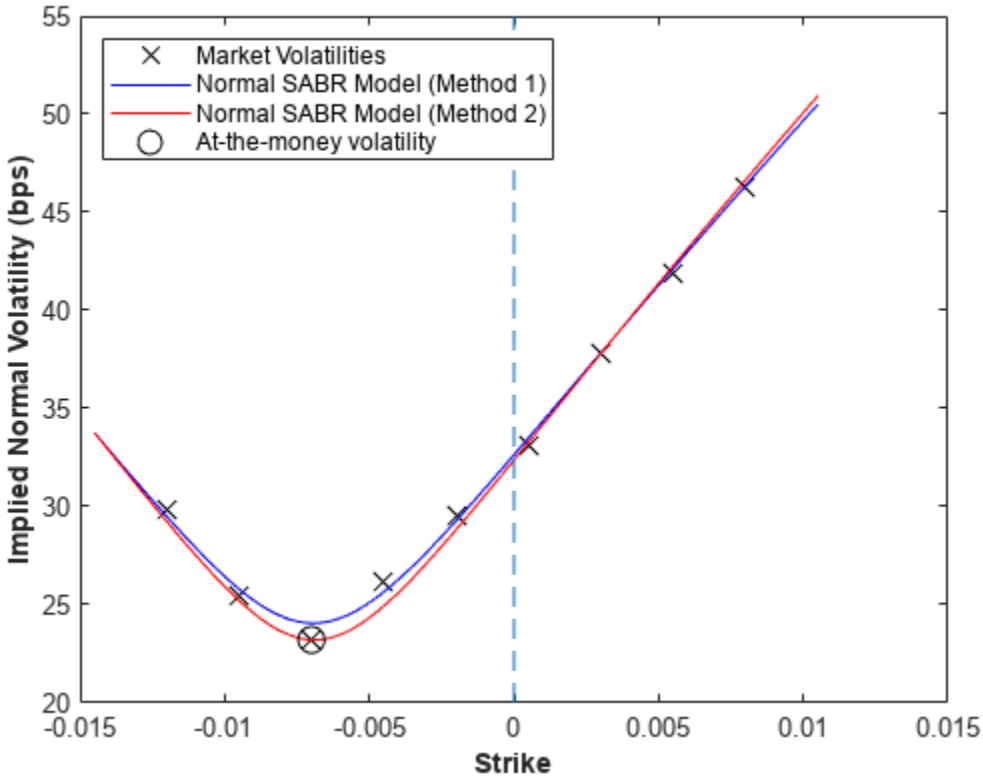
Use the Calibrated Models

Use the calibrated models to compute new volatilities at any strike value, including negative strikes.

Compute volatilities for models calibrated using Method 1 and Method 2, then plot the results. The model calibrated using Method 2 reproduces the market at-the-money volatility (marked with a circle) exactly.

```
PlottingStrikes = (min(MarketStrikes)-0.0025:0.0001:max(MarketStrikes)+0.0025)';

% Compute volatilities for model calibrated by Method 1
SABR_Model_Method_1 = finmodel("SABR", ...
    'Alpha', Alpha1, 'Beta', Beta1, 'Rho', Rho1, 'Nu', Nu1, ...
    'VolatilityType', 'Normal');

ComputedVols1 = volatilities(finpricer("Analytic", ...
    'Model', SABR_Model_Method_1, 'DiscountCurve', ZeroCurve), ...
    ExerciseDate, CurrentForwardValue, PlottingStrikes);

% Compute volatilities for model calibrated by Method 2
SABR_Model_Method_2 = finmodel("SABR", ...
    'Alpha', Alpha2, 'Beta', Beta2, 'Rho', Rho2, 'Nu', Nu2, ...
    'VolatilityType', 'Normal');

ComputedVols2 = volatilities(finpricer("Analytic", ...
    'Model', SABR_Model_Method_2, 'DiscountCurve', ZeroCurve), ...
    ExerciseDate, CurrentForwardValue, PlottingStrikes);
```

```
figure;
plot(MarketStrikes,MarketVolatilities*10000,'xk',...
    PlottingStrikes,ComputedVols1*10000,'b', ...
    PlottingStrikes,ComputedVols2*10000,'r', ...
    CurrentForwardValue,ATMVolatility*10000,'ok',...
    'MarkerSize',10);

h = gca;
line([0,0],[min(h.YLim),max(h.YLim)],'LineStyle','--');

xlabel('Strike', 'FontWeight', 'bold');
ylabel('Implied Normal Volatility (bps)', 'FontWeight', 'bold');
legend('Market Volatilities', 'Normal SABR Model (Method 1)', ...
    'Normal SABR Model (Method 2)', 'At-the-money volatility', ...
    'Location', 'northwest');
```



References

[1] Hagan, Patrick S., Deep Kumar, Andrew S. Lesniewski, and Diana E. Woodward. "Managing Smile Risk." *Wilmott Magazine*, (January 2002):84-108.

[2] West, Graeme. "Calibration of the SABR Model in Illiquid Markets." *Applied Mathematical Finance*. 12, no. 4 (December 2005): 371–385.

Calibrate SABR Model Using Analytic Pricer

This example shows how to use two different methods to calibrate a SABR stochastic volatility model from market implied Black volatilities. Both approaches use the SABR analytic pricer.

Load Market Implied Black Volatility Data

This example sets up hypothetical market implied Black volatilities for European swaptions over a range of strikes before calibration. The swaptions expire in three years from the `Settle` date and have five-year swaps as the underlying instrument. The rates are expressed in decimals. (Changing the units affects the numerical value and interpretation of the `Alpha` parameter in the SABR model.)

Load the market implied Black volatility data for swaptions expiring in three years.

```
Settle = datetime(2013, 7, 10);
ZeroDates = Settle + [calmonths([1 2 3 6 9]) calyears([1 2 3 4 5 6 7 8 9 10 12])]';
ZeroRates = [0.25 0.3 0.33 0.42 0.8 0.9 1.1 1.2 ...
    1.8 2.2 2.4 2.71 2.95 3.02 3.24 3.58]'/100;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Compounding',Compounding);
ExerciseDate = datetime(2016, 7, 10);
MarketStrikes = [2.46 2.96 3.46 3.96 4.46 4.96 5.46]'/100;
MarketVolatilities = [44.3 40.2 36.7 35.7 37.2 38.1 39.8]'/100;
```

At the time of `Settle`, define the underlying forward rate and the at-the-money volatility.

```
CurrentForwardValue = MarketStrikes(4)
```

```
CurrentForwardValue = 0.0396
```

```
ATMVolatility = MarketVolatilities(4)
```

```
ATMVolatility = 0.3570
```

Method 1: Calibrate `Alpha`, `Rho`, and `Nu` Directly

You can calibrate the `Alpha`, `Rho`, and `Nu` model parameters directly. Set the value of `Beta` either by fitting historical market volatility data or by choosing a value appropriate for the market [1 on page 2-184]. For this example, use the value `0.5`.

```
% Define the predetermined Beta.
Beta1 = 0.5;
```

After fixing the value of β (`Beta`), fit the parameters α (`Alpha`), ρ (`Rho`), and ν (`Nu`) directly. The Optimization Toolbox™ function `lsqnonlin` generates the parameter values that minimize the squared error between the market volatilities and the volatilities computed by the SABR analytic pricer.

```
% Calibrate Alpha, Rho, and Nu.
objFun = @(X) MarketVolatilities - ...
    volatilities(finpricer("Analytic", 'Model', ...
    finmodel("SABR", 'Alpha', X(1), 'Beta', Beta1, 'Rho', X(2), 'Nu', X(3)), ...
    'DiscountCurve', ZeroCurve), ExerciseDate, CurrentForwardValue, MarketStrikes);

X = lsqnonlin(objFun, [0.5 0 0.5], [0 -1 0], [Inf 1 Inf]);
```

```
Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.
```

```
Alpha1 = X(1);
Rho1 = X(2);
Nu1 = X(3);
```

Method 2: Calibrating Rho and Nu by Implying Alpha from At-The-Money Volatility

You can also use an alternative calibration method. Set the value of β (`Beta`) as in the first method.

```
Beta2 = 0.5;
```

Next you fit the parameters ρ (`Rho`) and ν (`Nu`) directly while α (`Alpha`) is implied from the market at-the-money volatility. Models calibrated using this method produce at-the-money volatilities that are equal to market quotes. This approach is widely used in swaptions, where at-the-money volatilities are quoted most frequently and are important to match. To imply α (`Alpha`) from market at-the-money volatility (σ_{ATM}), the following cubic polynomial is solved for α (`Alpha`), and the smallest positive real root is selected [2 on page 2-184].

$$\frac{(1-\beta)^2 T}{24F^{(2-2\beta)}}\alpha^3 + \frac{\rho\beta\nu T}{4F^{(1-\beta)}}\alpha^2 + \left(1 + \frac{2-3\rho^2}{24}\nu^2 T\right)\alpha - \sigma_{ATM}F^{(1-\beta)} = 0$$

Here:

F is the current forward value

T is the year fraction to maturity.

Fit the parameters by defining an anonymous function.

```
% Year fraction from Settle to option maturity.
T = yearfrac(Settle, ExerciseDate, 1);

% This function solves the SABR at-the-money volatility equation as a
% polynomial of Alpha.
alpharoots = @(Rho,Nu) roots([...
    (1 - Beta2)^2*T/24/CurrentForwardValue^(2 - 2*Beta2) ...
    Rho*Beta2*Nu*T/4/CurrentForwardValue^(1 - Beta2) ...
    (1 + (2 - 3*Rho^2)*Nu^2*T/24) ...
    -ATMVolatility*CurrentForwardValue^(1 - Beta2)]);

% This function converts at-the-money volatility into Alpha by picking the
% smallest positive real root.
atmVol2SabrAlpha = @(Rho,Nu) min(real(arrayfun(@(x) ...
    x*(x>0) + realmax*(x<0 || abs(imag(x))>1e-6), alpharoots(Rho,Nu))));
```

The function `atmVol2SabrAlpha` converts at-the-money volatility into α (`Alpha`) for a given set of ρ (`Rho`) and ν (`Nu`). This function is then used in the objective function to fit the parameters ρ (`Rho`) and ν (`Nu`).

```
% Calibrate Rho and Nu (while converting at-the-money volatility into Alpha
% using atmVol2SabrAlpha).

objFun = @(X) MarketVolatilities - ...
```

```
    volatilities(finpricer("Analytic", 'Model', finmodel("SABR", ...
    'Alpha', atmVol2SabrAlpha(X(1), X(2)), 'Beta', Beta2, 'Rho', X(1), 'Nu', X(2)), ...
    'DiscountCurve', ZeroCurve), ExerciseDate, CurrentForwardValue, MarketStrikes);

X = lsqnonlin(objFun, [0 0.5], [-1 0], [1 Inf]);

Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.

Rho2 = X(1);
Nu2 = X(2);
```

The calibrated parameter α (`Alpha`) is computed using the calibrated parameters ρ (`Rho`) and ν (`Nu`).

```
% Obtain final Alpha from at-the-money volatility using calibrated
% parameters.
Alpha2 = atmVol2SabrAlpha(Rho2, Nu2);

% Display calibrated parameters.
C = {Alpha1 Beta1 Rho1 Nu1;Alpha2 Beta2 Rho2 Nu2};
CalibratedPrameters = cell2table(C,...
    'VariableNames',{'Alpha' 'Beta' 'Rho' 'Nu'},...
    'RowNames',{'Method 1';'Method 2'})

CalibratedPrameters=2×4 table
                Alpha      Beta      Rho        Nu
              ________    ____    _______    _______

    Method 1    0.060203    0.5     0.19131    0.85327
    Method 2    0.058851    0.5     0.18901    0.88627
```

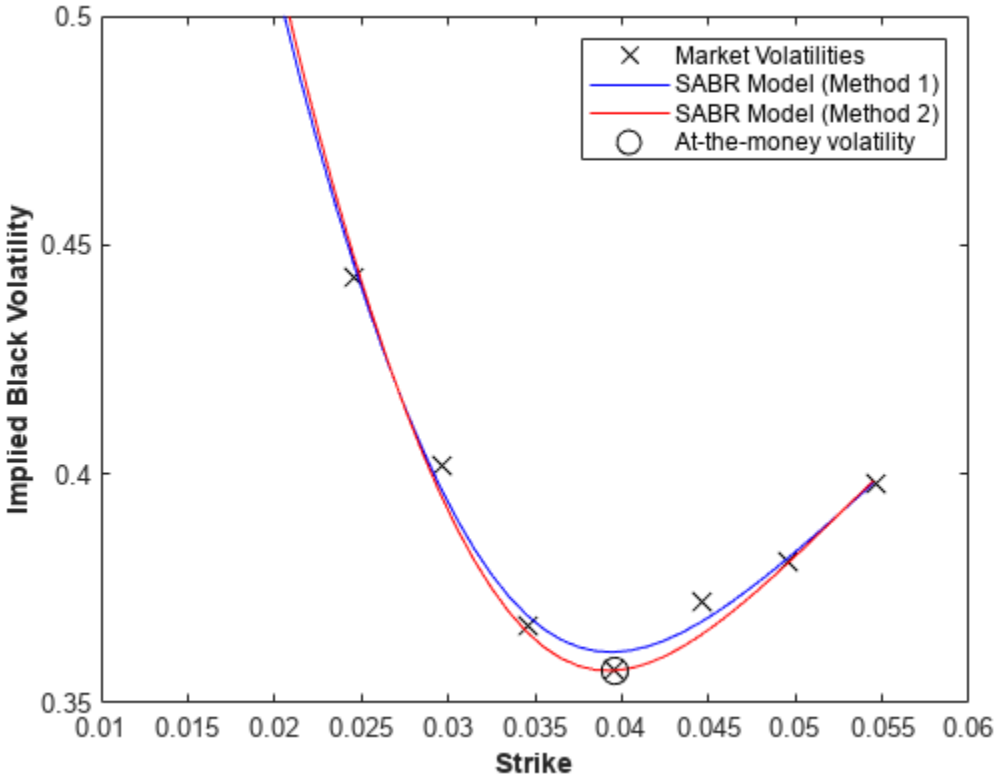
Use Calibrated Models

Use the calibrated models to compute new volatilities at any strike value.

Compute volatilities for models calibrated using Method 1 and Method 2 and plot the results.

```
PlottingStrikes = (1.75:0.1:5.50)'/100;

% Compute volatilities for model calibrated by Method 1.
SABR_Model_Method_1 = finmodel("SABR", ...
    'Alpha', Alpha1, 'Beta', Beta1, 'Rho', Rho1, 'Nu', Nu1);

ComputedVols1 = volatilities(finpricer("Analytic", ...
    'Model', SABR_Model_Method_1, 'DiscountCurve', ZeroCurve), ...
    ExerciseDate, CurrentForwardValue, PlottingStrikes);

% Compute volatilities for model calibrated by Method 2.
SABR_Model_Method_2 = finmodel("SABR", ...
    'Alpha', Alpha2, 'Beta', Beta2, 'Rho', Rho2, 'Nu', Nu2);

ComputedVols2 = volatilities(finpricer("Analytic", ...
    'Model', SABR_Model_Method_2, 'DiscountCurve', ZeroCurve), ...
    ExerciseDate, CurrentForwardValue, PlottingStrikes);

figure;
```

```
plot(MarketStrikes,MarketVolatilities,'xk',...
    PlottingStrikes,ComputedVols1,'b', ...
    PlottingStrikes,ComputedVols2,'r', ...
    CurrentForwardValue,ATMVolatility,'ok',...
    'MarkerSize',10);
xlim([0.01 0.06]);
ylim([0.35 0.5]);
xlabel('Strike', 'FontWeight', 'bold');
ylabel('Implied Black Volatility', 'FontWeight', 'bold');
legend('Market Volatilities', 'SABR Model (Method 1)',...
    'SABR Model (Method 2)', 'At-the-money volatility');
```



The model calibrated using Method 2 reproduces the market at-the-money volatility (marked with a circle) exactly.

References

[1] Hagan, Patrick S., Deep Kumar, Andrew S. Lesniewski, and Diana E. Woodward. "Managing Smile Risk." *Wilmott Magazine*. (January 2002): 84–108.

[2] West, Graeme. "Calibration of the SABR Model in Illiquid Markets." *Applied Mathematical Finance*. 12, no. 4 (December 2005): 371–385.

Price a Swaption Using SABR Model and Analytic Pricer

This example shows how to price a swaption using the SABR model. First, you construct a swaption volatility surface from market volatilities by calibrating the SABR model parameters separately for each swaption maturity using the SABR analytic pricer. You then compute the swaption price by using the implied Black volatility on the surface with the SABR analytic pricer.

Step 1. Load market swaption volatility data.

Load the zero curve and market implied Black volatility data for swaptions.

```
Settle = datetime(2013, 6, 14);
ZeroDates = Settle + [calmonths([1 3 6]) calyears([1 2 3 4 5 6 7 8 9 10 12 15 20])]';
ZeroRates = [0.22 0.31 0.45 0.73 0.54 0.72 1.22 1.54 1.83 1.92 ...
    2.16 2.32 2.52 2.93 3.12 3.36]'/100;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Compounding',1);
ExerciseDates =  Settle + [calmonths(3) calyears([1 2 3 4 5 7 10])]';
YearsToExercise = yearfrac(Settle,ExerciseDates,1);
NumMaturities = length(YearsToExercise);

MarketVolatilities = [ ...
   56.5 52.7 49.1 44.9 43.5 40.5 34.8 32.2
   45.8 46.2 44.2 41.1 39.1 36.1 33.2 31.3
   34.7 38.8 39.0 37.2 36.8 33.2 30.1 28.1
   33.9 35.9 36.9 35.8 34.2 30.5 29.0 27.0
   40.8 41.2 38.6 37.0 35.3 32.0 29.5 26.5
   45.1 42.8 41.2 38.3 37.2 33.2 30.3 27.2
   50.2 45.4 43.2 39.9 38.0 34.1 31.5 28.3]/100;

MarketStrikes = [ ...
  1.02 1.31 1.78 2.08 2.21 2.34 2.60 2.69;
  1.52 1.81 2.28 2.58 2.71 2.84 3.10 3.19;
  2.02 2.31 2.78 3.08 3.21 3.34 3.60 3.69;
  2.52 2.81 3.28 3.58 3.71 3.84 4.10 4.19;
  3.02 3.31 3.78 4.08 4.21 4.34 4.60 4.69;
  3.52 3.81 4.28 4.58 4.71 4.84 5.10 5.19;
  4.02 4.31 4.78 5.08 5.21 5.34 5.60 5.69]/100;

CurrentForwardValues = MarketStrikes(4,:)
```

```
CurrentForwardValues = 1×8

    0.0252    0.0281    0.0328    0.0358    0.0371    0.0384    0.0410    0.0419
```

```
ATMVolatilities = MarketVolatilities(4,:)
```

```
ATMVolatilities = 1×8

    0.3390    0.3590    0.3690    0.3580    0.3420    0.3050    0.2900    0.2700
```

The current underlying forward rates and the corresponding at-the-money volatilities across the eight swaption maturities are represented in the fourth rows of the two matrices.

Step 2. Calibrate the SABR model parameters for each swaption maturity.

When you use a static SABR model, where the model parameters are assumed to be constant with respect to time, the parameters are calibrated separately for each swaption maturity (years to exercise) in a `for` loop using the SABR analytic pricer. To better represent market at-the-money volatilities, the `Alpha` parameter values are implied by the market at-the-money volatilities (for details, see Method 2 in "Calibrate SABR Model Using Analytic Pricer" on page 2-181).

```
% Define the predetermined Beta, calibrate SABR model parameters for each
% swaption maturity, and display the calibrated parameters in a table.
Beta = 0.5;
Betas = repmat(Beta, NumMaturities, 1);
Alphas = zeros(NumMaturities, 1);
Rhos = zeros(NumMaturities, 1);
Nus = zeros(NumMaturities, 1);

options = optimoptions('lsqnonlin','Display','none');

for k = 1:NumMaturities
    % This function solves the SABR at-the-money volatility equation as a
    % polynomial of Alpha.
    alpharoots = @(Rho,Nu) roots([...
        (1 - Beta)^2*YearsToExercise(k)/24/CurrentForwardValues(k)^(2 - 2*Beta) ...
        Rho*Beta*Nu*YearsToExercise(k)/4/CurrentForwardValues(k)^(1 - Beta) ...
        (1 + (2 - 3*Rho^2)*Nu^2*YearsToExercise(k)/24) ...
        -ATMVolatilities(k)*CurrentForwardValues(k)^(1 - Beta)]);

    % This function converts at-the-money volatility into Alpha by picking the
    % smallest positive real root.
    atmVol2SabrAlpha = @(Rho,Nu) min(real(arrayfun(@(x) ...
        x*(x>0) + realmax*(x<0 || abs(imag(x))>1e-6), alpharoots(Rho,Nu))));

    % Fit Rho and Nu (while converting at-the-money volatility into Alpha).
    objFun = @(X) MarketVolatilities(:,k) - ...
    volatilities(finpricer("Analytic", 'Model', finmodel("SABR", ...
    'Alpha', atmVol2SabrAlpha(X(1), X(2)), 'Beta', Beta, 'Rho', X(1), 'Nu', X(2)), ...
    'DiscountCurve', ZeroCurve), ExerciseDates(k), CurrentForwardValues(k), MarketStrikes(:,k));

    X = lsqnonlin(objFun, [0 0.5], [-1 0], [1 Inf], options);
    Rho = X(1);
    Nu = X(2);

    % Get final Alpha from the calibrated parameters.
    Alpha = atmVol2SabrAlpha(Rho, Nu);

    Alphas(k) = Alpha;
    Rhos(k) = Rho;
    Nus(k) = Nu;
end

CalibratedParameters = array2table([Alphas Betas Rhos Nus],...
    'VariableNames',{'Alpha' 'Beta' 'Rho' 'Nu'},...
    'RowNames',{'3M into 10Y';'1Y into 10Y';...
    '2Y into 10Y';'3Y into 10Y';'4Y into 10Y';...
    '5Y into 10Y';'7Y into 10Y';'10Y into 10Y'})
```

```
CalibratedParameters=8×4 table
                     Alpha      Beta      Rho         Nu
```

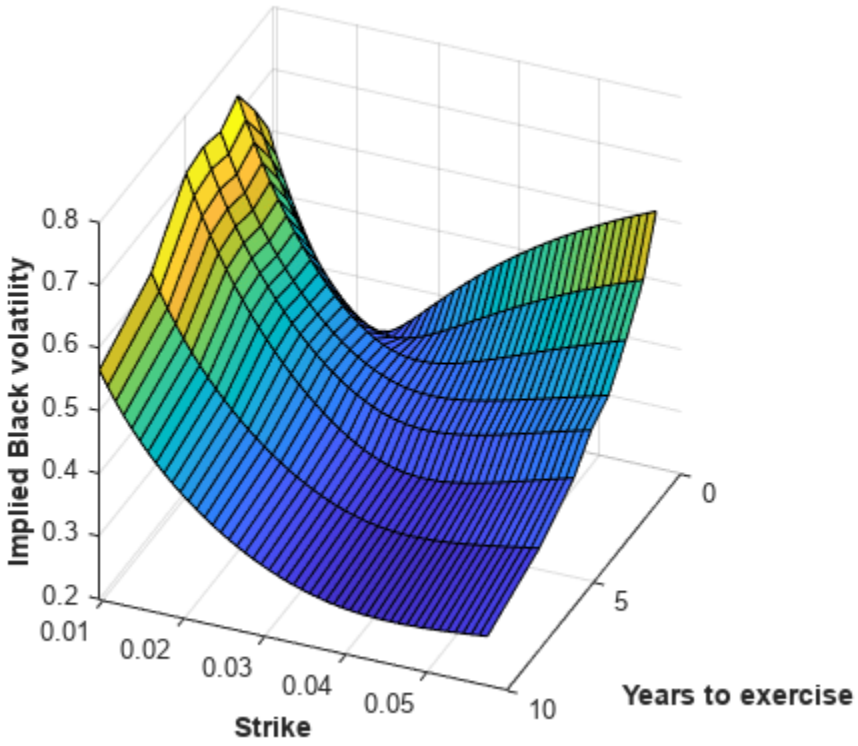
```
                  ________    ____    ________    _______

    3M into 10Y     0.051895    0.5      0.40869     1.4054
    1Y into 10Y     0.054381    0.5      0.28066     1.1234
    2Y into 10Y     0.057325    0.5       0.2166    0.97407
    3Y into 10Y     0.057243    0.5      0.19837    0.82932
    4Y into 10Y     0.053387    0.5      0.17304    0.81441
    5Y into 10Y      0.04673    0.5      0.11618    0.80138
    7Y into 10Y     0.046986    0.5       0.1554    0.63377
    10Y into 10Y     0.04443    0.5     0.080169    0.52515
```

Step 3. Construct a volatility surface.

Use the calibrated SABR model to compute new volatilities at any strike value to produce a smooth smile for a given maturity. This can be repeated for each maturity to form a volatility surface.

Compute volatilities using the calibrated models for each maturity and plot the volatility surface.

```
PlottingStrikes = (0.95:0.1:5.8)'/100;
ComputedVols = zeros(length(PlottingStrikes), NumMaturities);

for k = 1:NumMaturities
    SABRModel = finmodel("SABR", ...
        'Alpha', Alphas(k), 'Beta', Betas(k), 'Rho', Rhos(k), 'Nu', Nus(k));

    ComputedVols(:,k) = volatilities(finpricer("Analytic", ...
        'Model', SABRModel, 'DiscountCurve', ZeroCurve), ...
        ExerciseDates(k), CurrentForwardValues(k), PlottingStrikes);
end

figure;
surf(YearsToExercise, PlottingStrikes, ComputedVols);
xlim([0 10]); ylim([0.0095 0.06]); zlim([0.2 0.8]);
view(113,32);
set(gca, 'Position', [0.13 0.11 0.775 0.815], ...
    'PlotBoxAspectRatioMode', 'manual');
xlabel('Years to exercise', 'Fontweight', 'bold');
ylabel('Strike', 'Fontweight', 'bold');
zlabel('Implied Black volatility', 'Fontweight', 'bold');
```



Note that in this volatility surface, the smiles tend to get flatter for longer swaption maturities (years to exercise). This is consistent with the Nu parameter values tending to decrease with swaption maturity, as shown in the table for `CalibratedParameters`.

Step 4. Use the SABR analytic pricer to price a swaption.

Use the SABR analytic pricer to price a swaption that matures in five years. First, create the underlying 10-year swap instrument starting in five years and create the `Swaption` instrument for this underlying `Swap`. Then create the SABR model for this swaption maturing in five years, which is used for creating the SABR analytic pricer.

```
% Create the underlying 10-year swap starting in 5 years.
MaturityIdx = 6;
SwapStartDate = ExerciseDates(MaturityIdx);
SwapMaturity = SwapStartDate + calyears(10);
Swap = fininstrument("Swap", 'Maturity', SwapMaturity, ...
    'LegRate', [0 0], "LegType",["fixed" "float"],...
    "ProjectionCurve", ZeroCurve, "StartDate", SwapStartDate)
```

```
Swap =
  Swap with properties:

                    LegRate: [0 0]
                    LegType: ["fixed"    "float"]
                      Reset: [2 2]
                      Basis: [0 0]
                   Notional: 100
```

```
          LatestFloatingRate: [NaN NaN]
                 ResetOffset: [0 0]
    DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [1x2 ratecurve]
       BusinessDayConvention: ["actual"    "actual"]
                    Holidays: NaT
                EndMonthRule: [1 1]
                   StartDate: 14-Jun-2018
                    Maturity: 14-Jun-2028
                        Name: ""
```

```
% Create the swaption (for the underlying 10-year swap) maturing in five
% years.
SwaptionExerciseDate = SwapStartDate;
Reset = 1;
OptSpec = 'call';
Strike = 0.0272;
Swaption = fininstrument("Swaption", 'Strike', Strike, ...
    'ExerciseDate', SwaptionExerciseDate, 'Swap', Swap, 'OptionType', OptSpec)
```

```
Swaption =
  Swaption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 14-Jun-2018
           Strike: 0.0272
             Swap: [1x1 fininstrument.Swap]
             Name: ""
```

```
% Create the SABR model for the swaption maturing in five years.
SABRModel = finmodel("SABR", ...
        'Alpha', Alphas(MaturityIdx), 'Beta', Betas(MaturityIdx), ...
        'Rho', Rhos(MaturityIdx), 'Nu', Nus(MaturityIdx))
```

```
SABRModel =
  SABR with properties:

            Alpha: 0.0467
             Beta: 0.5000
              Rho: 0.1162
               Nu: 0.8014
            Shift: 0
    VolatilityType: "black"
```

```
% Create the SABR analytic pricer.
SABRPricer = finpricer("Analytic", 'Model', SABRModel, 'DiscountCurve', ZeroCurve)
```

```
SABRPricer =
  SABR with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.SABR]
```

To visualize the SABR implied Black volatility used in pricing the swaption, first compute the current forward swap rate by using the `parswaprate` function.

```
CurrentForwardSwapRate = parswaprate(Swap,ZeroCurve)
```

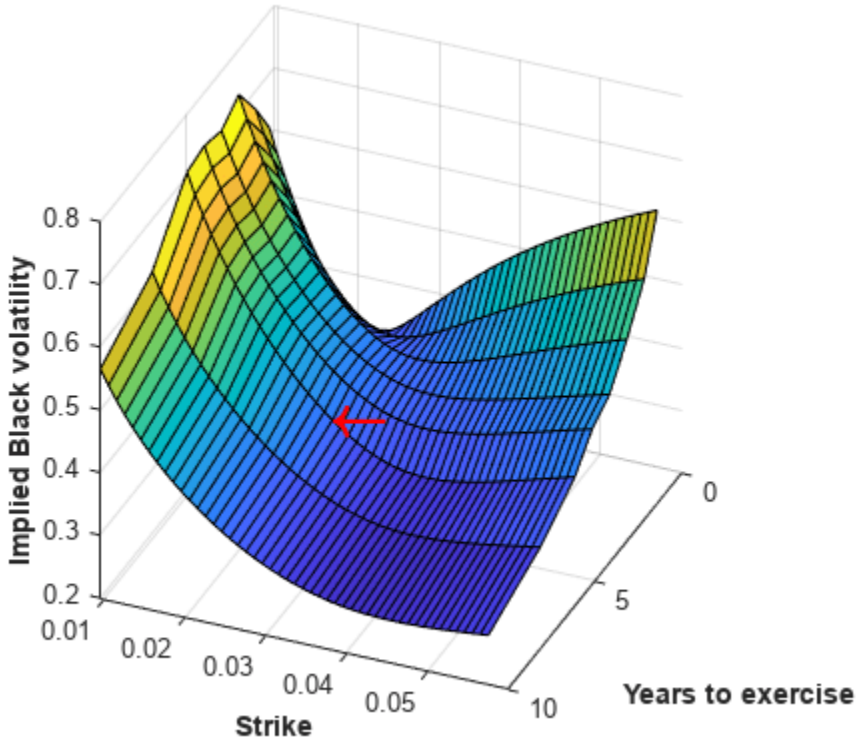
```
CurrentForwardSwapRate = 0.0384
```

Next, compute the SABR implied Black volatility for this `Swaption` by using the `volatilities` function, and it is marked with a red arrow in the figure at the bottom.

```
SABRBlackVolatility = volatilities(SABRPricer, ...
         SwaptionExerciseDate, CurrentForwardSwapRate, Strike)
```

```
SABRBlackVolatility = 0.3665
```

```
text (YearsToExercise(MaturityIdx), Strike, SABRBlackVolatility, ...
    '\leftarrow', 'Color', 'r', 'FontWeight', 'bold', 'FontSize', 22);
```



Finally, price the swaption using the `price` function of the `SABR` analytic pricer.

```
SwaptionPrice = price(SABRPricer, Swaption)
```

```
SwaptionPrice = 13.0141
```

References

[1] Hagan, Patrick S., Deep Kumar, Andrew S. Lesniewski, and Diana E. Woodward. "Managing Smile Risk." *Wilmott Magazine*. (January 2002): 84–108.

[2] West, Graeme. "Calibration of the SABR Model in Illiquid Markets." *Applied Mathematical Finance*. 12, no. 4 (December 2005): 371–385.

Compute LIBOR Fallback

This example shows how to compute a USD LIBOR fallback. Regulators and industry groups have recommended that firms transition away from the London inter-bank offered rate (LIBOR) and prepare to replace them with overnight Alternative Reference Rates (ARRs). What happens to contracts with a notional value of trillions of dollars if they refer to a benchmark that no longer exists? If the LIBOR benchmark is no longer published, references to that benchmark rate must change and benchmark rates “fall back” to a new benchmark in contracts. For example, if a 30-year floating-rate instrument with a three-month coupon based on the LIBOR rate is created in 2008 and expires in 2038, then the rate will need to change in 2023 because in 2023, the publication of a LIBOR rate permanently ceases. To calculate three-month coupon payments after 2023, you must use a LIBOR fallback. This example is based on the ISDA® 2020 IBOR Fallbacks Protocol.

Spread Adjustments

Use spread adjustments from the ISDA® website at LIBOR Cessation and the Impact on Fallbacks Protocol.

```
Adjustment = [.00644 .03839 .11448 .18456 .26161 .42826 .71513]'/100;
TenorLabel = ["ON","1W","1M","2M","3M","6M","12M]';
Tenors = [caldays(1) calweeks(1) calmonths([1 2 3 6 12])];
SpreadAdjustmentTable = table(TenorLabel,Adjustment);
nTenors = height(SpreadAdjustmentTable);
```

Example Data

Run this example using the following example data.

```
RateRecordDate = datetime(2021,2,26);
RateTenor = "1M";
ARR_DC = 360;
```

Obtain Calculation Date

Use `RateRecordDate` and `Tenors` to calculate `CalculationDate`.

```
CalculationDate = RateRecordDate + Tenors(RateTenor == TenorLabel);
if ~isbusday(CalculationDate)
    CalculationDate = busdate(CalculationDate);
end
```

Obtain Historical Data

For this example, the historical data is hard-coded. However, you can also use Datafeed Toolbox™ with Federal Reserve Economic Data (FRED®) to obtain the historical data.

```
getFredData = false;
if getFredData
    ARR_ID = 'SOFR';
    c = fred;
    c.DataReturnFormat = 'timetable';
    c.DatetimeType = 'datetime';
    FredData = fetch(c,ARR_ID,RateRecordDate,CalculationDate);
    SOFRData = FredData.Data{1};
    SOFRData(isnan(SOFRData{:,1}),:) = [];
else
```

```
    SOFRRates = [0.01 0.02 0.04 0.04 0.02 0.02 0.02 0.02 0.02 0.01 ...
        0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01]';
    SOFRDates = busdays(RateRecordDate,CalculationDate);
    SOFRData = timetable(SOFRDates,SOFRRates);
end
```

Compute Spread Adjustment

Obtain the spread adjustment from `SpreadAdjustmentTable`.

```
SpreadAdj = Adjustment(RateTenor == TenorLabel);
```

Compute ARR

Compute the alternative reference rate (ARR) using the relevant reference rate data.

```
tau = days(diff(SOFRData.Properties.RowTimes))/ARR_DC;
relRate = SOFRData{1:end-1,1};
CompRate = prod(1 + tau.*1/100.*relRate) - 1;
ARR = ARR_DC/days(CalculationDate - RateRecordDate)*CompRate;
ARR = round(ARR,7);
```

Compute Fallback Rate

`FallbackRate` is the sum of `ARR` and `SpreadAdj`.

```
FallbackRate = ARR + SpreadAdj
```

```
FallbackRate = 0.0013
```

Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond

This example demonstrates how to use `treeviewer` to examine tree information for a Hull-White tree when you price a European callable bond.

Specify Input Parameters

Define the interest-rate curve information.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2019';
EndDates = {'Jan-1-2020'; 'Jan-1-2021'; 'Jan-1-2022'; 'Jan-1-2023'};
Compounding = 1;
```

Define the callable bond instruments. The first instrument is the first entry in the arrays. So, for example, the first instrument has a strike price of $98 and a maturity of January 1, 2022.

```
Settle = '01-Jan-2019';
Maturity = {'01-Jan-2022'; '01-Jan-2023'};
CouponRate = {{'01-Jan-2021' .0425; '01-Jan-2023' .0450}};
OptType = 'call';
Strike = [98; 95];
ExerciseDates= {'01-Jan-2021'; '01-Jan-2022'};
Basis = 1;
```

Define the volatility information and HW one-factor parameters.

```
VolDates = ['1-Jan-2020'; '1-Jan-2021'; '1-Jan-2022'; '1-Jan-2023'];
VolCurve = 0.05;
AlphaDates = '01-01-2023';
AlphaCurve = 0.05;
```

Build Hull-White One-Factor Tree

Use `hwtree` to build a one-factor tree.

```
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis);
HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve, AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWTimeSpec.Basis = Basis;

HWT = hwtree(HWVolSpec, RateSpec, HWTimeSpec);
```

Price Both Callable Instruments

Use `optembndbyhw` to price the callable bond with embedded options.

```
[Price, PriceTree] = optembndbyhw(HWT, CouponRate,  Settle, Maturity, OptType, Strike,...
ExerciseDates, 'Period', 1, 'Basis', Basis)
```

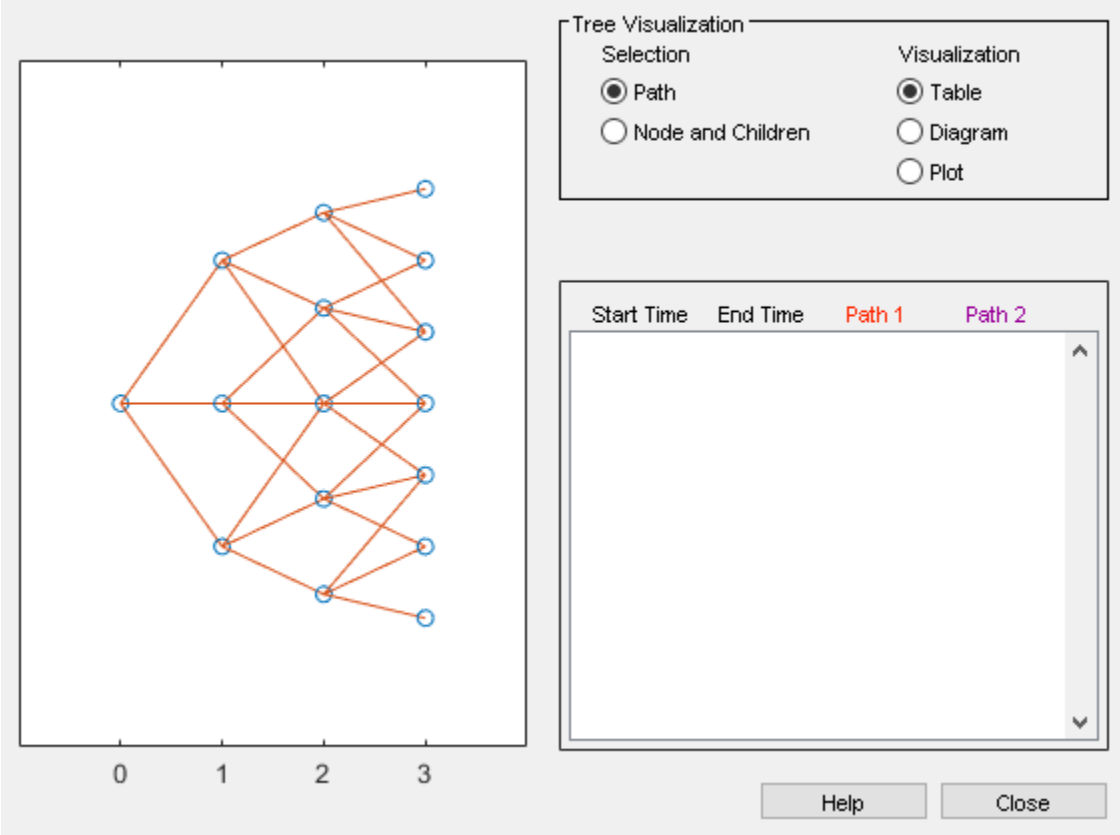
```
Price = 2×1

   96.4131
   92.9341
```

```
PriceTree = struct with fields:
                FinObj: 'HWPriceTree'
                  tObs: [0 1 2 3 4]
                 PTree: {1x5 cell}
               ProbTree: {1x5 cell}
                 ExTree: {1x5 cell}
             ExProbTree: {1x5 cell}
    ExProbsByTreeLevel: [2x5 double]
               Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
```

Examine Hull-White Tree Structure

Use `treeviewer` to examine the Hull-White interest-rate tree that is the input for the embedded option pricer.

```
treeviewer(HWT)
```



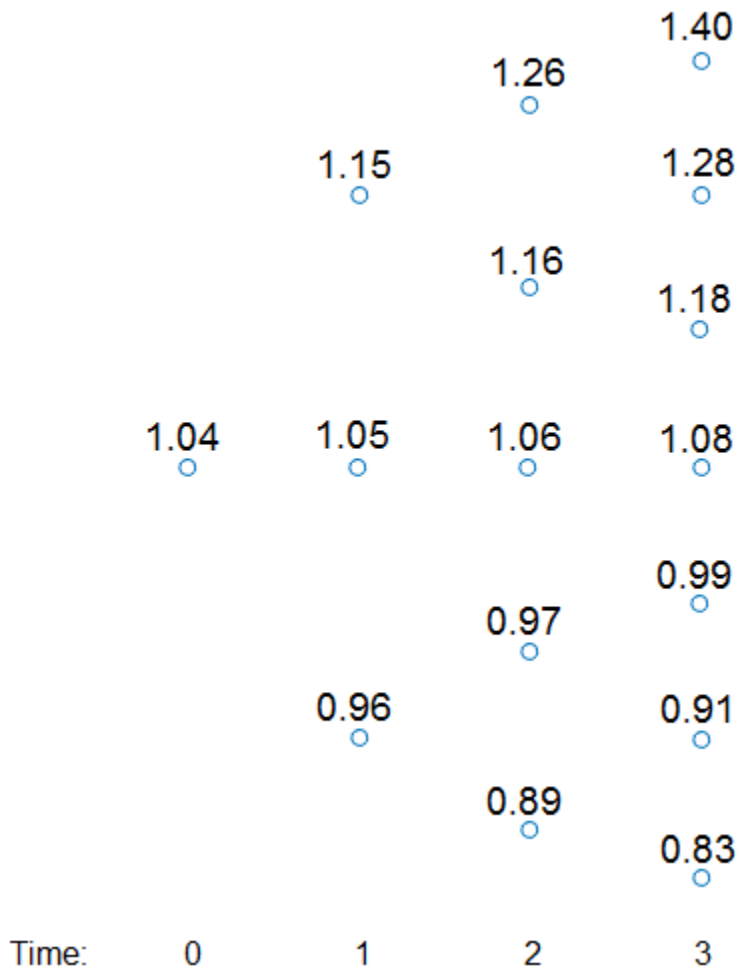
The Hull-White tree has 4 levels of nodes. The root node is at $t = 0$, three nodes are at $t = 1$, five nodes are at $t = 2$, and seven nodes are at $t = 3$. Each node represents a particular state. In this case, the state is defined by the forward interest-rate curve, `HWT.FwdTree`. The combination of `HWT.FwdTree` and `HWT.Connect` defines the tree structure. `FwdTree` contains the values of the forward interest rate at each node. 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.

For example, `HWT.FwdTree` is:

```
HWT.FwdTree
```

```
ans=1×4 cell array
    {[1.0350]}    {[1.1457 ... ]}    {[1.2639 ... ]}    {[1.4003 ... ]}
```

If you display the nodes graphically with the forward rates superimposed it looks like:



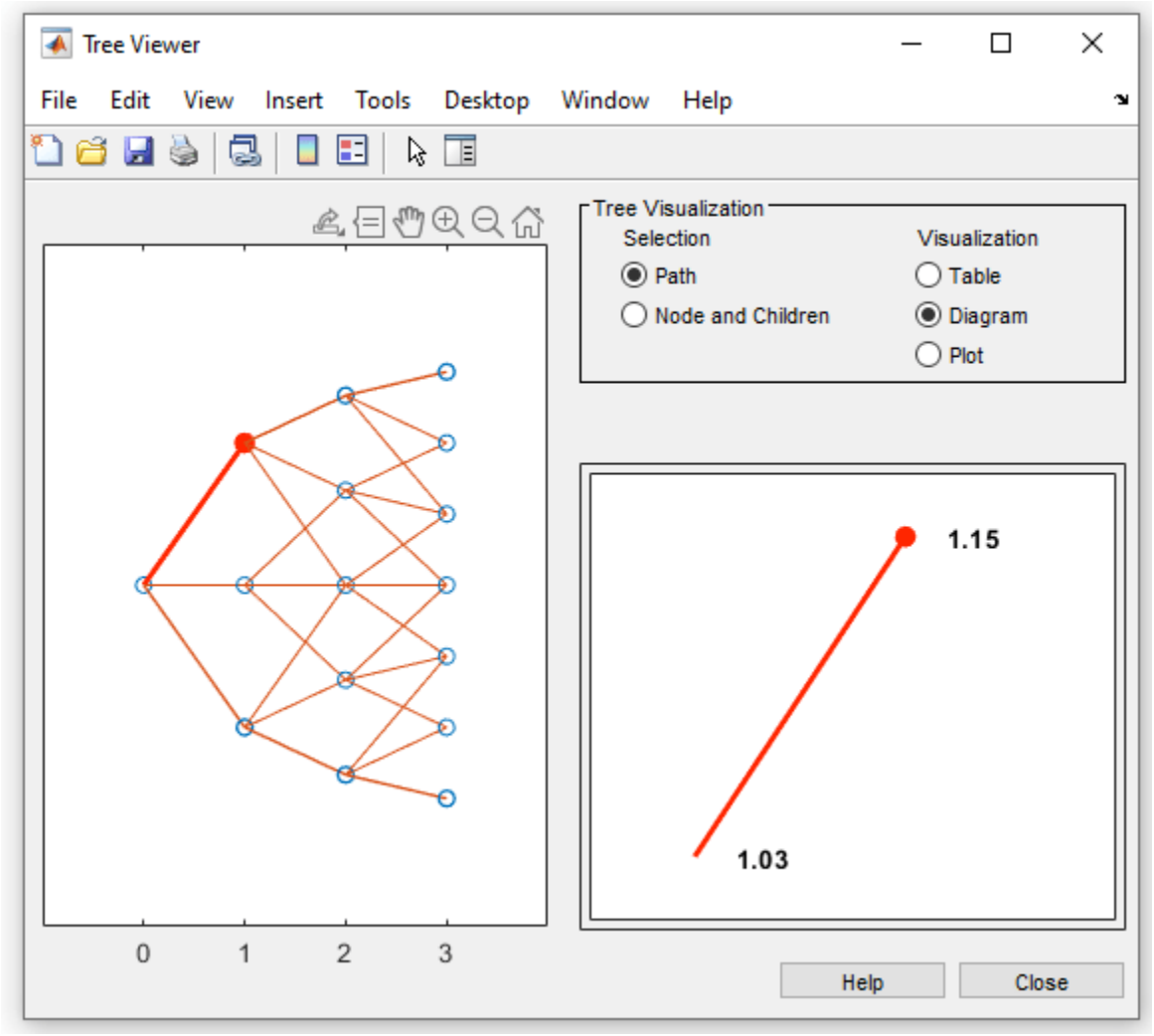
You can use the `treeviewer` function to visualize the rates in the tree with `treeviewer(HWT)`. This function displays the structure of a Hull-White tree (`HWT`) in the left pane. The tree visualization in the right pane is blank. Visualize the actual interest-rate tree:

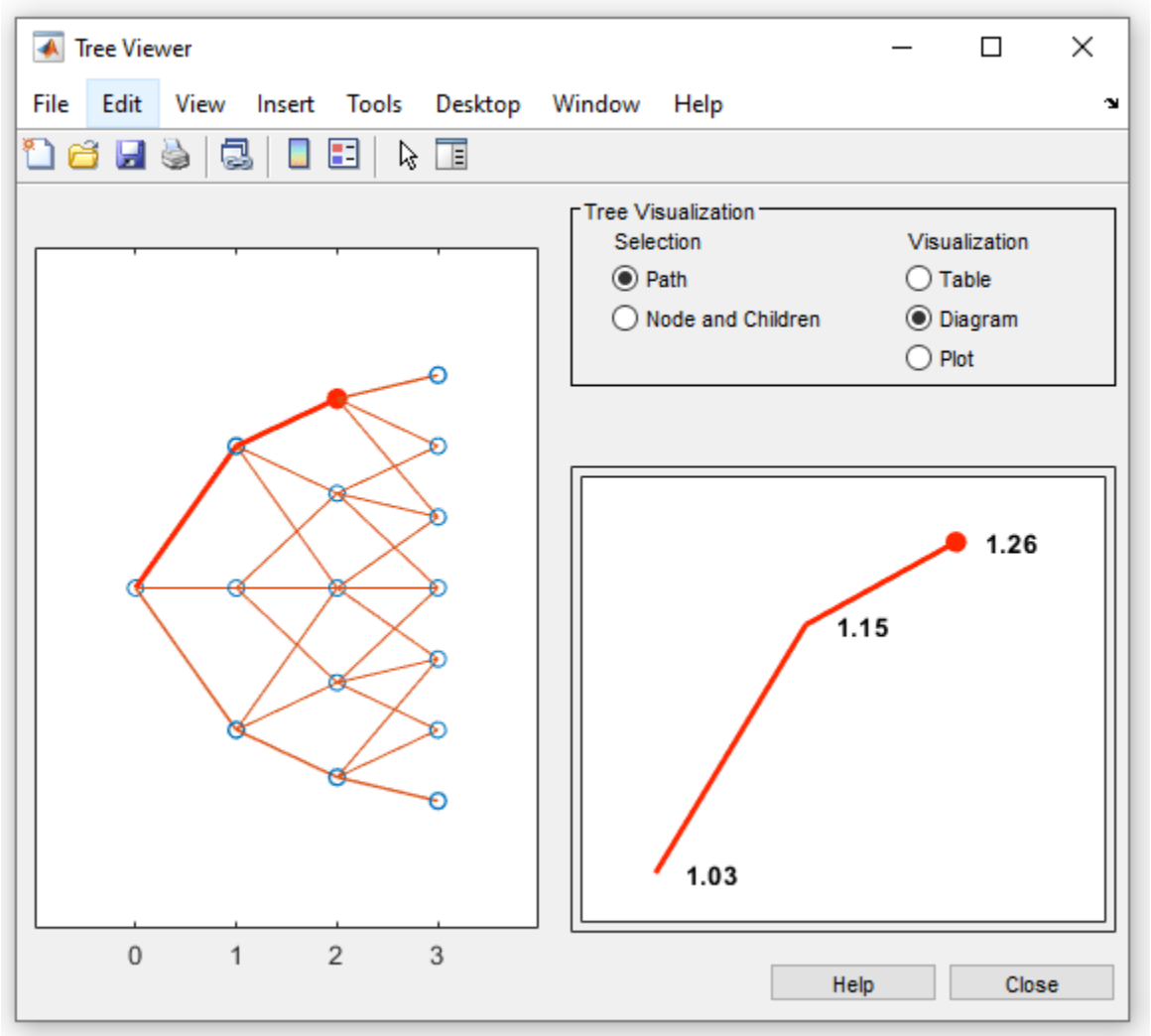
1. In the **Tree Visualization** pane, click **Path** and **Diagram**.

2. Select the first path by clicking the first node of the up branch at $t = 1$.

3. Continue by clicking the up branch at the next node at $t = 2$.

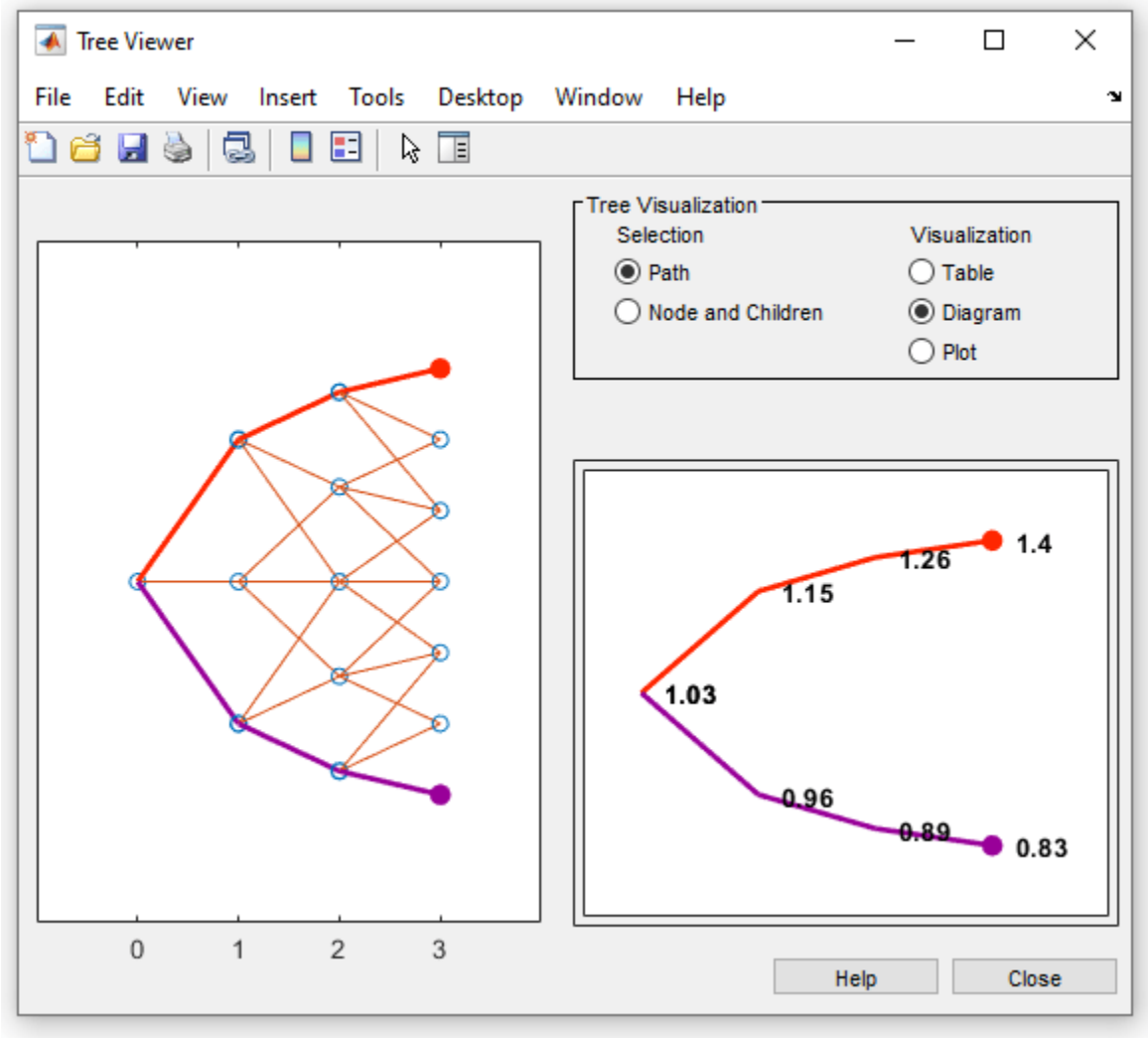
The following figures show the `treeviewer` path diagrams for these selections.


Tree Viewer
File Edit View Insert Tools Desktop Window Help
Tree Visualization
Selection
Path
Node and Children
Visualization
Table
Diagram
Plot
1.15
1.03
0 1 2 3
Help
Close



4. Continue clicking all nodes in succession until you reach the end of the branch. The entire path you selected is highlighted in red.

5. Select a second path by clicking the first node of the lower branch at $t = 1$. Continue clicking lower nodes as you did on the first branch. The second branch is highlighted in purple.



Hull-White trees have two additional properties called `Probs` and `Connect`. The `Probs` property represents the transition probabilities and the `Connect` property defines how the nodes are connected together.

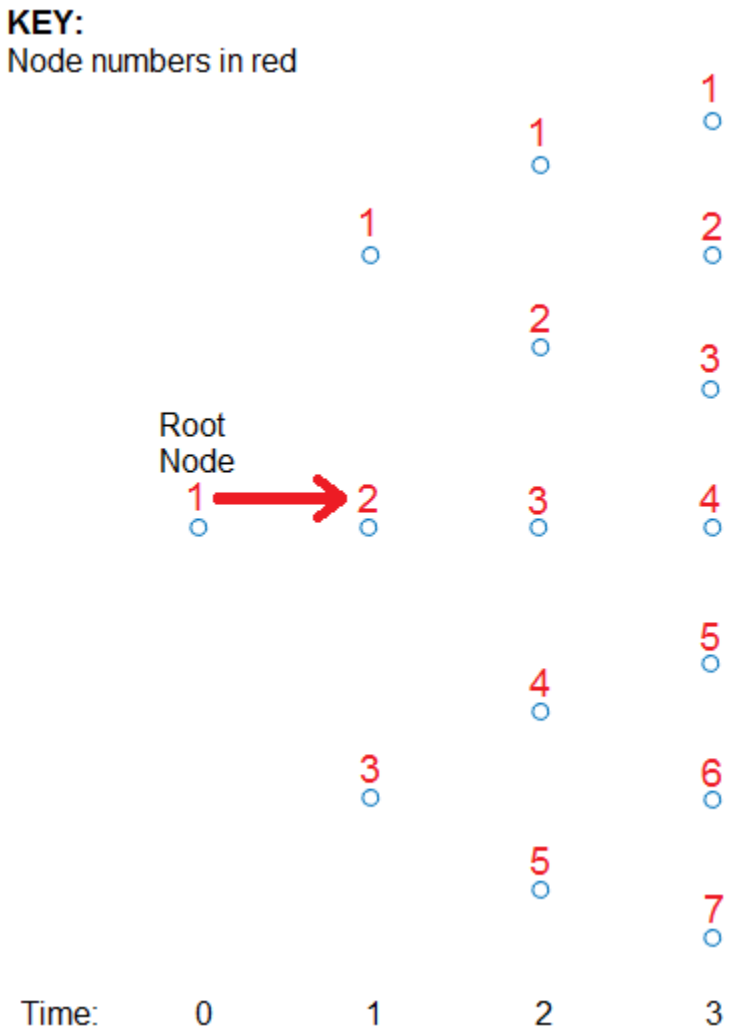
`Connect` Field

`HWT.Connect` describes the connectivity of the nodes of a given tree level to the nodes at the next tree level.

```
HWT.Connect
```

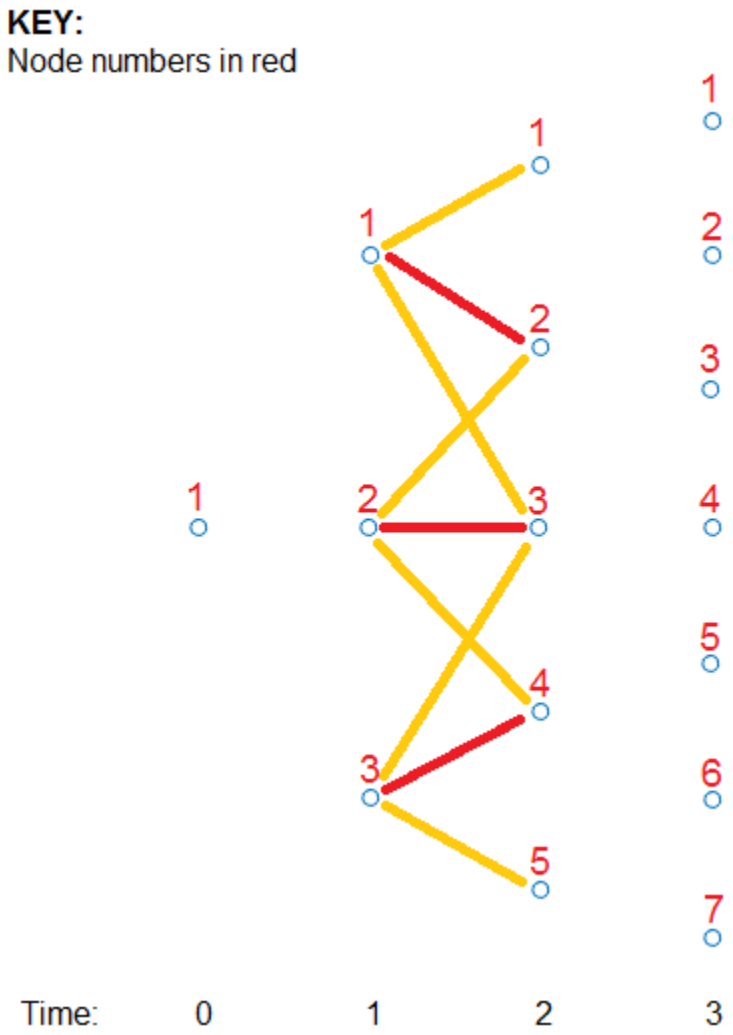
```
ans=1×3 cell array
    {[2]}    {[2 3 4]}    {[2 3 4 5 6]}
```

The first value of `HWT.Connect` corresponds to $t = 0$ for the root node and indicates that the root node connects to node 2 of the next level at $t = 1$. To visualize this, consider the following connection illustration of the tree, where the node numbers have been superimposed above each node.

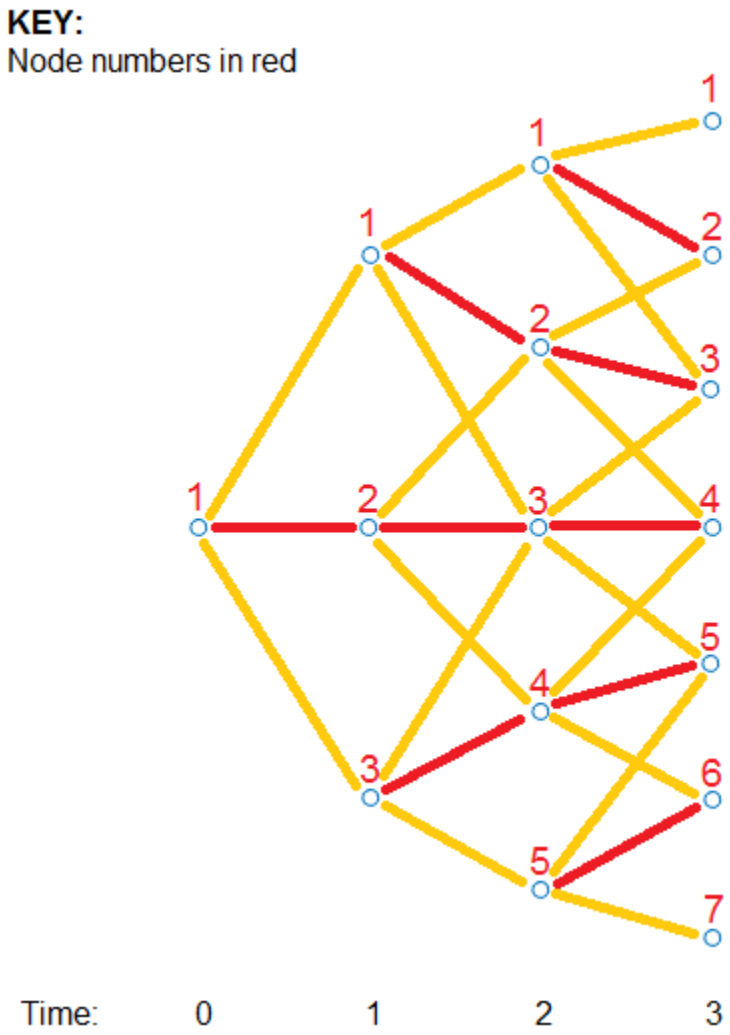


Specifically, `HWT.Connect` represents the index of the node in the next tree level ($t + 1$) that the middle branch of the node connects to.

The next entry in `HWT.Connect` at $t = 1$ is `[2,3,4]`. This means that node 1 at $t = 1$ has a middle branch with node 2 at $t = 2$, node 2 at $t = 1$ has a middle branch with node 3 at $t = 2$, and node 3 at $t = 1$ has a middle branch with node 4 at $t = 2$. A graphic representation follows.



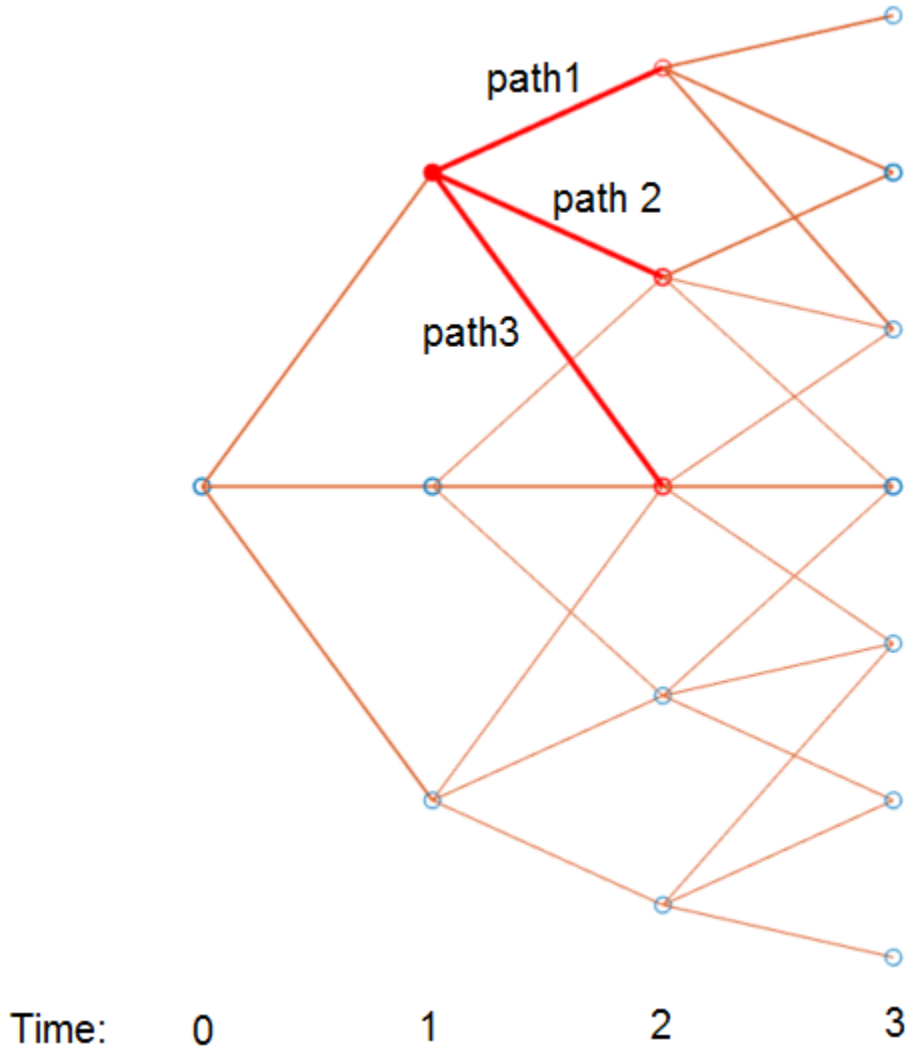
The middle branch path, as explicitly defined in `HWT.Connect`, is colored red and the implicit upper and lower branch paths are colored yellow. Overlaying all paths defined in `HWT.Connect` as red and the implicit upper and lower branches as yellow produces the following tree structure.



The shape in the figure is the same shape obtained by running the function `treeviewer(HWT)`.

`Probs` Field

Using the following illustration, consider that you want to know the probability at $t = 1$ (second level of the tree) that the top node takes of one of the three paths.



`HWT.Probs` gives the probabilities that a particular branch has of moving from a given node to a node on the next level of the tree.

```
HWT.Probs
```

```
ans=1×3 cell array
    {3x1 double}    {3x3 double}    {3x5 double}
```

The `Probs` field consists of a cell array with one cell per level of the tree. Find the probabilities of all three nodes at $t = 1$, which corresponds to the second level of the tree.

```
HWT.Probs{2}
```

```
ans = 3×3

    0.1429    0.1667    0.1929
    0.6642    0.6667    0.6642
```

```
    0.1929    0.1667    0.1429
```

Each column represents a different node. The first node at $t = 1$ corresponds to the first column and the probabilities are 14.29%, 66.42%, and 19.30%.

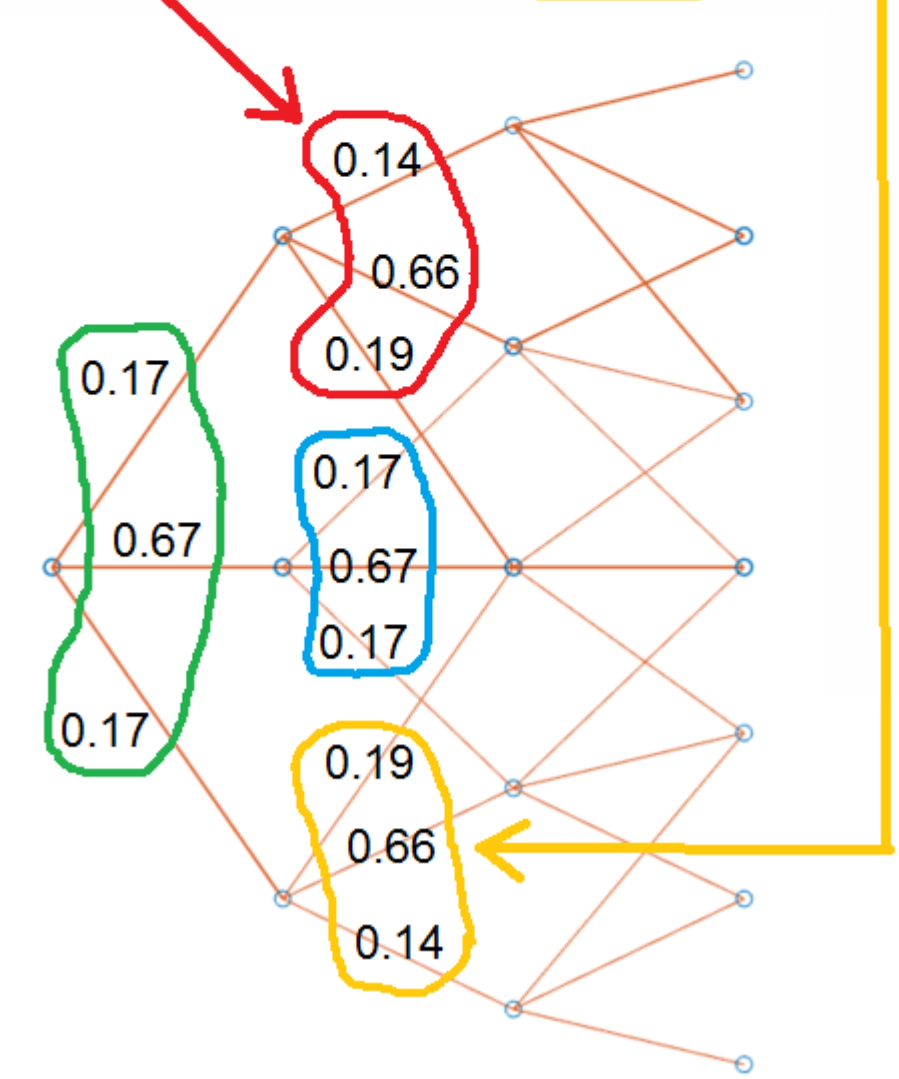
The probability of moving up (path 1) is the top value (14.29%), the middle path is the middle value (66.42%), and the path going down (path 3) is the bottom value in the array (19.30%). The following diagram summarizes this information.

```
HWT.Probs{1}
```

```
ans = 3×1
    0.16667
    0.66667
    0.16667
```

```
HWT.Probs{2}
```

```
ans = 3×3
    0.14292    0.16667    0.19292
    0.66417    0.66667    0.66417
    0.19292    0.16667    0.14292
```



Examine the PriceTree Structure

The output of the pricing function is `PriceTree`. `PriceTree` has the following fields.

- `PriceTree.PTree` — contains the clean prices of each instrument.
- `PriceTree.ExTree` — contains the exercise indicator arrays where a value of `1` indicates the option has been exercised and a value of `0` indicates the option has not been exercised.
- `PriceTree.ExProbTree` — contains the exercise probabilities. A value of 0 indicates there was no exercise and a nonzero value gives the probability of reaching that node where the exercise happens.
- `PriceTree.ProbTree` — contains the probability tree indicating how likely any node is reached from the root node.
- `PriceTree.ExProbsByTreeLevel` — contains the exercise probability for a given option at each tree observation time. This is an aggregated view of `PriceTree.ExProbTree` that sums the values along all nodes at a particular time.

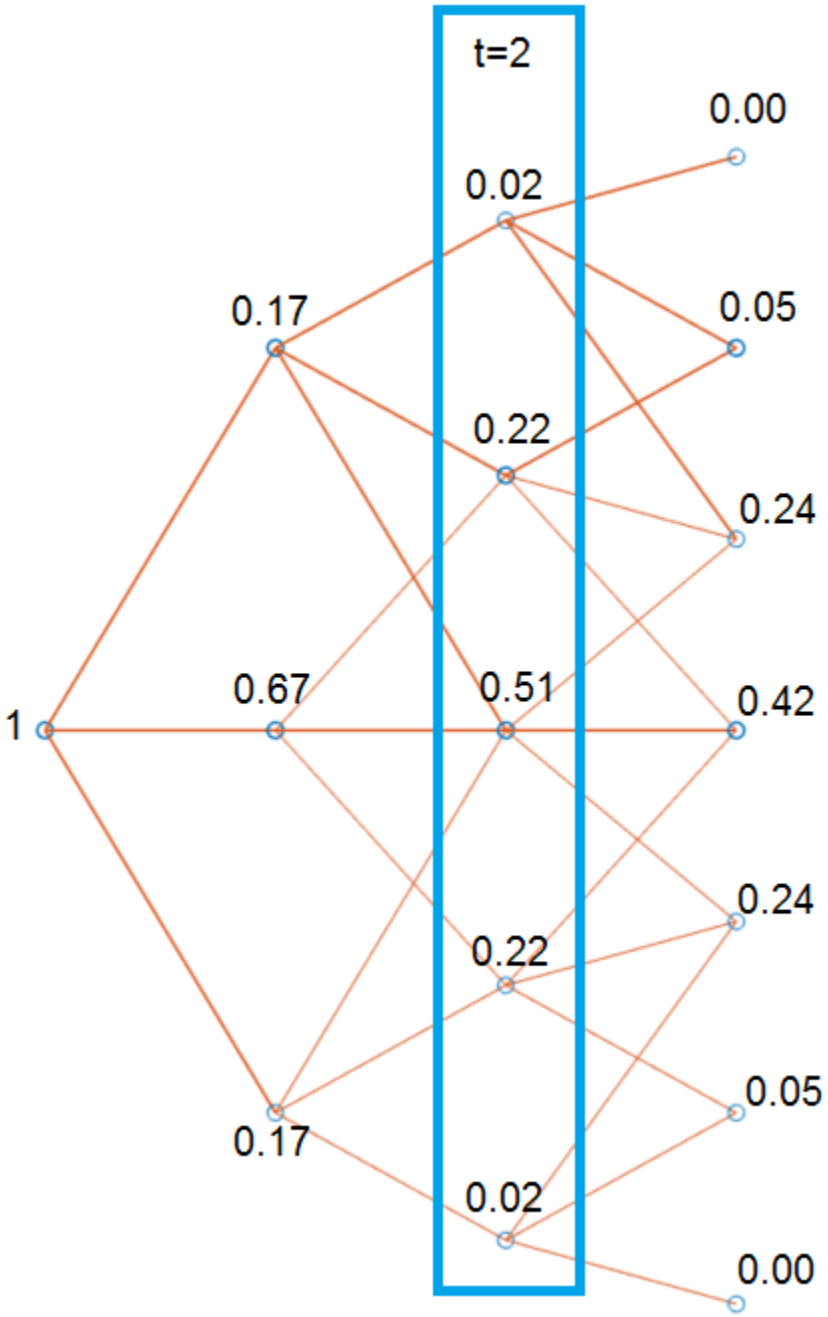
Note that for `ProbTree`, `PTree`, `ExTree`, and `ExProbTree`, each cell represents a different time in the tree, and inside each cell is an array. Each column in the array represents a different node on the tree at that particular tree level. This structure is the same as in `HWT.Probs`. However, for `PTree`, `ExTree`, and `ExProbTree` each row represents a different instrument. Because this example prices two instruments, there are only two rows. `ProbTree` contains only one row as the probability of reaching a particular node is independent of the instrument being priced.

Looking at `PriceTree.ProbTree`, examine the probabilities of reaching each of the five nodes from the root node at $t = 2$, which is the third level of the tree.

```
PriceTree.ProbTree{3}
```

```
ans = 1×5

    0.0238    0.2218    0.5087    0.2218    0.0238
```

These results are displayed in the following diagram, where all nodes are overlayed with their probabilities. The root node at $t = 0$ always has a probability of being reached, hence, it has a value of `1`.



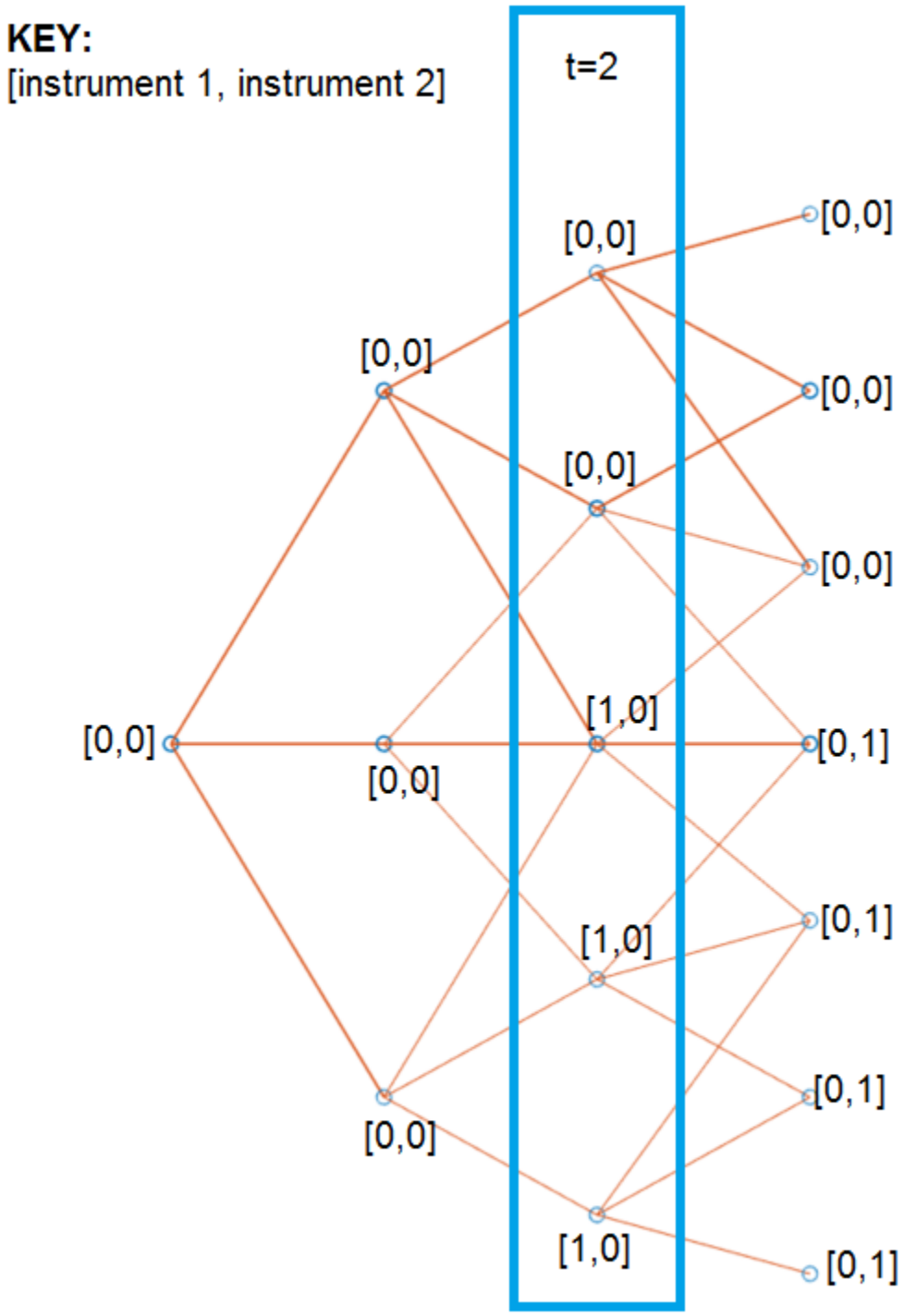
By looking at `PriceTree.ExTree`, you can determine if the options are exercised. If either of the two instruments has options exercised at $t = 2$, which is the third level of the tree, the values in `ExTree` are `1`; otherwise, the value is `0`.

```
PriceTree.ExTree{3}
```

```
ans = 2×5
```

```
     0     0     1     1     1
     0     0     0     0     0
```

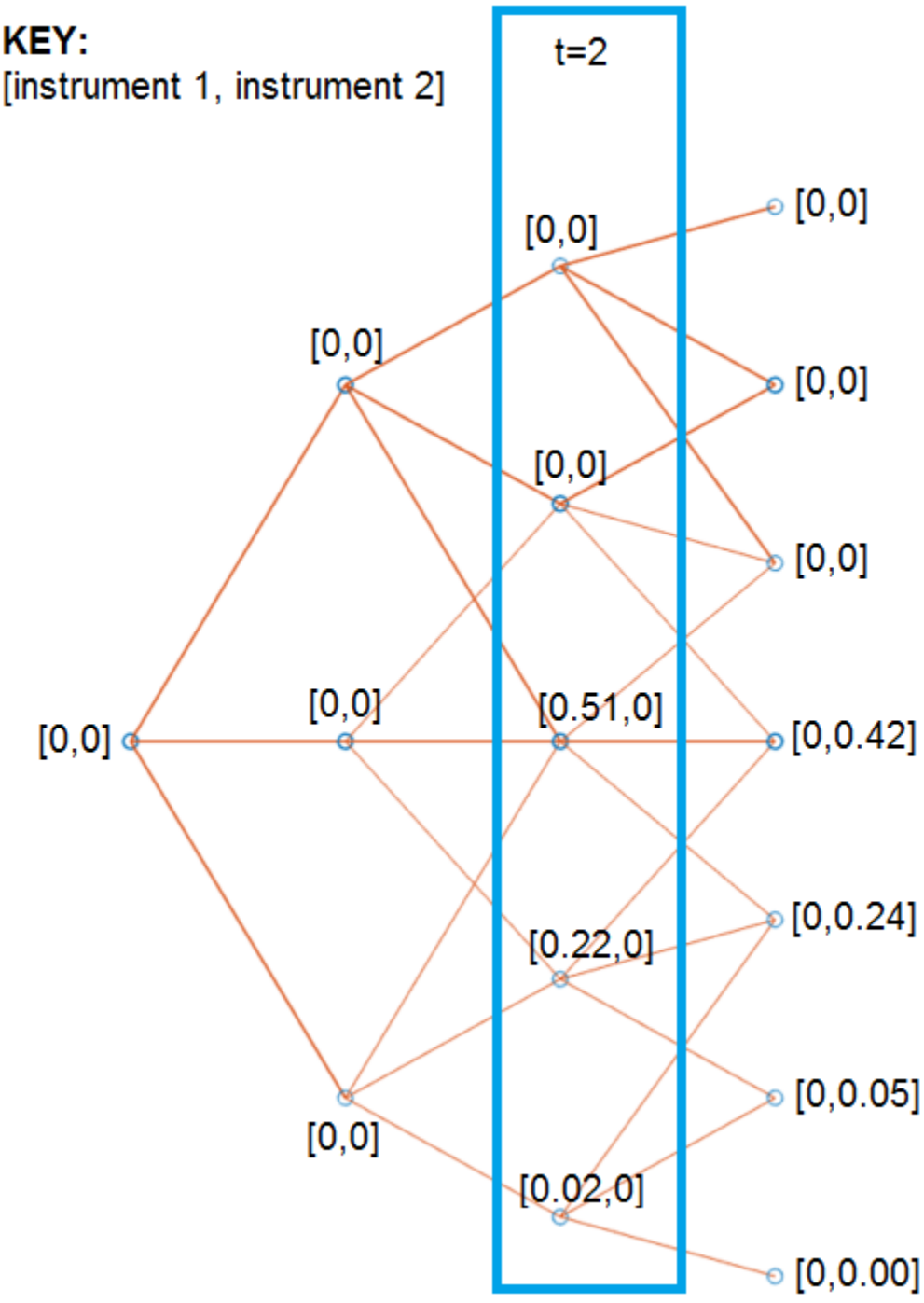
At $t = 2$, the first instrument has its option exercised at some nodes, while there is no exercise for the second instrument. The following diagram summarizes the exercise indicators on the tree.



Examine `ExProbTree`, which contains the exercise probabilities. These values indicate the probability of exercising the option.

```
PriceTree.ExProbTree{3}
```

```
ans = 2×5

         0         0    0.5087    0.2218    0.0238
         0         0         0         0         0
```



`ExProbsByTreeLevel` is an aggregated view of `ExProbTrees`. Examine the exercise probabilities for the two options at each tree observation time.

```
PriceTree.ExProbsByTreeLevel
```

```
ans = 2×5

         0         0    0.7544         0         0
         0         0         0    0.7124         0
```

The first row corresponds to instrument 1, and the second row corresponds to instrument 2.

You can use `treeviewer` to examine the `tree` and `PriceTree` structures for any of the following tree types:

- `bdttree`
- `bktree`
- `hjmtree`
- `hwtree`
- `cirtree`
- `crrtree`
- `itttree`
- `stttree`
- `lrtree`
- `eqptree`

Select Cheapest-to-Deliver Bond Using BondFuture Instrument

This example shows how to select the cheapest-to-deliver (CTD) bond for Treasury bond future contracts using a `BondFuture` instrument. It demonstrates two workflows for selecting the CTD bond:

- Select the CTD bond at the time of delivery on page 2-212.
- Select the CTD bond months before the delivery date on page 2-215.

When a Treasury bond future contract matures, the party that is in the short position of the contract can deliver the underlying bond to the party that is in the long position of the contract. Typically, there can be many different bonds that satisfy the terms of the contract during the delivery month. For example, you can meet the delivery obligation for a 30-year US Treasury bond future contract with any US Treasury bond that has at least 15 years to maturity at the time of delivery. Furthrermore, these deliverable bonds can have different coupons and maturities. Since the party in the short position has the option to choose which bond to deliver from these deliverable bonds, it is in the interest of that party to select the CTD bond that minimizes the delivery cost. You can select the CTD bond at the time of delivery during the delivery month of the future contract using the current market prices. Alternatively, you can predict the CTD bond several months before the delivery month based on the current zero curve.

Select CTD Bond at Time of Delivery

In this workflow, you determine the delivery cost by comparing the invoice price that the short position receives from the long position against the price of the deliverable bond. For a particular deliverable bond, the invoice price that the short position receives from the long position is:

```
InvoicePrice = (QuotedFuturePrice x ConversionFactor + AccruedInterest)/BondPrincipal x FutureNo
```

For example, in the 30-year US Treasury bond futures market, typically, `BondPrincipal` is $100 and `FutureNotional` is $100,000. In return, the short position delivers the bond to the long position by buying the deliverable bond that has the following purchase cost in the market:

```
PurchaseCost = (SpotPrice + AccruedInterest)/BondPrincipal x FutureNotional
```

Here, `SpotPrice` is the clean market price of the bond at the time of delivery. From this purchase cost, subtracting the invoice amount to be received by the short position from the long position gives the following formula for the overall delivery cost:

```
OverallDeliveryCost = (SpotPrice - QuotedFuturePrice x ConversionFactor)/BondPrincipal x FutureN
```

You can compute the delivery cost for every deliverable bond and determine the CTD bond by selecting the bond with the lowest delivery cost. The `OverallDeliveryCost` formula applies only at the time of delivery. This method is equivalent to using the `cashsettle` method of the `BondFuture` instrument at the time of delivery. To demonstrate this workflow, the following example uses fictitious data, and selects the CTD bond using the `BondFuture` instrument on the delivery day of a hypothetical 30-year US Treasury bond future contract maturing in December 2021.

```
% Create a zero curve.
Settle = datetime(2021,12,21);
ZeroDates = Settle + [calmonths([1 2 3 6]) calyears([1 2 3 5 7 10 20 30])]';
ZeroRates = [0.06 0.04 0.06 0.07 0.21 0.60 0.95 1.34 1.60 1.69 2.15 2.05]'./100;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

```
% Define the data for the deliverable bonds.
ReferenceDate = datetime(2021,12,1);
BondMaturity = datetime(datevec(["2/15/2045","2/15/2037","8/15/2040",...
    "5/15/2041","5/15/2037","2/15/2042","11/15/2043","11/15/2044",...
    "5/15/2038","2/15/2041"],'mm/dd/yyyy'));
CouponRate = [2.5 4.75 3.875 2.25 5.0 3.125 3.75 3.0 4.5 4.75]'/100;
IssueDate = datetime(datevec(["2/17/2015","2/15/2007","8/16/2010",...
    "6/1/2021","8/15/2007","2/15/2012","11/15/2013","11/17/2014",...
    "8/15/2008","2/15/2011"],'mm/dd/yyyy'));
SpotPrice = [109.06;140.78;130.60;104.04;144.02;119.03;131.05;118.23;138.08;145.25];
NumBonds = length(BondMaturity);
BondID = (1:NumBonds)';

% Use convfactor to compute the conversion factors.
ConversionFactor = convfactor(ReferenceDate,BondMaturity,CouponRate)

ConversionFactor = 10×1

    0.5664
    0.8775
    0.7645
    0.5752
    0.9009
    0.6677
    0.7286
    0.6302
    0.8456
    0.8594


% Create a vector of FixedBond instruments.
BondPrincipal = 100;
DeliverableBonds = fininstrument("FixedBond",Maturity=BondMaturity,...
    CouponRate=CouponRate,IssueDate=IssueDate,Principal=BondPrincipal)

DeliverableBonds=10×1 object
  10x1 FixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    Name


% Create a vector of BondFuture instruments.
FutureMaturity = datetime(2021,12,21);
QuotedFuturePrice = 159.53;
FutureNotional = 100000;
```

```
BondFutureContracts = fininstrument("BondFuture",Maturity=FutureMaturity,...
    QuotedPrice=QuotedFuturePrice,Bond=DeliverableBonds,...
    ConversionFactor=ConversionFactor,Notional=FutureNotional)

BondFutureContracts=10×1 object
  10x1 BondFuture array with properties:

    Maturity
    QuotedPrice
    Bond
    ConversionFactor
    Notional
    Name
```

To compute the delivery cost, use the `cashsettle` method of the `BondFuture` instrument to compute the delivery cost because this method estimates the undiscounted net cash settlement amount that could be paid by the short position to the long position at future maturity instead of physical delivery.

```
% Use the cashsettle method to compute the delivery costs.
DeliveryCost = nan(NumBonds,1);
for k=1:NumBonds
    outCS = cashsettle(BondFutureContracts(k), SpotPrice(k), ZeroCurve);
    DeliveryCost(k) = outCS.CashSettleAmount;
end

% List the deliverable bonds and delivery costs in a table.
DeliverableBondTable = table(BondID, BondMaturity, IssueDate, ...
    CouponRate, SpotPrice, ConversionFactor, DeliveryCost)
```

```
DeliverableBondTable=10×7 table
    BondID    BondMaturity     IssueDate     CouponRate    SpotPrice    ConversionFactor    Deliv
    ______    ____________    ___________    __________    _________    ________________    _____

       1      15-Feb-2045     17-Feb-2015       0.025       109.06          0.56643             1
       2      15-Feb-2037     15-Feb-2007      0.0475       140.78           0.8775            79
       3      15-Aug-2040     16-Aug-2010     0.03875        130.6          0.76447            80
       4      15-May-2041     01-Jun-2021      0.0225       104.04          0.57524             1
       5      15-May-2037     15-Aug-2007        0.05       144.02           0.9009            29
       6      15-Feb-2042     15-Feb-2012     0.03125       119.03          0.66773             1
       7      15-Nov-2043     15-Nov-2013      0.0375       131.05          0.72859             1
       8      15-Nov-2044     17-Nov-2014        0.03       118.23          0.63022             1
       9      15-May-2038     15-Aug-2008       0.045       138.08          0.84558            31
      10      15-Feb-2041     15-Feb-2011      0.0475       145.25          0.85942            81
```

Once you have computed the delivery costs for all of the deliverable bonds, you can determine the CTD bond by selecting the bond with the lowest delivery cost.

```
% Determine the CTD bond with the lowest delivery cost.
[~,CTDBondIdx] = min(DeliverableBondTable.DeliveryCost);
CTDBondTableDecember2021 = DeliverableBondTable(CTDBondIdx,:)
```

```
CTDBondTableDecember2021=1×7 table
    BondID    BondMaturity     IssueDate     CouponRate    SpotPrice    ConversionFactor    Deliv
    ______    ____________    ___________    __________    _________    ________________    _____
```

```
      5       15-May-2037     15-Aug-2007       0.05        144.02          0.9009            29
```

```
BondFutureContracts(CTDBondIdx)
```

```
ans =
  BondFuture with properties:

           Maturity: 21-Dec-2021
       QuotedPrice: 159.5300
              Bond: [1x1 fininstrument.FixedBond]
    ConversionFactor: 0.9009
           Notional: 100000
               Name: ""
```

Among the ten deliverable bonds, the CTD bond with the lowest delivery cost for the December 2021 future contract is the bond with a 5% coupon rate maturing on May 15, 2037. This CTD bond is determined at the time of delivery on December 21, 2021.

Since the delivery costs in this example are computed at the time of delivery, the delivery costs computed by the `cashsettle` method of the `BondFuture` instrument for this future contract are in agreement with those computed using the following formula for the overall delivery cost at delivery:

```
OverallDeliveryCost = (SpotPrice - QuotedFuturePrice x ConversionFactor)/BondPrincipal x FutureNo
```

```
% Compare calculated delivery cost using cashsettle with the delivery cost formula.
DeliveryCostFormula = ...
    (SpotPrice - QuotedFuturePrice.*ConversionFactor)/BondPrincipal*FutureNotional;
table(DeliveryCostFormula, DeliveryCost)
```

```
ans=10×2 table
    DeliveryCostFormula    DeliveryCost
    ___________________    ____________

           18697               18697
          792.87              792.87
          8643.6              8643.6
           12272               12272
          299.73              299.73
           12508               12508
           14818               14818
           17690               17690
          3185.1              3185.1
          8146.4              8146.4
```

Select CTD Bond Months Before Delivery Month

The previous overall delivery cost formula on page 2-215 applies only at the time of delivery. However, if a current zero curve is available, you can still compute the estimated delivery cost using the `cashsettle` method of the `BondFuture` instrument, even when it is several months before the delivery month. In the following example, you select the CTD bond using the `BondFuture` instrument in November 2021 for a hypothetical 30-year US Treasury bond future contract maturing several months later in June 2022. The overall workflow is similar to the workflow in Select CTD Bond at Time of Delivery on page 2-212, except that you select the CTD bond several months before the delivery month and the overall delivery cost formula no longer applies.

```
% Create a zero curve.
Settle = datetime(2021,11,22);
ZeroDates = Settle + [calmonths([1 2 3 6]) calyears([1 2 3 5 7 10 20 30])]';
ZeroRates = [0.07 0.04 0.05 0.07 0.20 0.63 0.95 1.34 1.57 1.65 2.09 2.01]'./100;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);

% Define the data for the deliverable bonds.
ReferenceDate = datetime(2022,6,1);
BondMaturity = datetime(datevec(["11/15/2046","2/15/2038","11/15/2040",...
    "11/15/2041","5/15/2044","5/15/2042","5/15/2045","8/15/2041",...
    "8/15/2043","8/15/2039"],'mm/dd/yyyy'));
CouponRate = [2.875 4.375 4.25 3.125 3.375 3.0 3.0 1.75 3.625 4.5]'/100;
IssueDate = datetime(datevec(["11/15/2016","2/15/2008","11/15/2010",...
    " 11/15/2011","5/15/2014","5/15/2012","5/15/2015","8/31/2021",...
    "8/15/2013","8/17/2009"],'mm/dd/yyyy'));
SpotPrice = [117.84;137.13;137.53;119.17;125.03;117.34;119.03;95.97;128.84;140.56];
NumBonds = length(BondMaturity);
BondID = 10+(1:NumBonds)';

% Use convfactor to compute the conversion factors.
ConversionFactor = convfactor(ReferenceDate,BondMaturity,CouponRate)
```

```
ConversionFactor = 10×1

    0.6033
    0.8375
    0.8074
    0.6743
    0.6834
    0.6555
    0.6302
    0.5220
    0.7185
    0.8415
```

```
% Create a vector of FixedBond instruments.
BondPrincipal = 100;
DeliverableBonds = fininstrument("FixedBond",Maturity=BondMaturity, ...
    CouponRate=CouponRate,IssueDate=IssueDate,Principal=BondPrincipal)
```

```
DeliverableBonds=10×1 object
  10x1 FixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    Name
```

```
% Create a vector of BondFuture instruments.
FutureMaturity = datetime(2022,6,21);
QuotedFuturePrice = 160.31;
FutureNotional = 100000;
BondFutureContracts = fininstrument("BondFuture",Maturity=FutureMaturity, ...
    QuotedPrice=QuotedFuturePrice,Bond=DeliverableBonds, ...
    ConversionFactor=ConversionFactor,Notional=FutureNotional)
```

```
BondFutureContracts=10×1 object
  10x1 BondFuture array with properties:

    Maturity
    QuotedPrice
    Bond
    ConversionFactor
    Notional
    Name
```

```
% Use the cashsettle method to compute the delivery costs.
DeliveryCost = nan(NumBonds,1);
for k=1:NumBonds
    outCS = cashsettle(BondFutureContracts(k), SpotPrice(k), ZeroCurve);
    DeliveryCost(k) = outCS.CashSettleAmount;
end

% List the deliverable bonds and delivery costs in a table.
DeliverableBondTable = table(BondID, BondMaturity, IssueDate, ...
    CouponRate, SpotPrice, ConversionFactor, DeliveryCost)
```

```
DeliverableBondTable=10×7 table
    BondID    BondMaturity     IssueDate     CouponRate    SpotPrice    ConversionFactor    Deliv
    ______    ____________    ___________    __________    _________    ________________    _____

      11      15-Nov-2046     15-Nov-2016     0.02875       117.84          0.60331           1
      12      15-Feb-2038     15-Feb-2008     0.04375       137.13           0.8375           40
      13      15-Nov-2040     15-Nov-2010      0.0425       137.53          0.80741           56
      14      15-Nov-2041     15-Nov-2011     0.03125       119.17          0.67433           93
      15      15-May-2044     15-May-2014     0.03375       125.03          0.68337            1
      16      15-May-2042     15-May-2012        0.03       117.34          0.65551            1
      17      15-May-2045     15-May-2015        0.03       119.03          0.63022            1
      18      15-Aug-2041     31-Aug-2021      0.0175        95.97          0.52204            1
      19      15-Aug-2043     15-Aug-2013     0.03625       128.84          0.71855            1
      20      15-Aug-2039     17-Aug-2009       0.045       140.56          0.84151           31
```

```
% Determine the CTD bond with the lowest delivery cost.
[~,CTDBondIdx] = min(DeliverableBondTable.DeliveryCost);
CTDBondTableJune2022 = DeliverableBondTable(CTDBondIdx,:)
```

```
CTDBondTableJune2022=1×7 table
    BondID    BondMaturity     IssueDate     CouponRate    SpotPrice    ConversionFactor    Deliv
    ______    ____________    ___________    __________    _________    ________________    _____

      12      15-Feb-2038     15-Feb-2008     0.04375       137.13           0.8375           40
```

```
BondFutureContracts(CTDBondIdx)
```

```
ans =
  BondFuture with properties:

           Maturity: 21-Jun-2022
       QuotedPrice: 160.3100
              Bond: [1x1 fininstrument.FixedBond]
   ConversionFactor: 0.8375
           Notional: 100000
               Name: ""
```

Among the ten deliverable bonds, the CTD bond with the lowest delivery cost for the June 2022 future contract is the bond with a 4.375% coupon rate maturing on February 15, 2038. This CTD bond is predicted using the zero curve available on November 22, 2021, which is several months before the delivery month in June 2022.

See Also

Functions
`finmodel` | `finpricer` | `convfactor` | `BondFuture` | `ratecurve` | `cashsettle`

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
```

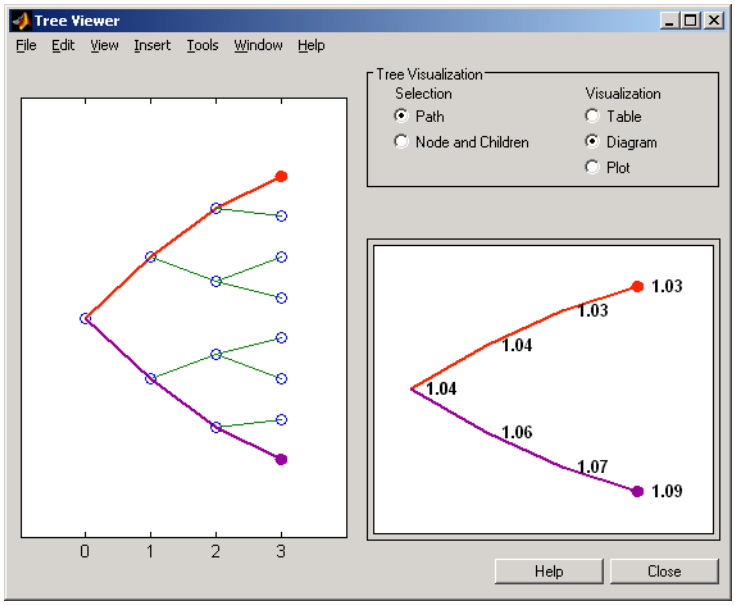
Note `treeviewer` price tree diagrams follow the convention that increasing prices appear on the upper branch of a tree and, consequently, decreasing prices appear on the lower branch. Conversely, for interest rate displays, *decreasing* interest rates appear on the upper branch (prices are rising) and *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 2-219. For information on using `treeviewer` to observe the movement of prices, see “Observing Instrument Prices” on page 2-222.

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)
```

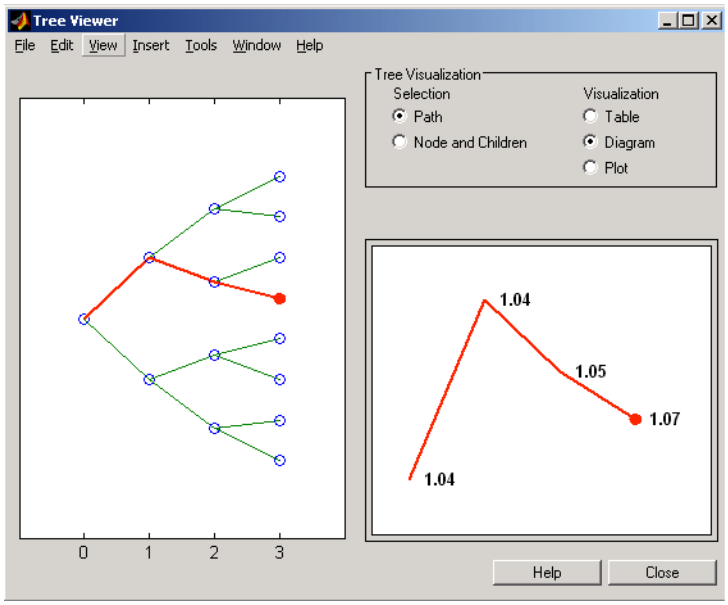


The example in “Isolating a Specific Node” on page 2-75 used `bushpath` to find the path of forward rates along an HJM tree by taking the first branch up and then two branches down the rate tree.

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

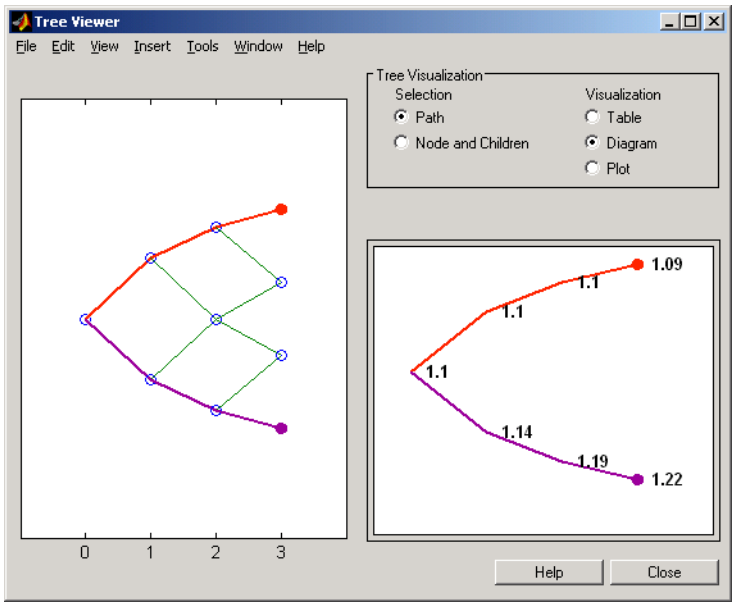
```
FRates =

    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)
```



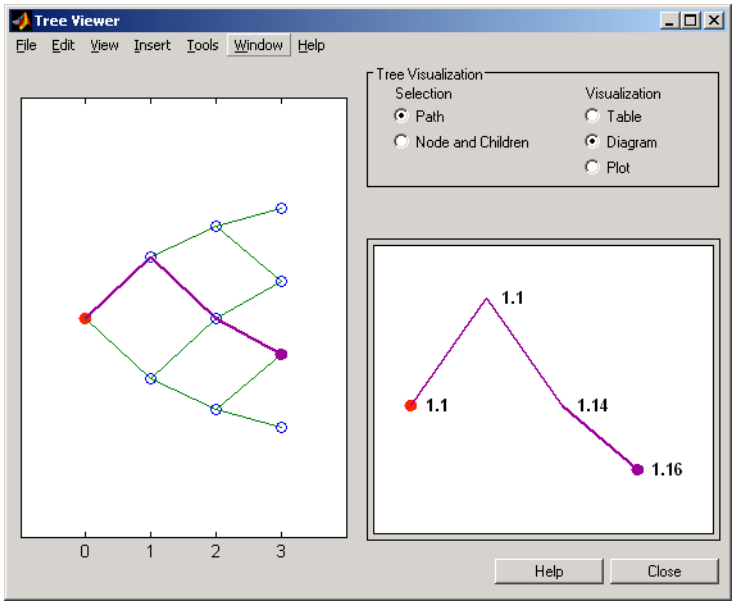
Note When using `treeviewer` with recombining trees, such as BDT, BK, and HW, you must click each node in succession from the beginning to the end. Because these trees can recombine, `treeviewer` is unable to complete the path automatically.

The example in “Isolating a Specific Node” on page 2-75 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.

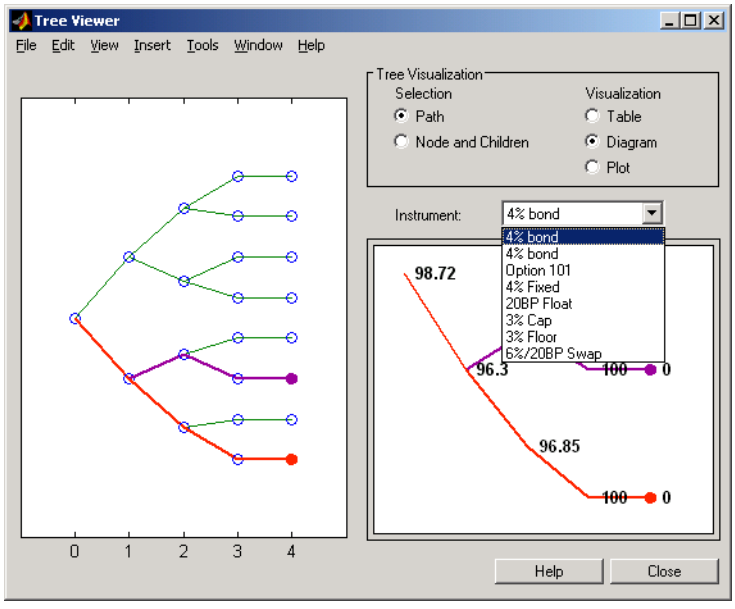


Observing Instrument Prices

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:

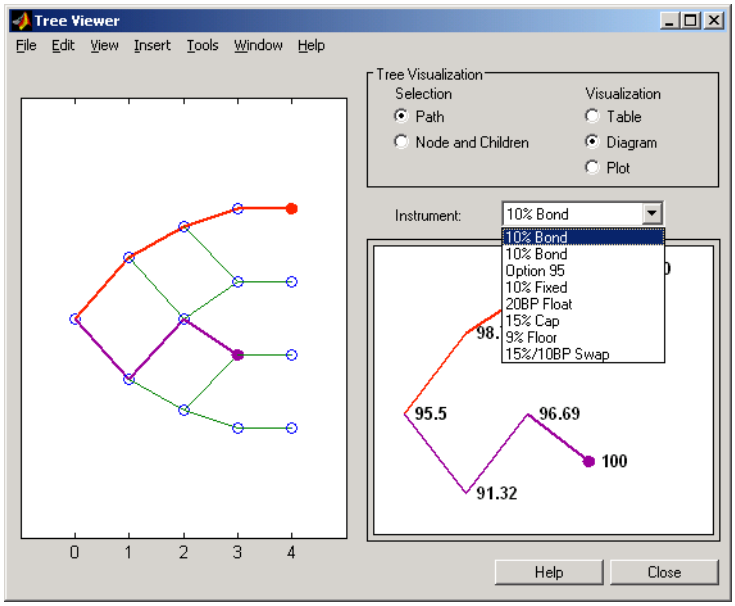
```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet);
treeviewer(PriceTree, HJMInstSet)
```

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



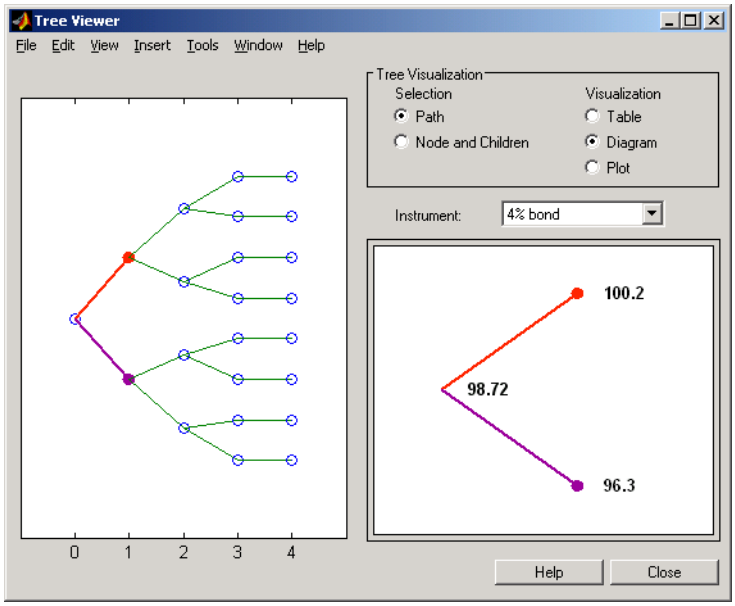
You can use an analogous process to view instrument prices based on the BDT interest rate tree included in `deriv.mat`.

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

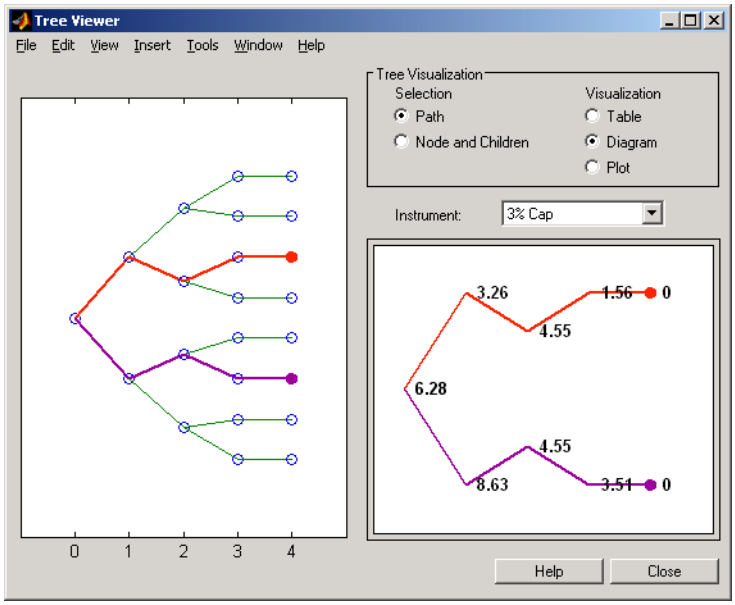


Valuation Date Prices

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

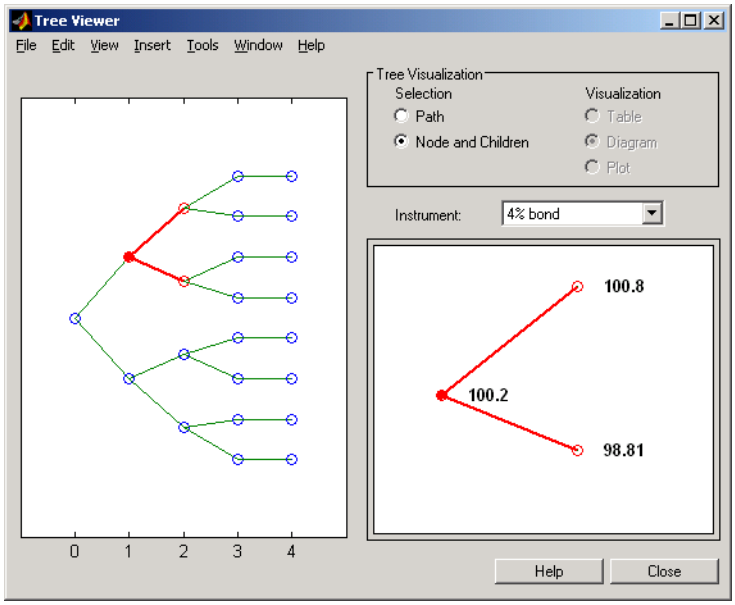
The second node represents the first-rate observation time, `tObs = 1`. This node displays two states, one representing the branch going up and the other one representing the branch going down.

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.

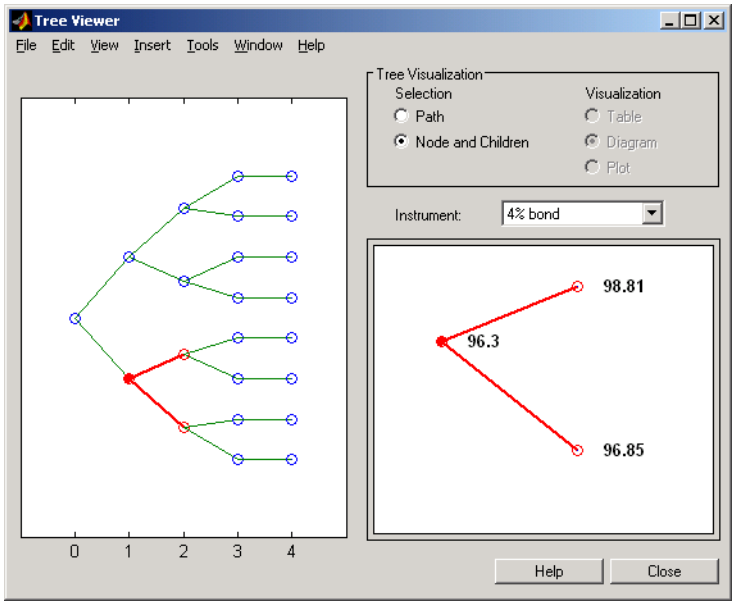


Now examine the corresponding down branch.

```
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.



See Also

instbond | instcap | instcf | instfixed | instfloat | instfloor | instoptbnd | instoptembnd | instoptfloat | instoptemfloat | instrangefloat | instswap | instswaption | intenvset | bondbyzero | cfbyzero | fixedbyzero | floatbyzero | intenvprice | intenvsens | swapbyzero | floatmargin | floatdiscmargin | hjmtimespec | hjmtree | hjmvolspec | bondbyhjm | capbyhjm | cfbyhjm | fixedbyhjm | floatbyhjm | floorbyhjm | hjmprice | hjmsens | mmktbyhjm | oasbyhjm | optbndbyhjm | optfloatbyhjm | optembndbyhjm | optemfloatbyhjm | rangefloatbyhjm | swapbyhjm | swaptionbyhjm | bdttimespec | bdttree | bdtvolspec | bdtprice | bdtsens | bondbybdt | capbybdt | cfbybdt | fixedbybdt | floatbybdt | floorbybdt | mmktbybdt | oasbybdt | optbndbybdt | optfloatbybdt | optembndbybdt | optemfloatbybdt | rangefloatbybdt | swapbybdt | swaptionbybdt | hwtimespec | hwtree | hwvolspec | bondbyhw | capbyhw | cfbyhw | fixedbyhw | floatbyhw | floorbyhw | hwcalbycap | hwcalbyfloor | hwprice | hwsens | oasbyhw | optbndbyhw | optfloatbyhw | optembndbyhw | optemfloatbyhw | rangefloatbyhw | swapbyhw | swaptionbyhw | bktimespec | bktree | bkvolspec | bkprice | bksens | bondbybk | capbybk | cfbybk | fixedbybk | floatbybk | floorbybk | oasbybk | optbndbybk | optfloatbybk | optembndbybk | optemfloatbybk | rangefloatbybk | swapbybk | swaptionbybk | capbyblk | floorbyblk | swaptionbyblk

Related Examples

- “Overview of Interest-Rate Tree Models” on page 2-44
- “Pricing Using Interest-Rate Term Structure” on page 2-61
- “Pricing Using Interest-Rate Tree Models” on page 2-81
- “Understanding Interest-Rate Tree Models” on page 2-66
- “Understanding the Interest-Rate Term Structure” on page 2-48

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34

Basis

Day count basis determines how interest accrues over time for various instruments and the amount transferred on interest payment dates. The calculation of accrued interest for dates between payments also uses day count basis. Day count basis is a fraction of `Number of interest accrual days` / `Days in the relevant coupon period`. Supported day count conventions and basis values are:

Basis Value	Day Count Convention
`0`	actual/actual (default) — Number of days in both a period and a year is the actual number of days.
`1`	30/360 SIA — Year fraction is calculated based on a 360 day year with 30-day months, after applying the following rules: If the first date and the second date are the last day of February, the second date is changed to the 30th. If the first date falls on the 31st or is the last day of February, it is changed to the 30th. If after the preceding test, the first day is the 30th and the second day is the 31st, then the second day is changed to the 30th.
`2`	actual/360 — Number of days in a period is equal to the actual number of days, however the number of days in a year is 360.
`3`	actual/365 — Number of days in a period is equal to the actual number of days, however the number of days in a year is 365 (even in a leap year).
`4`	30/360 PSA — Number of days in every month is set to 30 (including February). If the start date of the period is either the 31st of a month or the last day of February, the start date is set to the 30th, while if the start date is the 30th of a month and the end date is the 31st, the end date is set to the 30th. The number of days in a year is 360.
`5`	30/360 ISDA — Number of days in every month is set to 30, except for February where it is the actual number of days. If the start date of the period is the 31st of a month, the start date is set to the 30th while if the start date is the 30th of a month and the end date is the 31st, the end date is set to the 30th. The number of days in a year is 360.
`6`	30E /360 — Number of days in every month is set to 30 except for February where it is equal to the actual number of days. If the start date or the end date of the period is the 31st of a month, that date is set to the 30th. The number of days in a year is 360.
`7`	actual/365 Japanese — Number of days in a period is equal to the actual number of days, except for leap days (29th February) which are ignored. The number of days in a year is 365 (even in a leap year).
`8`	actual/actual ICMA — Number of days in both a period and a year is the actual number of days and the compounding frequency is annual.
`9`	actual/360 ICMA — Number of days in a period is equal to the actual number of days, however the number of days in a year is 360 and the compounding frequency is annual.
`10`	actual/365 ICMA — Number of days in a period is equal to the actual number of days, however the number of days in a year is 365 (even in a leap year) and the compounding frequency is annual.
`11`	30/360 ICMA — Number of days in every month is set to 30, except for February where it is equal to the actual number of days. If the start date or the end date of the period is the 31st of a month, that date is set to the 30th. The number of days in a year is 360 and the compounding frequency is annual.

Basis Value	Day Count Convention
12	actual/365 ISDA — The day count fraction is calculated using the following formula: `(Actual number of days in period that fall in a leap year / 366) + (Actual number of days in period that fall in a normal year / 365 ).`
13	bus/252 — The number of days in a period is equal to the actual number of business days. The number of business days in a year is 252.

Equity Derivatives

Understanding Equity Trees

In this section...

Introduction

Financial Instruments Toolbox supports five types of recombining tree models to represent the evolution of stock prices:

- Cox-Ross-Rubinstein (CRR) model
- Equal probabilities (EQP) model
- Leisen-Reimer (LR) model
- Implied trinomial tree (ITT) model
- Standard trinomial tree (STT) model

For a discussion of recombining trees, see “Rate and Price Trees” on page 2-45.

The CRR, EQP, LR, STT, 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 LR model is another discrete time model. It has the advantage of producing estimates close to the Black-Scholes model using only a few steps, while also minimizing the oscillation.

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.

The STT model is another discrete time model. It is considered to produce more accurate results than the binomial model when fewer time steps are modeled. The STT model is sometimes more stable and accurate than the binomial model when pricing exotic 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`, `eqptree`, and `lrtree` create CRR trees, EQP trees, and LR trees, respectively. These functions create an output tree structure along with information about the parameters used for creating the tree.

The functions `crrtree`, `eqptree`, and `lrtree` 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)
```

And, the calling syntax for `lrtree` is:

```
LRTree = lrtree(StockSpec, RateSpec, TimeSpec, Strike)
```

All three 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`, `eqptimespec`, and `lrtimespec`. 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 character vector 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, 2004, July 01, 2005, and January 01, 2006. 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. “Modeling the Interest-Rate Term Structure” on page 2-57 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)
```

```
TimeSpec = lrtimespec(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 datetime or a date character vector.
- `Maturity` is a scalar date marking the maturity of the tree, entered as a serial date number or a date character vector.
- `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 = datetime(2003,1,1);
Maturity = datetime(2008,1,1);
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`), a CRR tree (`CRRTree`), or an LR tree (`LRTree`). 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 = datetime(2003,1,1);
Maturity = datetime(2008,1,1);
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 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 character vector 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. “Modeling the Interest-Rate Term Structure” on page 2-57 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 datetime or a date character vector.
- `Maturity` is a scalar date marking the maturity of the tree, entered as a serial date number or a date character vector.
- `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 = datetime(2006,1,1);
EndDate = datetime(2008,1,1);
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 character vectors for the values `'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.

Sometimes, 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 3-7, “TimeSpec Example Using an

Implied Trinomial Tree” on page 3-8, and “Option Stock Structure Example Using an Implied Trinomial Tree” on page 3-9 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 = datetime(2006,1,1);
EndDate = datetime(2008,1,1);

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

```
RateSpec =

  struct with fields:

           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8521
            Rates: 0.0800
         EndTimes: 2
       StartTimes: 0
         EndDates: 733408
       StartDates: 732678
    ValuationDate: 732678
            Basis: 0
     EndMonthRule: 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]}
```

Building Standard 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 function `stttree` creates an output tree structure along with the information about the parameters used to create the tree.

The function `stttree` takes 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 Standard Trinomial Trees

The calling syntax for `stttree` is:

```
STTTree = stttree (StockSpec,RateSpec,TimeSpec)
```

- `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 `stttimespec`. 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.

Specifying the Stock Structure for Standard 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 character vector 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 Standard 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 Standard Trinomial Trees

The structure `RateSpec` defines the interest rate environment used when building the stock price binary tree. “Modeling the Interest-Rate Term Structure” on page 2-57 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 Standard 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 = stttimespec(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 datetime or a date character vector.
- `Maturity` is a scalar date marking the maturity of the tree, entered as a serial date number or a date character vector.
- `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 Standard Trinomial Tree

Consider building an STT 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 = datetime(2006,1,1);
EndDate = datetime(2008,1,1);
NumPeriods = 4;

TimeSpec = stttimespec(ValuationDate, EndDate, NumPeriods)
```

```
TimeSpec =

           FinObj: 'STTTimeSpec'
    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`).

Creating a Standard Trinomial Tree

You can now use the `StockSpec`, `TimeSpec` structures described in "Stock Structure Example Using an Implied Trinomial Tree" on page 3-7 and "TimeSpec Example Using an Implied Trinomial Tree" on page 3-8, to build a standard trinomial tree (STT). 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 = datetime(2006,1,1);
EndDate = datetime(2008,1,1);

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

```
RateSpec =

  struct with fields:

           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8521
            Rates: 0.0800
         EndTimes: 2
       StartTimes: 0
         EndDates: 733408
       StartDates: 732678
    ValuationDate: 732678
            Basis: 0
     EndMonthRule: 1
```

To build an `STTTree`, enter:

```
STTTree = sttree(StockSpec, RateSpec, TimeSpec)
```

```
STTTree =

       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.5000 1 1.5000 2]
         dObs: [732678 732860 733043 733225 733408]
        STree: {1x5 cell}
        Probs: {[3x1 double]  [3x3 double]  [3x5 double]  [3x7 double]}
```

Examining Equity Trees

Financial Instruments Toolbox uses equity binary and 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             27344  struct
  BDTTree           1x1              7322  struct
  BKInstSet         1x1             27334  struct
  BKTree            1x1              8532  struct
  CRRInstSet        1x1             21066  struct
  CRRTree           1x1              7086  struct
  EQPInstSet        1x1             21066  struct
  EQPTree           1x1              7086  struct
  HJMInstSet        1x1             27336  struct
  HJMTree           1x1              8334  struct
  HWInstSet         1x1             27334  struct
  HWTree            1x1              8532  struct
  ITTInstSet        1x1             21070  struct
  ITTTree           1x1             12660  struct
  STTInstSet        1x1             21070  struct
  STTTree           1x1              7782  struct
  ZeroInstSet       1x1             17458  struct
  ZeroRateSpec      1x1              2152  struct
```

Examining a CRRTree

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

```
CRRTree
```

```
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, that is, the times and dates of the levels of the tree. In this particular case, `tObs` reveals that the tree has a maturity of four 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 one 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 character vector representations.

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

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

Examining an ITTTree

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

```
ITTTree
```

```
ITTTree =

          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 and the times and dates of the levels of the tree. In this particular case, `tObs` reveals that the tree has a maturity of four 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 one 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 character vector 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 tapped 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 tapped 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 tapped 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 tapped 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.

See Also

`crrtree` | `eqptree` | `lrtree` | `stockspec` | `intenvset` | `crrtimespec` | `eqptimespec` | `lrtimespec` | `itttree` | `itttimespec` | `stockoptspec` | `treepath` | `trintreepath`

Related Examples

- “Pricing Equity Derivatives Using Trees” on page 3-64
- “Creating Instruments or Properties” on page 1-16
- “Graphical Representation of Equity Derivative Trees” on page 3-73

More About

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Supported Interest-Rate Instrument Functions” on page 2-3

- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Supported Equity Derivative Functions

In this section...

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

where:

S_{av} is the average price of underlying asset.

S is the price of the underlying asset.

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).
asianbystt	Price Asian options using a standard trinomial tree (STT).
instasian	Construct an Asian option.
asianbyls	Price European or American Asian options using the Longstaff-Schwartz model.
asiansensbyls	Calculate prices and sensitivities of European or American Asian options using the Longstaff-Schwartz model.
asianbykv	Price European geometric Asian options using the Kemna Vorst model.
asiansensbykv	Calculate prices and sensitivities of European geometric Asian options using the Kemna Vorst model.
asianbylevy	Price European arithmetic Asian options using the Levy model.
asiansensbylevy	Calculate prices and sensitivities of European arithmetic Asian options using the Levy model.
asianbyhhm	Calculate prices of European discrete arithmetic fixed Asian options using the Haug, Haug, Margrabe model.
asiansensbyhhm	Calculate prices and sensitivities of European discrete arithmetic fixed Asian options using the Haug, Haug, Margrabe model
asianbytw	Calculate prices of European arithmetic fixed Asian options using the Turnbull Wakeman model.
asiansensbytw	Calculate prices and sensitivities of European arithmetic fixed Asian options using the Turnbull Wakeman model.

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).
`barrierbystt`	Price barrier options using a standard trinomial tree (STT).
`barrierbyfd`	Price barrier option using finite difference method.
`barriersensbyfd`	Calculate barrier option price and sensitivities using finite difference method.
`barrierbybls`	Price a European barrier option using Black-Scholes option pricing model.
`barriersensbybls`	Calculate price and sensitivities for a European barrier option using Black-Scholes option pricing model.
`barrierbyls`	Price a barrier option using Longstaff-Schwartz model.
`barriersensbyls`	Calculate price and sensitivities for a barrier option using Longstaff-Schwartz model.
`instbarrier`	Construct a barrier option.

Double Barrier Option

A double barrier option is similar to the standard single barrier option except that they have two barrier levels: a lower barrier (LB) and an upper barrier (UB). The payoff for a double barrier option depends on whether the underlying asset remains between the barrier levels during the life of the option. Double barrier options are less expensive than single barrier options as the probability of being knocked out is higher. Because of this, double barrier options allow investors to achieve reduction in the option premiums as and match an investor’s belief about the future movement of the underlying price process.

There are two types of double barrier options:

- Double Knock-in

  This option becomes effective when the price of the underlying asset reaches one of the barriers. It gives the option holder, the right but not the obligation to buy or sell the underlying security at the strike price, if the underlying asset goes above or below the barrier levels during the life of the option.

- Double Knock-out

  This option gives the option holder, the right but not the obligation to buy or sell the underlying security at the strike price, as long as the underlying asset remains between the barrier levels during the life of the option. This option terminates when the price of the underlying asset passes one of the barriers.

The following functions support double barrier options.

Function	Purpose
`dblbarrierbybls`	Price European double barrier options using the Black-Scholes option pricing model.
`dblbarriersensbybls`	Calculate the price and sensitivities for a European double barrier options using the Black-Scholes option pricing model.
`dblbarrierbyfd`	Price double barrier option prices using the finite difference method.
`dblbarriersensbyfd`	Calculate the price and sensitivities for a double barrier option using the finite difference method.

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 Instruments 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.

Chooser Option

A chooser option enables the holder to decide before the option expiration date whether the option is a call or put.

A chooser option has a specified decision time t_1 where the holder has to make the decision whether the option is a call or put. At the expiration time t_2 the option expires. If the holder chooses a call option, the payout is $max\ (\ S - K\ ,\ 0)$. For the choice of a put option, the payout is $max\ (\ K - S\ ,\ 0)$ where K is the strike price of the option and S is the equity price at expiry.

Function	Purpose
`chooserbybls`	Price a European simple chooser options using Black-Scholes model.

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).
`compoundbystt`	Price compound options using a standard trinomial tree (STT).
`instcompound`	Construct a compound option.

Convertible Bond

A convertible bond is a financial instrument that combines equity and debt features. It is a bond with the embedded option to turn it into a fixed number of shares. The holder of a convertible bond has the right, but not the obligation, to exchange the convertible security for a predetermined number of equity shares at a preset price. The debt component is derived from the coupon payments and the principal. The equity component is provided by the conversion feature.

Convertible bonds have several defining features:

- Coupon — The coupon in convertible bonds are typically lower than coupons in vanilla bonds since investors are willing to take the lower coupon for the opportunity to participate in the company's stock via the conversion.
- Maturity — Most convertible bonds are issued with long-stated maturities. Short-term maturity convertible bonds usually do not have call or put provisions.
- Conversion ratio — Conversion ratio is the number of shares that the holder of the convertible bond will receive from exercising the call option of the convertible bond:

  ```
  Conversion ratio = par value convertible bond/conversion price of equity
  ```

  For example, a conversion ratio of 25 means a bond can be exchanged for 25 shares of stock. This also implies a conversion price of $40 (1000/25). This, $40, would be the price at which the owner would buy the shares. This can be expressed as a ratio or as the conversion price and is specified in the contract along with other provisions.
- Option type:
  - Callable Convertible: a convertible bond that is callable by the issuer. The issuer of the bond forces conversion, removing the advantage that conversion is at the discretion of the bondholder. Upon call, the bondholder can either convert the bond or redeem at the call price. This option enables the issuer to control the price of the convertible bond and if necessary refinance the debt with a new cheaper one.
  - Puttable Convertible: a convertible bond with a put feature that allows the bondholder to sell back the bond at a premium on a specific date. This option protects the holder against rising interest rates by reducing the year to maturity.

Function	Purpose
`cbondbycrr`	Price convertible bonds using a CRR binomial tree with the Tsiveriotis and Fernandes model.
`cbondbyeqp`	Price convertible bonds using an EQP binomial tree with the Tsiveriotis and Fernandes model.
`cbondbyitt`	Price convertible bonds using an implied trinomial tree with the Tsiveriotis and Fernandes model.
`cbondbystt`	Price convertible bonds using a standard trinomial tree with the Tsiveriotis and Fernandes model.
`instcbond`	Construct a `cbond` instrument for a convertible bond.

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 Instruments 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. So, 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).
`lookbackbystt`	Price lookback options using standard trinomial tree.
`instlookback`	Construct a lookback option based on an equity tree model.

Function	Purpose
`lookbackbycvgsg`	Calculate prices of European lookback fixed and floating strike options using the Conze-Viswanathan and Goldman-Sosin-Gatto models. For more information, see “Lookback Option” on page 3-39.
`lookbacksensbycvgsg`	Calculate prices and sensitivities of European fixed and floating strike lookback options using the Conze-Viswanathan and Goldman-Sosin-Gatto models. For more information, see “Lookback Option” on page 3-39.
`lookbackbyls`	Calculate prices of lookback fixed and floating strike options using the Longstaff-Schwartz model. For more information, see “Lookback Option” on page 3-39.
`lookbacksensbyls`	Calculate prices and sensitivities of lookback fixed and floating strike options using the Longstaff-Schwartz model. For more information, see “Lookback Option” on page 3-39.

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 when 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 Instruments 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 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.
- One-touch and double one-touch — (also known as binary barrier options or American digitals) are path-dependent options in which the existence and payment of the options depend on the movement of the underlying spot through their option life. For more information, see “One-Touch and Double One-Touch Options” on page 3-30.

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.

Function	Purpose
`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 Instruments 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 asset at time t.

K is the strike price.

The following functions support specifying or pricing a vanilla option.

Function	Purpose
`optstockbybaw`	Calculate the American options prices using the Barone-Adesi-Whaley option pricing model.
`optstocksensbybaw`	Calculate the American options prices and sensitivities using the Barone-Adesi-Whaley option pricing model.
`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.
`optstockbyfd`	Calculate vanilla option prices using finite difference method.
`optstocksensbyfd`	Calculate vanilla option prices and sensitivities using finite difference method.
`optByLocalVolFD`	Calculate vanilla option price by local volatility model, using finite differences.
`optSensByLocalVolFD`	Calculate vanilla option price or sensitivities by local volatility model, using finite differences.
`optByHestonFD`	Calculate vanilla option price by Heston model using finite differences.
`optSensByHestonFD`	Calculate vanilla option price and sensitivities by Heston model using finite differences.
`optByBatesFD`	Calculates vanilla European option price by Bates model using finite differences.
`optSensByBatesFD`	Calculates vanilla European option price and sensitivities by Bates model using finite differences.
`optByMertonFD`	Calculates vanilla European option price by Merton76 model using finite differences.
`optSensByMertonFD`	Calculates vanilla European option price and sensitivities by Merton76 model using finite differences.
`optstockbyitt`	Calculate the price of a European, Bermuda, or American stock option using an ITT tree.
`optstockbystt`	Calculate the price of a European, Bermuda, or American stock option using an STT tree.
`optstockbylr`	Calculate the price of a European, Bermuda, or American stock option using the Leisen-Reimer (LR) binomial tree model.

Function	Purpose
optstocksensbylr	Calculate the price and sensitivities of a European, Bermuda, or American stock option using the Leisen-Reimer (LR) binomial tree model.
optstockbybls	Price options using the Black-Scholes option pricing model.
optstocksensbybls	Calculate option prices and sensitivities using the Black-Scholes option pricing model.
optstockbyrgw	Calculate American call option prices using the Roll-Geske-Whaley option pricing model.
optstocksensbyrgw	Calculate American call option prices and sensitivities using the Roll-Geske-Whaley option pricing model.
optstockbybjs	Price American options using the Bjerksund-Stensland 2002 option pricing model.
optstocksensbybjs	Calculate American option prices and sensitivities using the Bjerksund-Stensland 2002 option pricing model.
optstockbyls	Price vanilla options using the Longstaff-Schwartz model.
optstocksensbyls	Calculate vanilla option prices and sensitivities using the Longstaff-Schwartz model.
optByHestonFFT	Calculate option price by Heston model using FFT and FRFT.
optSensByHestonFFT	Calculate option price and sensitivities by Heston model using FFT and FRFT.
optByHestonNI	Calculate option price by Heston model using numerical integration.
optSensByHestonNI	Calculate option price and sensitivities by Heston model using numerical integration.
optByBatesFFT	Calculate option price by Bates model using FFT and FRFT.
optSensByBatesFFT	Calculate option price and sensitivities by Bates model using FFT and FRFT.
optByBatesNI	Calculate option price by Bates model using numerical integration.
optSensByBatesNI	Calculate option price or sensitivities by Bates model using numerical integration.
optByMertonFFT	Calculate option price by Merton76 model using FFT and FRFT.
optSensByMertonFFT	Calculate option price and sensitivities by Merton76 model using FFT and FRFT.
optByMertonNI	Calculate option price by Merton76 model using numerical integration.
optSensByMertonNI	Calculate option price and sensitivities by Merton76 model using numerical integration.
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 Instruments 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.

Spread Option

A spread option is an option written on the difference of two underlying assets. For example, a European call on the difference of two assets *X1* and *X2* would have the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

where:

K is the strike price.

The following functions support spread options.

Function	Purpose
`spreadbykirk`	Price European spread options using the Kirk pricing model.
`spreadsensbykirk`	Calculate European spread option prices and sensitivities using the Kirk pricing model.
`spreadbybjs`	Price European spread options using the Bjerksund-Stensland pricing model.
`spreadsensbybjs`	Calculate European spread option prices and sensitivities using the Bjerksund-Stensland pricing model.
`spreadbyfd`	Price European or American spread options using the Alternate Direction Implicit (ADI) finite difference method.
`spreadsensbyfd`	Calculate price and sensitivities of European or American spread options using the Alternate Direction Implicit (ADI) finite difference method.
`spreadbyls`	Price European or American spread options using Monte Carlo simulations.
`spreadsensbyls`	Calculate price and sensitivities for European or American spread options using Monte Carlo simulations.

One-Touch and Double One-Touch Options

A touch option (also known as binary barrier options or American digitals) are path-dependent options in which the existence and payment of the options depend on the movement of the underlying spot through their option life.

There are two types of touch options:

- One-Touch

The one-touch (no-touch) option provides a payoff if the underlying spot ever (never) trades at or beyond the barrier level and zero otherwise.

- Double One-Touch

  The double one-touch and double no-touch option works the same way as a one-touch option, but has two barriers. A double one-touch (double no-touch) option provides a payoff if the underlying spot ever (never) touches either the upper or lower barriers levels.

The following functions support touch options.

Function	Purpose
`touchbybls`	Price one-touch and no-touch binary options using the Black-Scholes option pricing model.
`touchsensbybls`	Calculate price and sensitivities for one-touch and no-touch binary options using the Black-Scholes option pricing model.
`dbltouchbybls`	Price double one-touch and double no-touch binary options using the Black-Scholes option pricing model
`dbltouchsensbybls`	Calculate price and sensitivities for double one-touch and double no-touch binary options using the Black-Scholes option pricing model

Forwards Option

A forward option is a non-standardized contract between two parties to buy or to sell an asset at a specified future time at a price agreed upon today. The buyer of a forward option contract has the right to hold a particular forward position at a specific price any time before the option expires. The forward option seller holds the opposite forward position when the buyer exercises the option. A call option is the right to enter into a long forward position and a put option is the right to enter into a short forward position. A closely related contract is a futures contract. A forward is like a futures in that it specifies the exchange of goods for a specified price at a specified future date. The table below displays some of the characteristics of forward and futures contracts.

Forwards	Futures
Customized contracts	Standardized contracts
Over the counter traded	Exchange traded
Exposed to default risk	Clearing house reduces default risk
Mostly used for hedging	Mostly used by hedgers and speculators
Settlement at the end of contract (no Margin required)	Daily changes are settled day by day (Margin required)
Delivery usually takes place	Delivery usually never happens

The payoff for a forward option, where the value of a forward position at maturity depends on the relationship between the delivery price (K) and the underlying price (S_T) at that time, is:

- For a long position: $f_T = S_T - K$
- For a short position: $f_T = K - S_T$

The following functions support pricing a forwards option.

Function	Purpose
optstockbyblk	Price options on forwards using the Black option pricing model.
optstocksensbyblk	Determine option prices and sensitivities on forwards using the Black pricing model.

Futures Option

A future option is a standardized contract between two parties to buy or sell a specified asset of standardized quantity and quality for a price agreed upon today (the futures price) with delivery and payment occurring at a specified future date, the delivery date. The contracts are negotiated at a futures exchange, which acts as an intermediary between the two parties. The party agreeing to buy the underlying asset in the future, the "buyer" of the contract, is said to be "long", and the party agreeing to sell the asset in the future, the "seller" of the contract, is said to be "short."

Forwards	Futures
Customized contracts	Standardized contracts
Over the counter traded	Exchange traded
Exposed to default risk	Clearing house reduces default risk
Mostly used for hedging	Mostly used by hedgers and speculators
Settlement at the end of contract (no Margin required)	Daily changes are settled day by day (Margin required)
Delivery usually takes place	Delivery usually never happens

A futures contract is the delivery of item J at time T and:

- There exists in the market a quoted price $F(t,T)$, which is known as the futures price at time t for delivery of J at time T.
- The price of entering a futures contract is equal to zero.
- During any time interval $[t,s]$, the holder receives the amount $F(s,T) - F(t,T)$ (this reflects instantaneous marking to market).
- At time T, the holder pays $F(T,T)$ and is entitled to receive J. Note that $F(T,T)$ should be the spot price of J at time T.

The following functions support pricing a futures option.

Function	Purpose
optstockbyblk	Price options on futures using the Black option pricing model.
optstocksensbyblk	Determine option prices and sensitivities on futures using the Black pricing model.

See Also

crrtree | eqptree | lrtree | stockspec | crrtimespec | eqptimespec | lrtimespec | itttree | itttimespec | treepath | trintreepath | asianbycrr | barrierbycrr | compoundbycrr | crrprice | crrsens | lookbackbycrr | optstockbycrr | instasian | instbarrier | instcompound | instlookback | instoptstock | asianbyeqp | barrierbyeqp | compoundbyeqp | eqpprice | eqpsens | lookbackbyeqp | optstockbyeqp | optstockbylr |

`optstocksensbylr` | `asianbyitt` | `barrierbyitt` | `compoundbyitt` | `ittprice` | `ittsens` | `lookbackbyitt` | `optstockbyitt` | `assetbybls` | `assetsensbybls` | `cashbybls` | `cashsensbybls` | `chooserbybls` | `gapbybls` | `gapsensbybls` | `impvbybls` | `optstockbybls` | `optstocksensbybls` | `supersharebybls` | `supersharesensbybls` | `impvbyblk` | `optstockbyblk` | `optstocksensbyblk` | `impvbyrgw` | `optstockbyrgw` | `optstocksensbyrgw` | `impvbybjs` | `optstockbybjs` | `optstocksensbybjs` | `spreadbybjs` | `spreadsensbybjs` | `basketbyju` | `basketsensbyju` | `basketstockspec` | `maxassetbystulz` | `maxassetsensbystulz` | `minassetbystulz` | `minassetsensbystulz` | `spreadbykirk` | `spreadsensbykirk` | `asianbykv` | `asiansensbykv` | `asianbylevy` | `asiansensbylevy` | `lookbackbycvgsg` | `lookbacksensbycvgsg` | `basketbyls` | `basketsensbyls` | `basketstockspec` | `asianbyls` | `asiansensbyls` | `lookbackbyls` | `lookbacksensbyls` | `lookbackbyls` | `lookbacksensbyls` | `spreadbyls` | `spreadsensbyls` | `optstockbyls` | `optstocksensbyls` | `optpricebysim`

Related Examples

- “Understanding Equity Trees” on page 3-2
- “Pricing Equity Derivatives Using Trees” on page 3-64
- “Creating Instruments or Properties” on page 1-16
- “Graphical Representation of Equity Derivative Trees” on page 3-73
- “Compute Option Prices on a Forward” on page 11-1597
- “Compute Forward Option Prices and Delta Sensitivities” on page 11-1662
- “Compute the Option Price on a Future” on page 11-1598
- “Pricing European Call Options Using Different Equity Models” on page 3-88
- “Pricing Asian Options” on page 3-110
- “Equity Derivatives Using Closed-Form Solutions” on page 3-79
- “Pricing Using the Bjerksund-Stensland Model” on page 3-84

More About

- “Basket Option” on page 3-22
- “Asian Option” on page 3-19
- “Spread Option” on page 3-30
- “Vanilla Option” on page 3-27
- “Rainbow Option” on page 3-27
- “Bjerksund-Stensland 2002 Model” on page 3-81
- “Roll-Geske-Whaley Model” on page 3-80
- “Black Model” on page 3-80
- “Digital Option” on page 3-26
- “Supported Energy Derivative Functions” on page 3-34
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Supported Energy Derivative Functions

In this section...

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

where:

S_{av} is the average price of underlying asset.

S is the price of the underlying asset.

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
`asianbyls`	Price European or American Asian options using the Longstaff-Schwartz model.
`asiansensbyls`	Calculate prices and sensitivities of European or American Asian options using the Longstaff-Schwartz model.
`asianbykv`	Price European geometric Asian options using the Kemna Vorst model.

Function	Purpose
asiansensbykv	Calculate prices and sensitivities of European geometric Asian options using the Kemna Vorst model.
asianbylevy	Price European arithmetic Asian options using the Levy model.
asiansensbylevy	Calculate prices and sensitivities of European arithmetic Asian options using the Levy model.
asianbyhhm	Calculate prices of European discrete arithmetic fixed Asian options using the Haug, Haug, Margrabe model.
asiansensbyhhm	Calculate prices and sensitivities of European discrete arithmetic fixed Asian options using the Haug, Haug, Margrabe model
asianbytw	Calculate prices of European arithmetic fixed Asian options using the Turnbull Wakeman model.
asiansensbytw	Calculate prices and sensitivities of European arithmetic fixed Asian options using the Turnbull Wakeman model.
asianbycrr	Price an Asian option from a Cox-Ross-Rubinstein binomial tree.
asianbyeqp	Price an Asian option from an Equal Probabilities binomial tree.
asianbyitt	Price an Asian option using an implied trinomial tree (ITT).
asianbystt	Price an Asian option using a standard trinomial tree.
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 asset 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 asset passes above a barrier that is above the initial asset 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 asset 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 asset 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 asset 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
`barrierbyfd`	Price barrier option using finite difference method.
`barriersensbyfd`	Calculate barrier option price and sensitivities using finite difference method.
`barrierbyls`	Price European or American barrier options using Monte Carlo simulations.
`barrierbybls`	Price European barrier options using Black-Scholes option pricing model.
`barrierbycrr`	Price a barrier option from a Cox-Ross-Rubinstein binomial tree.
`barrierbyeqp`	Price a barrier option from an Equal Probabilities binomial tree.
`barrierbyitt`	Price a barrier option using an implied trinomial tree (ITT).
`barrierbystt`	Price a barrier options using a standard trinomial tree.

Double Barrier Option

A double barrier option is similar to the standard single barrier option except that they have two barrier levels: a lower barrier (LB) and an upper barrier (UB). The payoff for a double barrier option depends on whether the underlying asset remains between the barrier levels during the life of the option. Double barrier options are less expensive than single barrier options as the probability of being knocked out is higher. Because of this, double barrier options allow investors to achieve reduction in the option premiums as and match an investor’s belief about the future movement of the underlying price process.

There are two types of double barrier options:

- Double Knock-in

  This option becomes effective when the price of the underlying asset reaches one of the barriers. It gives the option holder, the right but not the obligation to buy or sell the underlying security at the strike price, if the underlying asset goes above or below the barrier levels during the life of the option.
- Double Knock-out

  This option gives the option holder, the right but not the obligation to buy or sell the underlying security at the strike price, as long as the underlying asset remains between the barrier levels during the life of the option. This option terminates when the price of the underlying asset passes one of the barriers.

The following functions support double barrier options.

Function	Purpose
`dblbarrierbybls`	Price European double barrier options using the Black-Scholes option pricing model.
`dblbarriersensbybls`	Calculate the price and sensitivities for a European double barrier options using the Black-Scholes option pricing model.
`dblbarrierbyfd`	Price double barrier option prices using the finite difference method.
`dblbarriersensbyfd`	Calculate the price and sensitivities for a double barrier option using the finite difference method.

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 asset at time t.

K is the strike price.

The following functions support specifying or pricing a vanilla option.

Function	Purpose
`optstockbyls`	Price European, Bermudan, or American vanilla options using the Longstaff-Schwartz model.
`optstocksensbyls`	Calculate European, Bermudan, or American vanilla option prices and sensitivities using the Longstaff-Schwartz model.
`optstockbyfd`	Calculate vanilla option prices using finite difference method.
`optstocksensbyfd`	Calculate vanilla option prices and sensitivities using finite difference method.
`optstockbybaw`	Calculate American options prices using Barone-Adesi and Whaley option pricing model.
`optstocksensbybaw`	Calculate American options prices and sensitivities using Barone-Adesi and Whaley option pricing model.
`optstockbyrgw`	Calculate American call option prices using Roll-Geske-Whaley option pricing model.
`optstocksensbyrgw`	Calculate American call option prices or sensitivities using Roll-Geske-Whaley option pricing model.
`optByLocalVolFD`	Calculate vanilla option price by local volatility model, using finite differences.

Function	Purpose
optstockbybjs	Price American options using Bjerksund-Stensland 2002 option pricing model.
optstocksensbybjs	Determine American option prices or sensitivities using Bjerksund-Stensland 2002 option pricing model.
optSensByLocalVolFD	Calculate vanilla option price or sensitivities by local volatility model, using finite differences.
optByHestonFD	Calculate vanilla option price by Heston model using finite differences.
optSensByHestonFD	Calculate vanilla option price and sensitivities by Heston model using finite differences.
optByBatesFD	Calculates vanilla European option price by Bates model using finite differences.
optSensByBatesFD	Calculates vanilla European option price and sensitivities by Bates model using finite differences.
optByMertonFD	Calculates vanilla European option price by Merton76 model using finite differences.
optSensByMertonFD	Calculates vanilla European option price and sensitivities by Merton76 model using finite differences.
optByBatesFFT	Calculate option price by Bates model using FFT and FRFT.
optByHestonFFT	Calculate option price by Heston model using FFT and FRFT.
optByMertonFFT	Calculate option price by Merton76 model using FFT and FRFT.
optstockbycrr	Price an option from a Cox-Ross-Rubinstein binomial tree.
optstockbyeqp	Price an option from an Equal Probabilities binomial tree.
optstockbyitt	Price an option using an implied trinomial tree (ITT).
optstockbystt	Price an option using a standard trinomial tree.

Spread Option

A spread option is an option written on the difference of two underlying assets. For example, a European call on the difference of two assets *X1* and *X2* would have the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

where:

K is the strike price.

The following functions support spread options.

Function	Purpose
spreadbykirk	Price European spread options using the Kirk pricing model.
spreadsensbykirk	Calculate European spread option prices and sensitivities using the Kirk pricing model.

Function	Purpose
`spreadbybjs`	Price European spread options using the Bjerksund-Stensland pricing model.
`spreadsensbybjs`	Calculate European spread option prices and sensitivities using the Bjerksund-Stensland pricing model.
`spreadbyfd`	Price European or American spread options using the Alternate Direction Implicit (ADI) and Crank-Nicolson finite difference methods.
`spreadsensbyfd`	Calculate price and sensitivities of European or American spread options using the Alternate Direction Implicit (ADI) and Crank-Nicolson finite difference methods.
`spreadbyls`	Price European or American spread options using Monte Carlo simulations.
`spreadsensbyls`	Calculate price and sensitivities for European or American spread options using Monte Carlo simulations.

For more information on using spread options, see “Pricing European and American Spread Options” on page 3-97.

Lookback Option

A lookback option is a path-dependent option based on the maximum or minimum value the underlying asset (e.g. electricity, stock) achieves during the entire life of the option. Basically the holder of the option can ‘look back’ over time to determine the payoff. This type of option provides price protection over a selected period, reduces uncertainties with the timing of market entry, moderates the need for the ongoing management, and therefore, is usually more expensive than vanilla options.

Lookback call options give the holder the right to buy the underlying asset at the lowest price. Lookback put options give the right to sell the underlying asset at the highest price.

Financial Instruments Toolbox software supports two types of lookback options: fixed and floating. The difference is related to how the strike price is set in the contract. Fixed lookback options have a specified strike price and the option pays out the maximum of the difference between the highest (lowest) observed price of the underlying during the life of the option and the strike. Floating lookback options have a strike price determined at maturity, and it is set at the lowest (highest) price of the underlying reached during the life of the option. This means that for a floating strike lookback call (put), the holder has the right to buy (sell) the underlying asset at its lowest (highest) price observed during the life of the option. So, 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 asset.

S_{min} is the minimum price of underlying asset.

S is the price of the underlying asset at maturity.

X is the strike price.

The following functions support lookback options.

Function	Purpose
`lookbackbycvgsg`	Calculate prices of European lookback fixed and floating strike options using the Conze-Viswanathan and Goldman-Sosin-Gatto models.
`lookbacksensbycvgsg`	Calculate prices and sensitivities of European fixed and floating strike lookback options using the Conze-Viswanathan and Goldman-Sosin-Gatto models.
`lookbackbyls`	Calculate prices of lookback fixed and floating strike options using the Longstaff-Schwartz model.
`lookbacksensbyls`	Calculate prices and sensitivities of lookback fixed and floating strike options using the Longstaff-Schwartz model.
`lookbackbycrr`	Price a lookback option from a Cox-Ross-Rubinstein binomial tree.
`lookbackbyeqp`	Price a lookback option from an Equal Probabilities binomial tree.
`lookbackbyitt`	Price a lookback option using an implied trinomial tree (ITT).
`lookbackbystt`	Price a lookback option using a standard trinomial tree.

Lookback options and Asian options are instruments used in the electricity market to manage purchase timing risk. Electricity purchasers cover part of their expected electricity consumption on the forward market to avoid the volatility and limited liquidity of the spot market. Using Asian options as a hedging tool is a passive approach to solving the purchase timing problem. An Asian option instrument diminishes the wrong timing risk but it also reduces any potential benefit to the buyer from falling prices. On the other hand, lookback options allow the purchasers to buy electricity at the lowest price, but as mentioned before, this instrument is more expensive than Asian and vanilla options.

Forwards Option

A forward option is a non-standardized contract between two parties to buy or to sell an asset at a specified future time at a price agreed upon today. The buyer of a forward option contract has the right to hold a particular forward position at a specific price any time before the option expires. The forward option seller holds the opposite forward position when the buyer exercises the option. A call option is the right to enter into a long forward position and a put option is the right to enter into a short forward position. A closely related contract is a futures contract. A forward is like a futures in that it specifies the exchange of goods for a specified price at a specified future date. The following table displays some of the characteristics of forward and futures contracts.

Forwards	Futures
Customized contracts	Standardized contracts
Over the counter traded	Exchange traded
Exposed to default risk	Clearing house reduces default risk

Forwards	Futures
Mostly used for hedging	Mostly used by hedgers and speculators
Settlement at the end of contract (no Margin required)	Daily changes are settled day by day (Margin required)
Delivery usually takes place	Delivery usually never happens

The payoff for a forward option, where the value of a forward position at maturity depends on the relationship between the delivery price (K) and the underlying price (S_T) at that time, is:

- For a long position: $f_T = S_T - K$
- For a short position: $f_T = K - S_T$

The following functions support pricing a forwards option.

Function	Purpose
`optstockbyblk`	Price options on forwards using the Black option pricing model.
`optstocksensbyblk`	Determine option prices and sensitivities on forwards using the Black pricing model.

Futures Option

A future option is a standardized contract between two parties to buy or sell a specified asset of standardized quantity and quality for a price agreed upon today (the futures price) with delivery and payment occurring at a specified future date, the delivery date. The contracts are negotiated at a futures exchange, which acts as an intermediary between the two parties. The party agreeing to buy the underlying asset in the future, the "buyer" of the contract, is said to be "long", and the party agreeing to sell the asset in the future, the "seller" of the contract, is said to be "short."

Forwards	Futures
Customized contracts	Standardized contracts
Over the counter traded	Exchange traded
Exposed to default risk	Clearing house reduces default risk
Mostly used for hedging	Mostly used by hedgers and speculators
Settlement at the end of contract (no Margin required)	Daily changes are settled day by day (Margin required)
Delivery usually takes place	Delivery usually never happens

A futures contract is the delivery of item J at time T and:

- There exists in the market a quoted price $F(t, T)$, which is known as the futures price at time t for delivery of J at time T.
- The price of entering a futures contract is equal to zero.
- During any time interval $[t,s]$, the holder receives the amount $F(s, T) - F(t, T)$ (this reflects instantaneous marking to market).
- At time T, the holder pays $F(T, T)$ and is entitled to receive J. Note that $F(T, T)$ should be the spot price of J at time T.

The following functions support pricing a futures option.

Function	Purpose
`optstockbyblk`	Price options on futures using the Black option pricing model.
`optstocksensbyblk`	Determine option prices and sensitivities on futures using the Black pricing model.

See Also

`spreadbyls` | `spreadsensbyls` | `asianbyls` | `asiansensbyls` | `lookbackbyls` | `lookbacksensbyls` | `optstockbyls` | `optstocksensbyls` | `optpricebysim` | `spreadbykirk` | `spreadsensbykirk` | `spreadbybjs` | `spreadsensbybjs` | `asianbykv` | `asiansensbykv` | `asianbylevy` | `asiansensbylevy` | `lookbackbycvgsg` | `lookbacksensbycvgsg` | `optstockbyblk` | `optstocksensbyblk` | `spreadbyfd` | `spreadsensbyfd`

Related Examples

- “Pricing European and American Spread Options” on page 3-97
- “Hedging Strategies Using Spread Options” on page 4-35
- “Pricing Swing Options Using the Longstaff-Schwartz Method” on page 3-43
- “Compute Option Prices on a Forward” on page 11-1597
- “Compute Forward Option Prices and Delta Sensitivities” on page 11-1662
- “Compute the Option Price on a Future” on page 11-1598
- “Simulating Electricity Prices with Mean-Reversion and Jump-Diffusion” on page 3-53
- “Pricing Asian Options” on page 3-110

More About

- “Forwards Option” on page 3-40
- “Futures Option” on page 3-41
- “Spread Option” on page 3-38
- “Asian Option” on page 3-34
- “Vanilla Option” on page 3-37
- “Lookback Option” on page 3-39
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Pricing Swing Options Using the Longstaff-Schwartz Method

This example shows how to price a swing option using a Monte Carlo simulation and the Longstaff-Schwartz method. A risk-neutral simulation of the underlying natural gas price is conducted using a mean-reverting model. The simulation results are used to price a swing option based on the Longstaff-Schwartz method [6 on page 3-51]. This approach uses a regression technique to approximate the continuation value of the option. A comparison is made between a polynomial and spline basis to fit the regression. Finally, the resulting prices are analyzed against lower and upper price boundaries derived from standard European and American options.

Overview of Swing Options

Swing options are popular financial instruments in the energy market, which provide flexibility in the volume of the delivered asset. In order for energy consumers to protect themselves against fluctuations in energy prices, they want to lock in a price by purchasing a forward contract, called the *baseload forward contract*. However, consumers do not know exactly how much energy will be used in the future, and energy commodities such as electricity and gas cannot easily be stored. Therefore, the consumer wants the flexibility to change the amount of energy that is delivered at each delivery date. Swing options provide this flexibility. Thus, the full contract is composed of two parts: the baseload forward contract, and the swing option component.

Swing options are generally over-the-counter (OTC) contracts that can be highly customized. Therefore, there are many different types of constraints and penalties (see [5 on page 3-51] for more details). In this example, a swing option is priced where the only constraint is the daily volume, which is known as the Daily Contract Quantity (DCQ). When a swing right is exercised, the volume cannot go below the minimum DCQ (`minDCQ`), or go above the maximum DCQ (`maxDCQ`).

There are several methods to price swing options, such as finite differences, simulation, and dynamic programming based on trees [5 on page 3-51]. This example uses the simulation-based approach with the Longstaff-Schwartz method. The benefit of the simulation-based approach is that the dynamics used to simulate the underlying asset price are separated from the pricing algorithm. In the finite difference and tree based methods, the pricing algorithm must be changed in order to consider pricing with a different underlying price dynamic.

Risk-Neutral Simulation of Natural Gas Price

In this example, natural gas is used as the underlying asset with the following mean-reverting dynamic [8]:

$$dS_t = \kappa(\mu - \log(S_t))S_t dt + \sigma S_t dW_t$$

where W_t is a standard Brownian motion. Applying Ito's Lemma to the logarithm of the price leads to an Orstein-Uhlenbeck process:

$$dX_t = \kappa(\theta - X_t)dt + \sigma dW_t$$

where $X_t = \log(S_t)$, $\kappa > 0$, and θ is defined as:

$$\theta = \mu - \frac{\sigma^2}{2\kappa}$$

θ is the mean-reversion level that determines the value at which the simulated values will revert to in the long run. κ is the mean-reversion speed that determines how fast this reversion occurs. σ is the

volatility of X. We first proceed by simulating the logarithm of the price. Afterwards, the exponential of the simulated values are taken to obtain the prices.

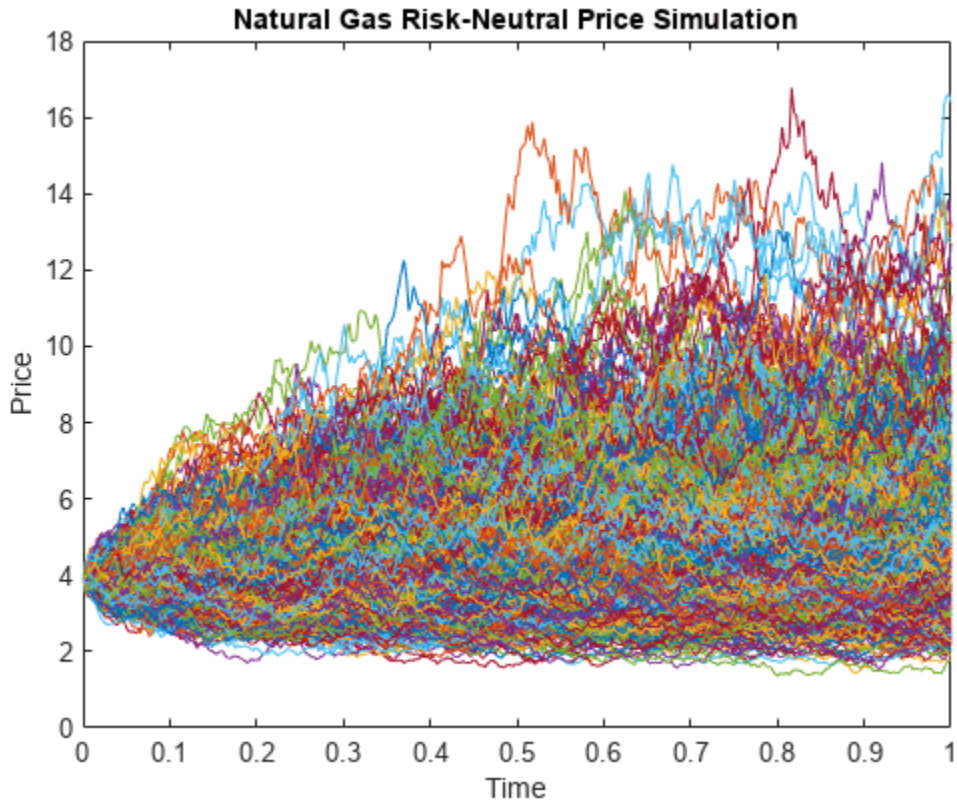
The length of the simulation is for a one year period, with the initial price of 3.9 dollars per MMBtu. The Monte Carlo simulation is conducted for 1,000 trials, with daily periods. In practice, these parameters are calibrated against market data. In this example, $\kappa = 1.2$, $\theta = 1.7$, and $\sigma = 59\%$. The `hwv` object from the Financial Toolbox™ is used to simulate the mean-reverting dynamics of the natural gas price.

```
% Settlement date
Settle = '01-Jun-2014';

% Maturity Date
Maturity = '01-Jun-2015';

% Actual/Actual basis
Basis = 0;

% Initial log(price in $/MMBtu)
X0 = log(3.9);

% Volatility of log(price)
Sigma = 0.59;

% Number of trials in the Monte Carlo simulation
NumTrials =1000;

% Number of periods (daily)
NumPeriods = daysdif(Settle, Maturity, Basis);

% Daily time step
dt = 1/NumPeriods;

% Mean reversion speed of log(price)
Kappa = 1.2;

% Mean reversion level of log(price)
Theta = 1.7;

% Create HWV object
hwvobj = hwv(Kappa, Theta, Sigma, 'StartState', X0);
```

The simulation is run and plotted.

```
% Set random number generator seed
savedState = rng(0, 'twister');

% Simulate gas prices
[Paths, Times] = hwvobj.simBySolution(NumPeriods, 'NTRIALS', NumTrials, ...
    'DeltaTime', dt);
Paths = squeeze(exp(Paths));

% Restore random number generator state
rng(savedState);

% Plot paths
figure;
plot(Times, Paths);
```

```
title('Natural Gas Risk-Neutral Price Simulation');
xlabel('Time');
ylabel('Price');
```



In this example, natural gas is used as the underlying asset with a mean-reverting dynamic. However, the Longstaff-Schwartz algorithm can be used for other underlying assets, such as electricity, with any underlying price dynamic.

Pricing the Swing Option

We consider a swing option with five swing rights at the strike of $4.69/MMBtu, which can be exercised daily between the day after the settlement date and the maturity date. The Daily Contract Quantity (DCQ) is 10,000 MMBtu, which is the average amount of natural gas that the consumer expects to purchase on a given day. The consumer has the flexibility to reduce the purchase amount (downswing) in one day to the minimum DCQ of 2,500 MMBtu, or increase the purchase (upswing) to 15,000 MMBtu. The continuously compounded annual risk-free rate is 1%.

`RateSpec` is used to represent the interest-rate term structure. For the sake of simplicity, we consider a flat interest-rate term structure in this example. The values of `RateSpec` can be modified to accommodate any interest-rate curve. The function `hswingbyls` in this example assumes a daily exercise if the `ExerciseDates` input is empty.

```
% Define RateSpec
rfrate = 0.01;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
                     'EndDates', Maturity, 'Rates', rfrate, ...
```

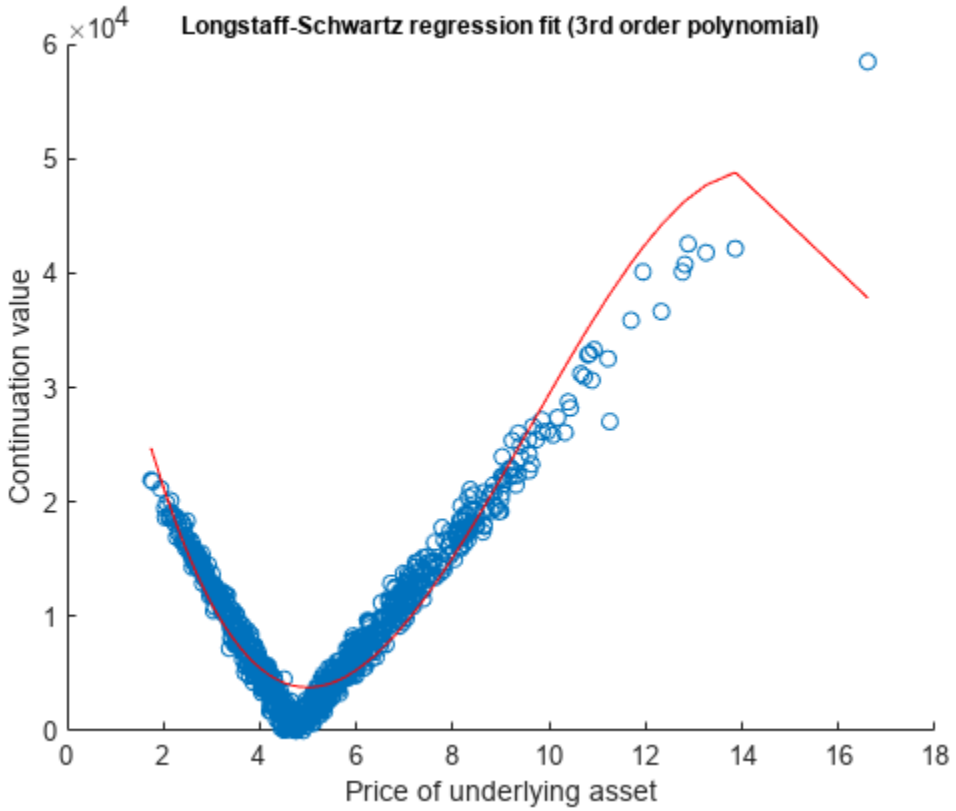
```
                    'Compounding', Compounding, 'Basis', Basis);

% Daily exercise
% hswingbyls assumes daily exercise for empty ExerciseDates
ExerciseDates = [];

% Number of swings
NumSwings = 5;

% Daily Contract Quantity in MMBtu
DCQ = 10000;

% Minimum DCQ constraint in MMBtu
minDCQ = 2500;

% Maximum DCQ constraint in MMBtu
maxDCQ = 15000;

% Strike
Strike = 4.69;
```

The Longstaff-Schwartz method is a backward iteration algorithm, which steps backward in time from the maturity date. At each exercise date, the algorithm approximates the continuation value, which is the value of the option if it is not exercised. This is done by fitting a regression against the values of the simulated prices and the discounted future value of the option at the next exercise date. The future value of the option is known as the algorithm moves backward in time. The continuation value is compared to the sum of the payoff from immediate exercise (a downswing or upswing) and the continuation value of a swing option with one less swing right. If this sum is smaller, the option holder's optimal strategy is to not exercise on that date. The function `hswingbyls` in this example uses this method to determine the optimal exercise strategy and the price for swing options [1 on page 3-50,2 on page 3-50,7 on page 3-51].

As discussed earlier, the only constraint considered in this example is the minimum and maximum DCQ. In this case, the optimal early exercise strategy is of a "bang-bang" type. This means that when it is optimal to upswing or downswing at a certain exercise date, the option holder should always exercise at the maximum or minimum DCQ to maximize profit. The "bang-bang" exercise would not be the optimal strategy if, for example, there is a terminal penalty based on volume. The pricing algorithm would then need to additionally keep track of all possible volume levels, which significantly adds to the runtime performance cost.

First, the swing option is priced using a third order polynomial to fit the regression of the Longstaff-Schwartz method. The function `hswingbyls` also generates a plot of the regression between the underlying price and the continuation value at the exercise date before maturity.

```
% Price swing option using 3rd order polynomial to fit Longstaff-Schwartz
% regression
tic;
useSpline = false;
SwingPrice = hswingbyls(Paths, Times, RateSpec, Settle, Maturity, ...
    Strike, ExerciseDates, NumSwings, DCQ, minDCQ, maxDCQ, useSpline, ...
    [], true)
```

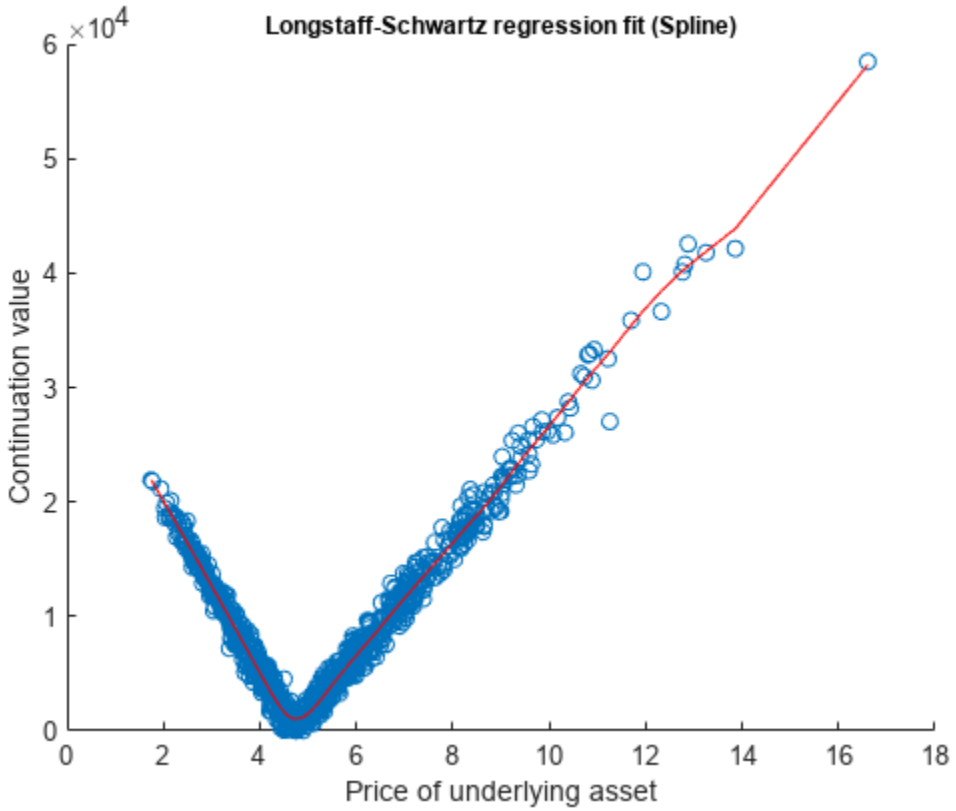



```
SwingPrice = 5.6943e+04
```

```
lsPolyTime = toc;
```

The above plot of the regression fit shows that the third order polynomial does not fit the continuation value perfectly, especially near the hinge and at the extreme points. Use the `csaps` function to fit the regression using a cubic smoothing spline with a smoothing parameter of 0.7. The Curve Fitting Toolbox™ is required to run the remainder of the example.

```
% Price swing option using smoothed splines to fit Longstaff-Schwartz
% regression
tic;
useSpline = true;
smoothingParam = 0.7;
SwingPriceSpline = hswingbyls(Paths, Times, RateSpec, Settle, Maturity, ...
    Strike, ExerciseDates, NumSwings, DCQ, minDCQ, maxDCQ, useSpline, ...
    smoothingParam, true)
```




```
SwingPriceSpline = 6.0729e+04
```

```
lsSplineTime = toc;
```

The plot of the regression shows that the cubic smoothing spline has a better fit against the data, thus obtaining a more accurate value for the continuation values. However, the comparison below shows that using a cubic smoothing spline takes longer than a third order polynomial.

```
% Print comparison of running times
displayRunningTimes(lsPolyTime, lsSplineTime)
```

```
Comparison of running times:

3rd order polynomial: 6.29 sec
Spline              : 20.44 sec
```

Also, it is important to note that the price represents solely the optionality component. Hence, the price of the baseload forward contract is not included in the above calculated price. Because we used a fixed strike price, the baseload contract has a non-zero value, which can be calculated by:

$$BaseLoadPrice = \sum_{i=1}^{N} e^{-rt_i} E(S_{t_i} - K)$$

where $t_i, i = 1, \ldots, N$, are the exercise dates (see [3 on page 3-50] for more details). The full price of the contract, including the baseload and the swing option, is calculated below using the swing option price from the smoothed cubic spline.

```
% Obtain discount factors
RS2 = intenvset(RateSpec, 'StartTimes', 0, 'EndTimes', Times(2:end));
D = intenvget(RS2,'Disc');

% Calculate baseload price
BaseLoadPrice = DCQ.*mean(Paths(2:end,:)-Strike,2)'*D;

% Calculate full contract price, based on results from cubic spline LS
FullContractPrice = BaseLoadPrice + SwingPriceSpline

FullContractPrice = 1.2479e+05
```

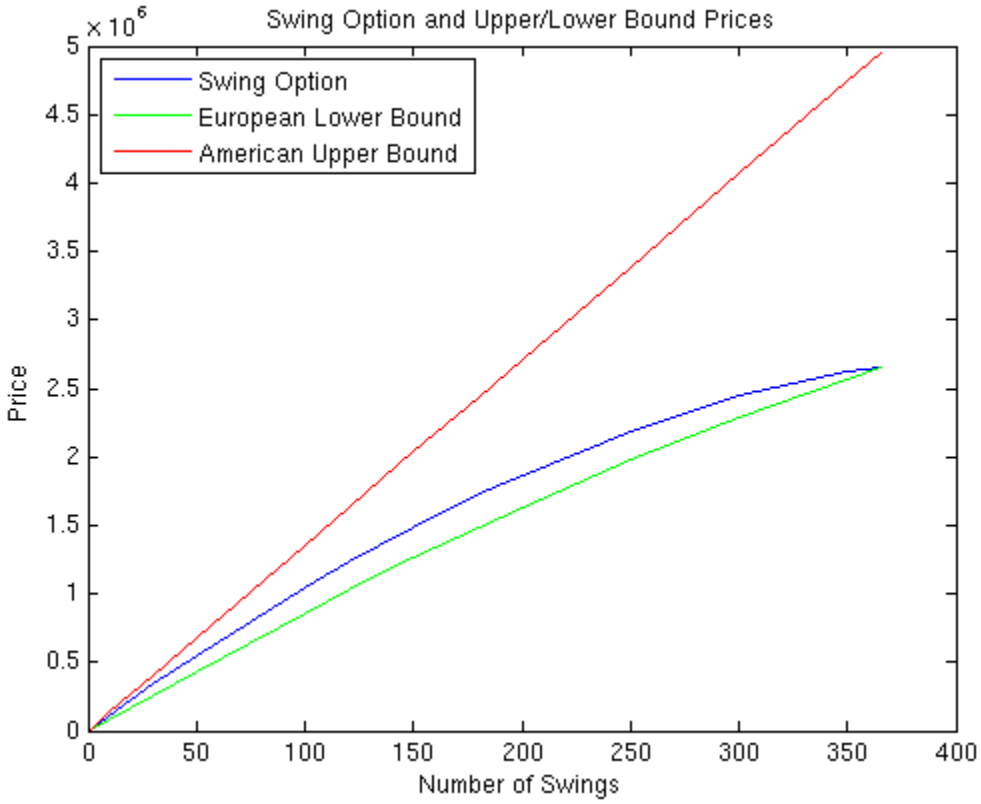
Price Bounds

A lower bound for the swing option is a strip of European options, and the upper bound is a strip of American options [4 on page 3-50]. Compared to European options, swing options have an early exercise premium at each exercise date, thus the price should be higher. The price is lower than the American option strips, because only a single swing right can be exercised at each exercise date. More than one strip can be exercised in a single day using American options.

The prices for the strips of the lower and upper bounds are calculated below to check that the swing option prices are within these bounds. The European strip prices are calculated against the last five exercise dates.

```
% Obtain discount factor for the last NumSwings exercise dates
D = D(end-NumSwings+1:end);

% European lower bound
idx = size(Paths, 1):-1:(size(Paths, 1) - NumSwings + 1);
putEuro = D'*mean(max(Strike - Paths(idx,:), 0),2);
callEuro = D'*mean(max(Paths(idx,:) - Strike, 0),2);
lowerBound = ((DCQ-minDCQ).*putEuro+(maxDCQ-DCQ).*callEuro);

% American upper bound
[putAmer, callAmer] = hamericanPrice(Paths, Times, RateSpec, Strike);
upperBound = NumSwings.*((DCQ-minDCQ).*putAmer+(maxDCQ-DCQ).*callAmer);

% Print price and lower/upper bounds
displaySummary(SwingPriceSpline, lowerBound, upperBound);

Comparison to lower and upper bounds:

Lower bound (European) : 44412.14
Swing Option Price     : 60729.00
Upper bound (American) : 68181.42
```

The prices calculated using the Longstaff-Schwartz algorithm are within the lower and upper bounds. The plot below shows a comparison between the swing option and the upper and lower bounds as the number of swings increases. When the number of swings is 1, the swing option is equivalent to an American option. In the case of daily exercise opportunity (`NumSwings = 365`), the swing option is equivalent to the strip of European options with daily maturity.



Conclusion

The example shows the use of the Longstaff-Schwartz method to price a swing option where the underlying asset follows a mean-reverting dynamic. A 3rd order polynomial and a smoothed cubic spline are used to fit the regression in the Longstaff-Schwartz algorithm to approximate the continuation value. It was shown that the smoothed cubic spline fits the data better at the cost of slower performance. Finally, the resulting swing option prices were checked against the lower bound of a strip of European options and an upper bound of a strip of American options.

References

[1] Boogert, A., de Jong, C. "Gas Storage Valuation Using a Monte Carlo Method." *Journal of Derivatives.* 15(3):81-98, 2008.

[2] Dorr, Uwe. "Valuation of Swing Options and Estimation of Exercise strategies by Monte Carlo Techniques." Oxford, 2002.

[3] Hull, John C. *Options, Futures, and Other Derivatives*. Sixth Edition, Pearson Education, Inc., 2006.

[4] Jaillet, P., Ronn, E. I., Tompaidis, S. "Valuation of Commodity-Based Swing Options, *Management Science*. 50(7):909-921, 2004.

[5] Loland, Ambers, Lindqvist, Ola. "Valuation of Commodity-Based Swing Options: A Survey." Norsk, Regnesentral, 2008.

[6] Longstaff, Francis A., Schwartz, Eduardo S. "Valuing American Options by Simulation: A Simple Least-Squares Approach." *The Review of Financial Studies.* 14(1):113-147, 2001.

[7] Meinshausen, N., Hambly, B.M. "Monte Carlo Methods for the Valuation of Multiple Exercise Options." *Mathematical Science.* 14:557-583, 2004.

[8] Schwartz, Eduardo S. "The Stochastic Behavior of Commodity Prices: Implications for Valuation and Hedging." *The Journal of Finance.* 52(3):923-973, 1997.

Utility Functions

```
function displaySummary(SwingPriceSpline, lowerBound, upperBound)
fprintf('Comparison to lower and upper bounds:\n');
fprintf('\n')
fprintf('Lower bound (European) : %.2f\n', lowerBound);
fprintf('Swing Option Price     : %.2f\n', SwingPriceSpline);
fprintf('Upper bound (American) : %.2f\n\n', upperBound);
end

function displayRunningTimes(lsPolyTime, lsSplineTime)
fprintf('Comparison of running times:\n');
fprintf('\n')
fprintf('3rd order polynomial: %.2f sec\n', lsPolyTime);
fprintf('Spline              : %.2f sec\n\n', lsSplineTime);
end
```

See Also

`spreadbyls` | `spreadsensbyls` | `asianbyls` | `asiansensbyls` | `lookbackbyls` | `lookbacksensbyls` | `optstockbyls` | `optstocksensbyls` | `optpricebysim` | `spreadbykirk` | `spreadsensbykirk` | `spreadbybjs` | `spreadsensbybjs` | `asianbykv` | `asiansensbykv` | `asianbylevy` | `asiansensbylevy` | `lookbackbycvgsg` | `lookbacksensbycvgsg` | `optstockbyblk` | `optstocksensbyblk` | `spreadbyfd` | `spreadsensbyfd`

Related Examples

- “Pricing European and American Spread Options” on page 3-97
- “Hedging Strategies Using Spread Options” on page 4-35
- “Compute Option Prices on a Forward” on page 11-1597
- “Compute Forward Option Prices and Delta Sensitivities” on page 11-1662
- “Compute the Option Price on a Future” on page 11-1598
- “Simulating Electricity Prices with Mean-Reversion and Jump-Diffusion” on page 3-53
- “Pricing Asian Options” on page 3-110

More About

- “Forwards Option” on page 3-40
- “Futures Option” on page 3-41
- “Spread Option” on page 3-38
- “Asian Option” on page 3-34
- “Vanilla Option” on page 3-37
- “Lookback Option” on page 3-39
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

- Energy Trading & Risk Management with MATLAB (47 min 31 sec)

Simulating Electricity Prices with Mean-Reversion and Jump-Diffusion

This example shows how to simulate electricity prices using a mean-reverting model with seasonality and a jump component. The model is calibrated under the real-world probability using historical electricity prices. The market price of risk is obtained from futures prices. A risk-neutral Monte Carlo simulation is conducted using the calibrated model and the market price of risk. The simulation results are used to price a Bermudan option with electricity prices as the underlying.

Overview of the Model

Electricity prices exhibit jumps in prices at periods of high demand when additional, less efficient electricity generation methods, are brought on-line to provide a sufficient supply of electricity. In addition, they have a prominent seasonal component, along with reversion to mean levels. Therefore, these characteristics should be incorporated into a model of electricity prices [2 on page 3-62].

In this example, electricity price is modeled as:

$$log(P_t) = f(t) + X_t$$

where P_t is the spot price of electricity. The logarithm of electricity price is modeled with two components: $f(t)$ and X_t. The component $f(t)$ is the deterministic seasonal part of the model, and X_t is the stochastic part of the model. Trigonometric functions are used to model $f(t)$ as follows [3]:

$$f(t) = s_1\sin(2\pi t) + s_2\cos(2\pi t) + s_3\sin(4\pi t) + s_4\cos(4\pi t) + s_5$$

where $s_i, i = 1, \ldots, 5$ are constant parameters, and t is the annualized time factors. The stochastic component X_t is modeled as an Ornstein-Uhlenbeck process (mean-reverting) with jumps:

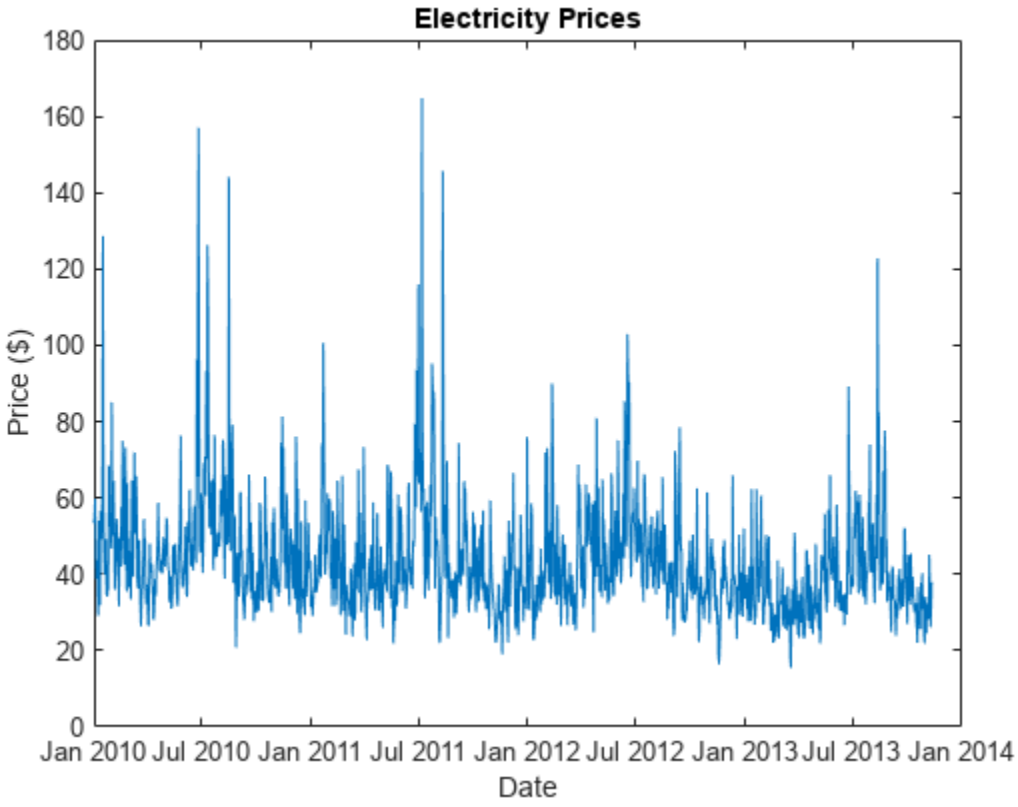
$$dX_t = (\alpha - \kappa X_t)dt + \sigma dW_t + J(\mu_J, \sigma_J)d\Pi(\lambda)$$

The parameters α and κ are the mean-reversion parameters. Parameter σ is the volatility, and W_t is a standard Brownian motion. The jump size is $J(\mu_J, \sigma_J)$, with a normally distributed mean μ_J, and a standard deviation σ_J. The Poisson process $\Pi(\lambda)$ has a jump intensity of λ.

Electricity Prices

Sample electricity prices from January 1, 2010 to November 11, 2013 are loaded and plotted below. `Prices` contain the electricity prices, and `PriceDates` contain the dates associated with the prices. The logarithm of the prices and annual time factors are calculated.

```
% Load the electricity prices and futures prices.
load('electricity_prices.mat');
PriceDates = datetime(PriceDates,'ConvertFrom','datenum');
FutExpiry = datetime(FutExpiry,'ConvertFrom','datenum');
FutValuationDate = datetime(FutValuationDate,'ConvertFrom','datenum');
% Plot the electricity prices.
figure;
plot(PriceDates, Prices);
title('Electricity Prices');
xlabel('Date');
ylabel('Price ($)');
```



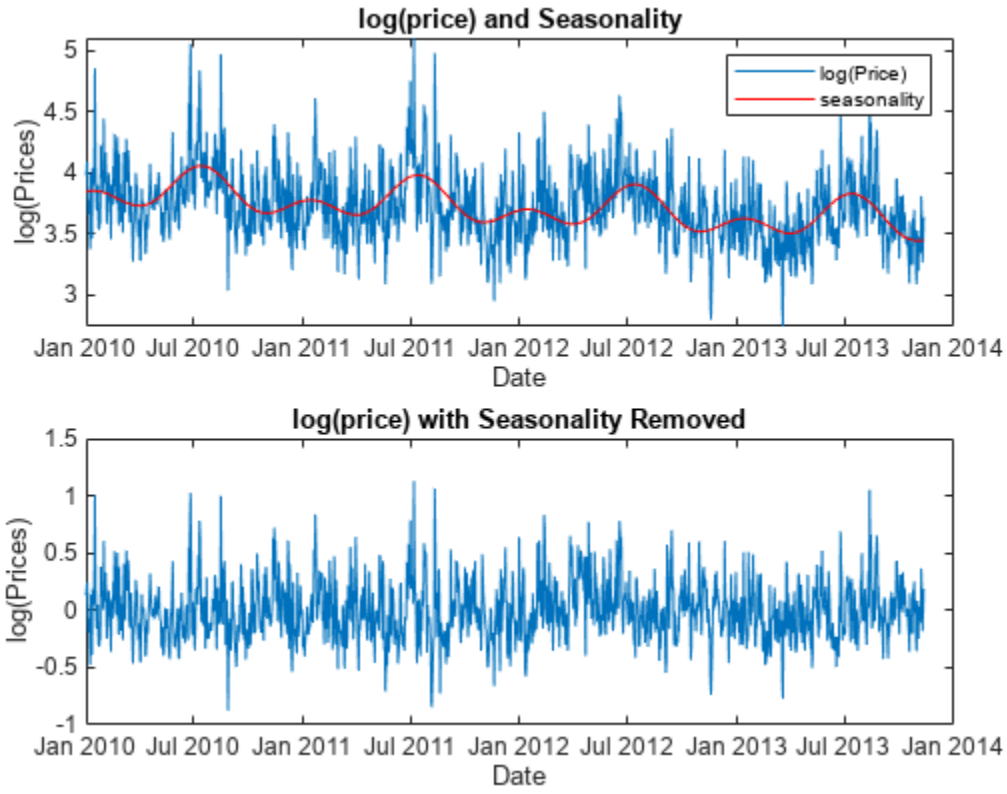

```
% Obtain the log of prices.
logPrices = log(Prices);

% Obtain the annual time factors from dates.
PriceTimes = yearfrac(PriceDates(1), PriceDates);
```

Calibration

First, the deterministic seasonality part is calibrated using the least squares method. Since the seasonality function is linear with respect to the parameters s_i, the backslash operator (`mldivide`) is used. After the calibration, the seasonality is removed from the logarithm of price. The logarithm of price and seasonality trends are plotted below. Also, the de-seasonalized logarithm of price is plotted.

```
% Calibrate parameters for the seasonality model.
seasonMatrix = @(t) [sin(2.*pi.*t) cos(2.*pi.*t) sin(4.*pi.*t) ...
    cos(4.*pi.*t) t ones(size(t, 1), 1)];
C = seasonMatrix(PriceTimes);
seasonParam = C\logPrices;

% Plot the log price and seasonality line.
figure;
subplot(2, 1, 1);
plot(PriceDates, logPrices);
title('log(price) and Seasonality');
xlabel('Date');
ylabel('log(Prices)');
```

```
hold on;
plot(PriceDates, C*seasonParam, 'r');
hold off;
legend('log(Price)', 'seasonality');

% Plot de-seasonalized log price
X = logPrices-C*seasonParam;
subplot(2, 1, 2);
plot(PriceDates, X);
title('log(price) with Seasonality Removed');
xlabel('Date');
ylabel('log(Prices)');
```



The second stage is to calibrate the stochastic part. The model for X_t needs to be discretized to conduct the calibration. To discretize, assume that there is a Bernoulli process for the jump events. That is, there is at most one jump per day since this example is calibrating against daily electricity prices. The discretized equation is:

$$X_t = \alpha\Delta t + \phi X_{t-1} + \sigma\xi$$

with probability $(1 - \lambda\Delta t)$ and,

$$X_t = \alpha\Delta t + \phi X_{t-1} + \sigma\xi + \mu_J + \sigma_J\xi_J$$

with probability $\lambda\Delta t$, where ξ and ξ_J are independent standard normal random variables, and $\phi = 1 - \kappa\Delta t$. The density function of X_t given X_{t-1} is [1 on page 3-62,4 on page 3-62]:

$$f(X_t|X_{t-1}) = (\lambda\Delta t)N_1(X_t|X_{t-1}) + (1-\lambda\Delta t)N_2(X_t|X_{t-1})$$

$$N_1(X_t|X_{t-1}) = (2\pi(\sigma^2+\sigma_J^2))^{-\frac{1}{2}}\exp(\frac{-(X_t-\alpha\Delta t-\phi X_{t-1}-\mu_J)^2}{2(\sigma^2+\sigma_J^2)})$$

$$N_2(X_t|X_{t-1}) = (2\pi\sigma^2)^{-\frac{1}{2}}\exp(\frac{-(X_t-\alpha\Delta t-\phi X_{t-1})^2}{2\sigma^2})$$

The parameters $\theta = \{\alpha, \phi, \mu_J, \sigma^2, \sigma_J^2, \lambda\}$ can be calibrated by minimizing the negative log likelihood function:

$$min_\theta - \sum_{t=1}^{T}\log(f(X_t|X_{t-1}))$$

$$subject\ to\ \phi < 1, \sigma^2 > 0, \sigma_J^2 > 0, 0 \le \lambda\Delta t \le 1$$

The first inequality constraint, $\phi < 1$, is equivalent to $\kappa > 0$. The volatilities σ and σ_J must be positive. In the last inequality, $\lambda\Delta t$ is between 0 and 1, because it represents the probability of a jump occurring in Δt time. In this example, assume that Δt is one day. Therefore, there is at most 365 jumps in one year. The `mle` function from the Statistics and Machine Learning Toolbox™ is well suited to solve the above maximum likelihood problem.

```
% Prices at t, X(t).
Pt = X(2:end);

% Prices at t-1, X(t-1).
Pt_1 = X(1:end-1);

% Discretization for the daily prices.
dt = 1/365;

% PDF for the discretized model.
mrjpdf = @(Pt, a, phi, mu_J, sigmaSq, sigmaSq_J, lambda) ...
    lambda.*exp((-(Pt-a-phi.*Pt_1-mu_J).^2)./ ...
    (2.*(sigmaSq+sigmaSq_J))).* (1/sqrt(2.*pi.*(sigmaSq+sigmaSq_J))) + ...
    (1-lambda).*exp((-(Pt-a-phi.*Pt_1).^2)/(2.*sigmaSq)).* ...
    (1/sqrt(2.*pi.*sigmaSq));

% Constraints:
% phi < 1 (k > 0)
% sigmaSq > 0
% sigmaSq_J > 0
% 0 <= lambda <= 1
lb = [-Inf -Inf -Inf 0 0 0];
ub = [Inf 1 Inf Inf Inf 1];

% Initial values.
x0 = [0 0 0 var(X) var(X) 0.5];

% Solve the maximum likelihood.
params = mle(Pt,'pdf',mrjpdf,'start',x0,'lowerbound',lb,'upperbound',ub,...
    'optimfun','fmincon');
```

```
% Obtain the calibrated parameters.
alpha = params(1)/dt
```

```
alpha = -20.1060
```

```
kappa = (1-params(2))/dt
```

```
kappa = 188.2535
```

```
mu_J = params(3)
```

```
mu_J = 0.2044
```

```
sigma = sqrt(params(4)/dt);
sigma_J = sqrt(params(5))
```

```
sigma_J = 0.2659
```

```
lambda = params(6)/dt
```

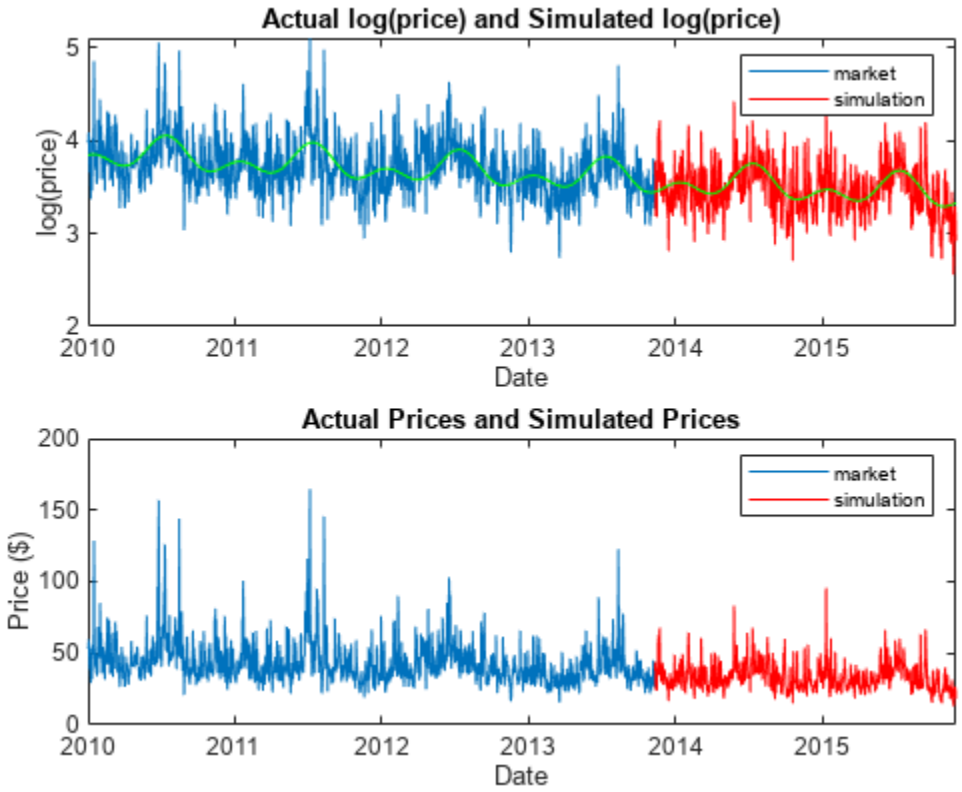
```
lambda = 98.3358
```

Monte Carlo Simulation

The calibrated parameters and the discretized model allow us to simulate electricity prices under the real-world probability. The simulation is conducted for approximately 2 years with 10,000 trials. It exceeds 2 years to include all the dates in the last month of simulation. This is because the expected simulation prices for the futures contract expiry date is required in the next section to calculate the market price of risk. The seasonality is added back on the simulated paths. A plot for a single simulation path is plotted below.

```
rng default;

% Simulate for about 2 years.
nPeriods = 365*2+20;
nTrials = 10000;
n1 = randn(nPeriods,nTrials);
n2 = randn(nPeriods, nTrials);
j = binornd(1, lambda*dt, nPeriods, nTrials);
SimPrices = zeros(nPeriods, nTrials);
SimPrices(1,:) = X(end);
for i=2:nPeriods
    SimPrices(i,:) = alpha*dt + (1-kappa*dt)*SimPrices(i-1,:) + ...
                sigma*sqrt(dt)*n1(i,:) + j(i,:).*(mu_J+sigma_J*n2(i,:));
end

% Add back seasonality.
SimPriceDates = PriceDates(end) + days(0:(nPeriods-1))';
SimPriceTimes = yearfrac(PriceDates(1), SimPriceDates);
CSim = seasonMatrix(SimPriceTimes);
logSimPrices = SimPrices + repmat(CSim*seasonParam,1,nTrials);

% Plot the logarithm of Prices and simulated logarithm of Prices.
figure;
subplot(2, 1, 1);
plot(PriceDates, logPrices);
hold on;
plot(SimPriceDates(2:end), logSimPrices(2:end,1), 'red');
seasonLine = seasonMatrix([PriceTimes; SimPriceTimes(2:end)])*seasonParam;
```

```
plot([PriceDates; SimPriceDates(2:end)], seasonLine, 'green');
hold off;
title('Actual log(price) and Simulated log(price)');
xlabel('Date');
ylabel('log(price)');
legend('market', 'simulation');

% Plot the prices and simulated prices.
PricesSim = exp(logSimPrices);
subplot(2, 1, 2);
plot(PriceDates, Prices);
hold on;
plot(SimPriceDates, PricesSim(:,1), 'red');
hold off;
title('Actual Prices and Simulated Prices');
xlabel('Date');
ylabel('Price ($)');
legend('market', 'simulation');
```



Calibration of the Market Price of Risk

Up to this point, the parameters were calibrated under the real-world probability. However, to price options, you need the simulation under the risk-neutral probability. To obtain this, calculate the market price of risk from futures prices to derive the risk-neutral parameters. Suppose that there are monthly futures contracts available on the market, which are settled daily during the contract month. For example, such futures for the PJM electricity market are listed on the Chicago Mercantile Exchange [5 on page 3-62].

The futures are settled daily during the contract month. Therefore, you can obtain daily futures values by assuming the futures value is constant for the contract month. The expected futures prices from the real-world measure are also needed to calculate the market price of risk. This can be obtained from the simulation conducted in the previous section.

```
% Obtain the daily futures prices.
FutPricesDaily = zeros(size(SimPriceDates));
for i=1:nPeriods
    idx = find(year(SimPriceDates(i)) == year(FutExpiry) & ...
        month(SimPriceDates(i)) == month(FutExpiry));
    FutPricesDaily(i) = FutPrices(idx);
end

% Calculate the expected futures price under real-world measure.
SimPricesExp = mean(PricesSim, 2);
```

To calibrate the market price of risk against market futures values, use the following equation:

$$\log(\frac{F_t}{E_t}) = -\sigma e^{-kt}\int_0^t e^{ks} m_s ds$$

where F_t is the observed futures value at time t, and E_t is the expected value under the real-world measure at time t. The equation was obtained using the same methodology as described in [3 on page 3-62]. This example assumes that the market price of risk is fully driven by the Brownian motion. The market price of risk, m_t, can be solved by discretizing the above equation and solving a system of linear equations.

```
% Setup system of equations.
t0 = yearfrac(PriceDates(1), FutValuationDate);
tz = SimPriceTimes-t0;
b = -log(FutPricesDaily(2:end) ./ SimPricesExp(2:end)) ./ ...
    (sigma.*exp(-kappa.*tz(2:end)));
A = (1/kappa).*(exp(kappa.*tz(2:end)) - exp(kappa.*tz(1:end-1)));
A = tril(repmat(A', size(A,1), 1));

% Precondition to stabilize numerical inversion.
P = diag(1./diag(A));
b = P*b;
A = P*A;

% Solve for the market price of risk.
riskPremium = A\b;
```

Simulation of Risk-Neutral Prices

Once m_t is obtained, risk-neutral simulation can be conducted using the following dynamics:

$$X_t = \alpha\Delta t + \phi X_{t-1} - \sigma m_{t-1}\Delta t + \sigma\xi$$

with probability $(1 - \lambda\Delta t)$ and

$$X_t = \alpha\Delta t + \phi X_{t-1} - \sigma m_{t-1}\Delta t + \sigma\xi + \mu_J + \sigma_J\xi_J$$

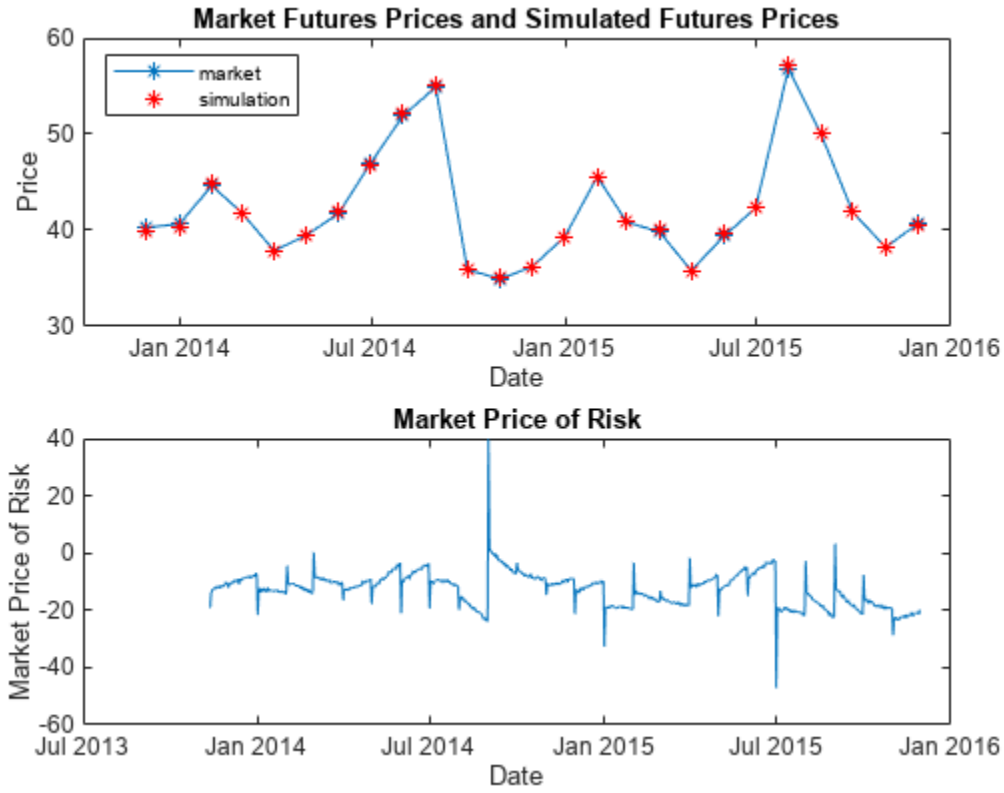
with probability $\lambda\Delta t$.

```
nTrials = 10000;
n1 = randn(nPeriods, nTrials);
```

```
n2 = randn(nPeriods, nTrials);
j = binornd(1, lambda*dt, nPeriods, nTrials);

SimPrices = zeros(nPeriods, nTrials);
SimPrices(1,:) = X(end);
for i=2:nPeriods
    SimPrices(i,:) = alpha*dt + (1-kappa*dt)*SimPrices(i-1,:) + ...
        sigma*sqrt(dt)*n1(i,:) - sigma*dt*riskPremium(i-1) + ...
        j(i,:).*(mu_J+sigma_J*n2(i,:));
end

% Add back seasonality.
CSim = seasonMatrix(SimPriceTimes);
logSimPrices = SimPrices + repmat(CSim*seasonParam,1,nTrials);

% Convert the log(Price) to Price.
PricesSim = exp(logSimPrices);
```

The expected values from the risk-neutral simulation are plotted against the market futures values. This confirms that the risk-neutral simulation closely reproduces the market futures values.

```
% Obtain expected values from the risk-neutral simulation.
SimPricesExp = mean(PricesSim,2);
fexp = zeros(size(FutExpiry));
for i = 1:size(FutExpiry,1)
    idx = SimPriceDates == FutExpiry(i);
    if sum(idx)==1
        fexp(i) = SimPricesExp(idx);
    end
end

% Plot expected values from the simulation against market futures prices.
figure;
subplot(2,1,1);
plot(FutExpiry, FutPrices(1:size(FutExpiry,1)),'-*');
hold on;
plot(FutExpiry, fexp, '*r');
hold off;
title('Market Futures Prices and Simulated Futures Prices');
xlabel('Date');
ylabel('Price');
legend('market', 'simulation', 'Location', 'NorthWest');
subplot(2,1,2);
plot(SimPriceDates(2:end), riskPremium);
title('Market Price of Risk');
xlabel('Date');
ylabel('Market Price of Risk');
```



Pricing a Bermudan Option

The risk-neutral simulated values are used as input into the function `optpricebysim` in the Financial Instruments Toolbox™ to price a European, Bermudan, or American option on electricity prices. Below, the price is calculated for a two-year Bermudan call option with two exercise opportunities. The first exercise is after one year, and the second is at the maturity of the option.

```
% Settle, maturity of option.
Settle = FutValuationDate;
Maturity = FutValuationDate + calyears(2);

% Create the interest-rate term structure.
riskFreeRate = 0.01;
Basis = 0;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
    'EndDates', Maturity, 'Rate', riskFreeRate, 'Compounding', ...
    Compounding, 'Basis', Basis);

% Cutoff the simulation at maturity.
endIdx = find(SimPriceDates == Maturity);
SimPrices = PricesSim(1:endIdx,:);
Times = SimPriceTimes(1:endIdx) - SimPriceTimes(1);

% Bermudan call option with strike 60, two exercise opportunities, after
% one year and at maturity.
OptSpec = 'call';
```

```
Strike = 60;
ExerciseTimes = [Times(366) Times(end)];
Price = optpricebysim(RateSpec, SimPrices, Times, OptSpec, Strike, ...
    ExerciseTimes)
```

```
Price = 1.1085
```

References

[1] Escribano, Alvaro, Pena, Juan Ignacio, Villaplana, Pablo. "Modeling Electricity Prices: International Evidence." Universidad Carloes III de Madrid, Working Paper 02-27, 2002.

[2] Lucia, Julio J., Schwartz, Eduaro. "Electricity Prices and Power Derivatives: Evidence from the Nordic Power Exchange." *Review of Derivatives Research*. Vol. 5, Issue 1, pp 5-50, 2002.

[3] Seifert, Jan, Uhrig-Homburg, Marliese. "Modelling Jumps in Electricity Prices: Theory and Empirical Evidence." *Review of Derivatives Research*. Vol. 10, pp 59-85, 2007.

[4] Villaplana, Pablo. "Pricing Power Derivatives: A Two-Factor Jump-Diffusion Approach." Universidad Carloes III de Madrid, Working Paper 03-18, 2003.

[5] https://www.cmegroup.com

See Also

`spreadbyls` | `spreadsensbyls` | `asianbyls` | `asiansensbyls` | `lookbackbyls` | `lookbacksensbyls` | `optstockbyls` | `optstocksensbyls` | `optpricebysim` | `spreadbykirk` | `spreadsensbykirk` | `spreadbybjs` | `spreadsensbybjs` | `asianbykv` | `asiansensbykv` | `asianbylevy` | `asiansensbylevy` | `lookbackbycvgsg` | `lookbacksensbycvgsg` | `optstockbyblk` | `optstocksensbyblk` | `spreadbyfd` | `spreadsensbyfd`

Related Examples

- “Pricing European and American Spread Options” on page 3-97
- “Hedging Strategies Using Spread Options” on page 4-35
- “Pricing Swing Options Using the Longstaff-Schwartz Method” on page 3-43
- “Compute Option Prices on a Forward” on page 11-1597
- “Compute Forward Option Prices and Delta Sensitivities” on page 11-1662
- “Compute the Option Price on a Future” on page 11-1598
- “Pricing Asian Options” on page 3-110

More About

- “Forwards Option” on page 3-40
- “Futures Option” on page 3-41
- “Spread Option” on page 3-38

- “Asian Option” on page 3-34
- “Vanilla Option” on page 3-37
- “Lookback Option” on page 3-39
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

- Energy Trading & Risk Management with MATLAB (47 min 31 sec)

Pricing 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, an implied trinomial price tree, or a standard trinomial 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)
```

The syntax for `sttprice` is:

```
[Price, PriceTree] = sttprice(STTTree, InstSet, Name, Value)
```

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`.

`STTTree` is a standard trinomial equity price tree created using `stttree`. See “Building Equity Binary Trees” on page 3-3 and “Building Implied Trinomial Trees” on page 3-6 to learn how to create these structures.

`InstSet` is a structure that represents the set of instruments to be priced independently using the model.

Optional Argument

You can enter a third optional argument, `Options`, used when pricing barrier options. For more specific information, see “Pricing Options Structure” on page A-2.

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, ITT, and STT 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             27344  struct
  BDTTree           1x1              7322  struct
  BKInstSet         1x1             27334  struct
  BKTree            1x1              8532  struct
  CRRInstSet        1x1             21066  struct
  CRRTree           1x1              7086  struct
  EQPInstSet        1x1             21066  struct
  EQPTree           1x1              7086  struct
  HJMInstSet        1x1             27336  struct
  HJMTree           1x1              8334  struct
  HWInstSet         1x1             27334  struct
  HWTree            1x1              8532  struct
  ITTInstSet        1x1             21070  struct
  ITTTree           1x1             12660  struct
  STTInstSet        1x1             21070  struct
  STTTree           1x1              7782  struct
  ZeroInstSet       1x1             17458  struct
  ZeroRateSpec      1x1              2152  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 UStrike USettle        UExerciseDates UAmericanOpt COptSpec CStrike CSettle        CExerciseDates CAmericanOpt N
```

```
4     Compound call     130     01-Jan-2003    01-Jan-2006    1            put      5       01-Jan-2003    01-Jan-2005    1           C

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             27344  struct
  BDTTree           1x1              7322  struct
  BKInstSet         1x1             27334  struct
  BKTree            1x1              8532  struct
  CRRInstSet        1x1             21066  struct
  CRRTree           1x1              7086  struct
  EQPInstSet        1x1             21066  struct
  EQPTree           1x1              7086  struct
  HJMInstSet        1x1             27336  struct
  HJMTree           1x1              8334  struct
```

```
  HWInstSet         1x1            27334  struct
  HWTree            1x1             8532  struct
  ITTInstSet        1x1            21070  struct
  ITTTree           1x1            12660  struct
  STTInstSet        1x1            21070  struct
  STTTree           1x1             7782  struct
  ZeroInstSet       1x1            17458  struct
  ZeroRateSpec      1x1             2152  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 UStrike USettle        UExerciseDates UAmericanOpt COptSpec CStrike CSettle        CExerciseDates CAmericanOpt N
4     Compound call     130     01-Jan-2003    01-Jan-2006    1            put      5       01-Jan-2003    01-Jan-2005    1            0

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

>> 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 UStrike USettle        UExerciseDates UAmericanOpt COptSpec CStrike CSettle        CExerciseDates CAmericanOpt N
4     Compound call     130     01-Jan-2003    01-Jan-2006    1            put      5       01-Jan-2003    01-Jan-2005    1            0

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             27344  struct
  BDTTree           1x1              7322  struct
  BKInstSet         1x1             27334  struct
  BKTree            1x1              8532  struct
  CRRInstSet        1x1             21066  struct
  CRRTree           1x1              7086  struct
  EQPInstSet        1x1             21066  struct
  EQPTree           1x1              7086  struct
  HJMInstSet        1x1             27336  struct
  HJMTree           1x1              8334  struct
  HWInstSet         1x1             27334  struct
  HWTree            1x1              8532  struct
  ITTInstSet        1x1             21070  struct
  ITTTree           1x1             12660  struct
  STTInstSet        1x1             21070  struct
  STTTree           1x1              7782  struct
  ZeroInstSet       1x1             17458  struct
  ZeroRateSpec      1x1              2152  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 N
4     Compound call     99      01-Jan-2006    01-Jan-2010    1            put      5       01-Jan-2006    01-Jan-2010    1            (

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 =

    1.6506
   10.6832
    2.4074
    3.2294
    0.5426
    6.1845
    3.2052
    6.6074
```

Computing Prices Using STT

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             27344  struct
  BDTTree           1x1              7322  struct
  BKInstSet         1x1             27334  struct
  BKTree            1x1              8532  struct
  CRRInstSet        1x1             21066  struct
  CRRTree           1x1              7086  struct
  EQPInstSet        1x1             21066  struct
  EQPTree           1x1              7086  struct
  HJMInstSet        1x1             27336  struct
  HJMTree           1x1              8334  struct
  HWInstSet         1x1             27334  struct
  HWTree            1x1              8532  struct
  ITTInstSet        1x1             21070  struct
  ITTTree           1x1             12660  struct
  STTInstSet        1x1             21070  struct
  STTTree           1x1              7782  struct
  ZeroInstSet       1x1             17458  struct
  ZeroRateSpec      1x1              2152  struct
```

`STTTree` and `STTInstSet` are the input arguments required to call the function `sttprice`. Use the command `instdisp` to examine the set of instruments contained in the variable `STTInstSet`.

```
instdisp(STTInstSet)
```

```
Index Type     OptSpec Strike Settle         ExerciseDates  AmericanOpt Name  Quantity
1     OptStock call    100    01-Jan-2009    01-Jan-2011    1           Call1 10
2     OptStock put      80    01-Jan-2009    01-Jan-2012    0           Put1   5

Index Type    OptSpec Strike Settle         ExerciseDates  AmericanOpt BarrierSpec Barrier Rebate Name     Quantity
3     Barrier call    105    01-Jan-2009    01-Jan-2012    1           ui          115     0      Barrier1 1

Index Type     UOptSpec UStrike USettle        UExerciseDates UAmericanOpt COptSpec CStrike CSettle        CExerciseDates CAmericanOpt N
4     Compound call     95      01-Jan-2009    01-Jan-2012    1            put      5       01-Jan-2009    01-Jan-2011    1           C

Index Type     OptSpec Strike Settle         ExerciseDates  AmericanOpt Name      Quantity
5     Lookback call    90     01-Jan-2009    01-Jan-2012    0           Lookback1 7
6     Lookback call    95     01-Jan-2009    01-Jan-2013    0           Lookback2 9

Index Type  OptSpec Strike Settle         ExerciseDates  AmericanOpt AvgType    AvgPrice AvgDate Name   Quantity
7     Asian call    100    01-Jan-2009    01-Jan-2012    0           arithmetic NaN      NaN     Asian1 4
8     Asian call    100    01-Jan-2009    01-Jan-2013    0           arithmetic NaN      NaN     Asian2 6
```

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 `sttprice`.

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

```
Price = sttprice(STTTree, STTInstSet)
```

```
Price =

    4.5025
    3.0603
    3.7977
    1.7090
   11.7296
   12.9120
    1.6905
    2.6203
```

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
```

So, 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
```

So, in the `Price` vector, the first element, 1.650, represents the price of the first instrument (`Call1`), and the eighth elements, 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}

ans =

    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. So, 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
```

Graphical Representation of Equity Derivative 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, EQP, and LR trees are equivalent to Black-Derman-Toy (BDT) trees, given that they are all binary recombining trees. The graphical representations of ITT and STT trees are equivalent to Hull-White (HW) trees, given that they are all trinomial recombining trees. See “Graphical Representation of Trees” on page 2-219 for an overview on the use of `treeviewer` with CRR trees, EQP trees, LR trees, ITT trees, and STT trees and their corresponding option price trees. Follow the instructions for BDT trees.

See Also

`crrtree` | `eqptree` | `lrtree` | `stockspec` | `crrtimespec` | `eqptimespec` | `lrtimespec` | `itttree` | `itttimespec` | `treepath` | `trintreepath` | `asianbycrr` | `barrierbycrr` | `compoundbycrr` | `crrprice` | `crrsens` | `lookbackbycrr` | `optstockbycrr` | `instasian` | `instbarrier` | `instcompound` | `instlookback` | `instoptstock` | `asianbyeqp` | `barrierbyeqp` | `compoundbyeqp` | `eqpprice` | `eqpsens` | `lookbackbyeqp` | `optstockbyeqp` | `optstockbylr` | `optstocksensbylr` | `asianbyitt` | `barrierbyitt` | `compoundbyitt` | `ittprice` | `ittsens` | `lookbackbyitt` | `optstockbyitt` | `asianbystt` | `barrierbystt` | `compoundbystt` | `sttprice` | `sttsens` | `lookbackbystt` | `optstockbystt`

Related Examples

- “Understanding Equity Trees” on page 3-2
- “Computing Equity Instrument Sensitivities” on page 3-75
- “Creating Instruments or Properties” on page 1-16
- “Graphical Representation of Equity Derivative Trees” on page 3-73
- “Pricing European Call Options Using Different Equity Models” on page 3-88

- “Pricing Asian Options” on page 3-110
- “Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194

More About

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Computing Equity 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`, `ittsens`, and `sttsens` 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, `ITTTree` and `ITTInstSet` for ITT, and `STTTree` and `STTInstSet` for STT).

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 “Pricing Options Structure” on page A-2 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`, `ittprice`, and `sttprice`. For the other options (stock option, barrier, and compound), delta and gamma are computed from the CRR, EQP, ITT, and STT 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', 'fininst: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', 'fininst: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 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 (line 680)
  In itttree (line 285)
  In stocktreesens>stocktreevega (line 193)
  In stocktreesens (line 94)
  In ittsens (line 79)

Delta =

          0.24
         -0.43
          0.35
         -0.07
          0.63
          0.60
          0.32
          0.67


Gamma =

          0.03
          0.02
          0.04
          0.00
     142945.66
      22703.21
    -142945.66
     -22703.21


Vega =

         19.35
         49.69
         12.29
          6.73
         38.90
         68.92
         18.48
         17.75


Price =

          1.65
         10.68
          2.41
          3.23
          0.54
          6.18
          3.21
          6.61
```

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.

STT Sensitivities Example

The calling syntax for the sensitivity function is:

```
[Delta, Gamma, Vega, Price] = sttsens(STTTree, InstSet, Name, Value)
```

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

```
load deriv.mat
[Delta, Gamma, Vega, Price] = sttsens(STTTree, STTInstSet);
```

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

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

All =

          0.53          0.02         52.90          4.50
         -0.09          0.00         42.44          3.06
          0.47          0.03         25.98          3.80
         -0.06          0.00         -9.53          1.71
          0.23    -186495.25         70.38         11.73
          0.33    -191186.43         92.92         12.91
          0.57     186495.25         25.81          1.69
          0.66     191186.43         37.88          2.62
```

See Also

crrtree | eqptree | lrtree | stockspec | crrtimespec | eqptimespec | lrtimespec | itttree | itttimespec | treepath | trintreepath | asianbycrr | barrierbycrr | compoundbycrr | crrprice | crrsens | lookbackbycrr | optstockbycrr | instasian | instbarrier | instcompound | instlookback | instoptstock | asianbyeqp | barrierbyeqp | compoundbyeqp | eqpprice | eqpsens | lookbackbyeqp | optstockbyeqp | optstockbylr | optstocksensbylr | asianbyitt | barrierbyitt | compoundbyitt | ittprice | ittsens | lookbackbyitt | optstockbyitt | asianbystt | barrierbystt | compoundbystt | sttprice | sttsens | lookbackbystt | optstockbystt

Related Examples

- “Understanding Equity Trees” on page 3-2
- “Pricing Equity Derivatives Using Trees” on page 3-64
- “Graphical Representation of Equity Derivative Trees” on page 3-73
- “Creating Instruments or Properties” on page 1-16
- “Graphical Representation of Equity Derivative Trees” on page 3-73
- “Pricing European Call Options Using Different Equity Models” on page 3-88
- “Pricing Asian Options” on page 3-110

More About

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Equity Derivatives Using Closed-Form Solutions

In this section...

Introduction

Financial Instruments Toolbox 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 Instruments 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 “Pricing Using the Black-Scholes Model” on page 3-82.

Black Model

Use the Black model for pricing European options on physical commodities, forwards or futures. The Black model supported by Financial Instruments 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 “Pricing Using the Black Model” on page 3-83.

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 “Pricing Using the Roll-Geske-Whaley Model” on page 3-84.

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`
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 “Pricing Using the Bjerksund-Stensland Model” on page 3-84.

Barone-Adesi-Whaley Model

The Barone-Adesi-Whaley model is used for pricing American vanilla options. Closed-form solutions for a Barone-Adesi-Whaley model support the following tasks.

Task	Function
Calculate the prices of an American call and put options using the Barone-Adesi-Whaley approximation model.	`optstockbybaw`
Calculate the prices and sensitivities of an American call and put options using the Barone-Adesi-Whaley approximation model.	`optstocksensbybaw`
Calculate the implied volatility for American options using the Barone-Adesi-Whaley model.	`impvbybaw`

For an example using the Barone-Adesi-Whaley model, see “Compute American Option Prices Using the Barone-Adesi and Whaley Option Pricing Model” on page 3-86.

Pricing 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 that 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 character vectors, 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 character vectors 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
```

Pricing 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
```

Pricing 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 that 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).

Pricing 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 that 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 character vectors, 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 character vectors 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` are 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
```

For more information on the Bjerksund-Stensland model, see “Closed-Form Solutions Modeling” on page B-3.

Compute American Option Prices Using the Barone-Adesi and Whaley Option Pricing Model

Consider an American call option with an exercise price of $120. The option expires on Jan 1, 2018. The stock has a volatility of 14% per annum, and the annualized continuously compounded risk-free rate is 4% per annum as of Jan 1, 2016. Using this data, calculate the price of the American call, assuming the price of the stock is $125 and pays a dividend of 2%.

```
StartDate  = datetime(2016,1,1);
EndDate = datetime(2018,1,1);
Basis = 1;
Compounding = -1;
Rates = 0.04;
```

Define the `RateSpec`.

```
RateSpec = intenvset('ValuationDate',StartDate,'StartDate',StartDate,'EndDate',EndDate, ...
'Rates',Rates,'Basis',Basis,'Compounding',Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9231
            Rates: 0.0400
         EndTimes: 2
       StartTimes: 0
         EndDates: 737061
       StartDates: 736330
    ValuationDate: 736330
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec`.

```
Dividend = 0.02;
AssetPrice = 125;
Volatility = 0.14;

StockSpec = stockspec(Volatility,AssetPrice,'Continuous',Dividend)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1400
         AssetPrice: 125
       DividendType: {'continuous'}
    DividendAmounts: 0.0200
    ExDividendDates: []
```

Define the American option.

```
OptSpec = 'call';
Strike = 120;
Settle = datetime(2016,1,1);
Maturity = datetime(2018,1,1);
```

Compute the price for the American option.

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

```
Price = 14.5180
```

See Also

`assetbybls` | `assetsensbybls` | `cashbybls` | `cashsensbybls` | `chooserbybls` | `gapbybls` | `gapsensbybls` | `impvbybls` | `optstockbybls` | `optstocksensbybls` | `supersharebybls` | `supersharesensbybls` | `impvbyblk` | `optstockbyblk` | `optstocksensbyblk` | `impvbyrgw` | `optstockbyrgw` | `optstocksensbyrgw` | `impvbybjs` | `optstockbybjs` | `optstocksensbybjs` | `spreadbybjs` | `spreadsensbybjs` | `basketbyju` | `basketsensbyju` | `basketstockspec` | `maxassetbystulz` | `maxassetsensbystulz` | `minassetbystulz` | `minassetsensbystulz` | `spreadbykirk` | `spreadsensbykirk` | `asianbykv` | `asiansensbykv` | `asianbylevy` | `asiansensbylevy` | `lookbackbycvgsg` | `lookbacksensbycvgsg` | `basketbyls` | `basketsensbyls` | `basketstockspec` | `asianbyls` | `asiansensbyls` | `lookbackbyls` | `lookbacksensbyls` | `spreadbyls` | `spreadsensbyls` | `optstockbyls` | `optstocksensbyls` | `optpricebysim` | `optstockbybaw` | `optstocksensbybaw`

Related Examples

- “Pricing European Call Options Using Different Equity Models” on page 3-88
- “Compute the Option Price on a Future” on page 3-95
- “Pricing European Call Options Using Different Equity Models” on page 3-88
- “Pricing Asian Options” on page 3-110

More About

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Pricing European Call Options Using Different Equity Models

This example illustrates how the Financial Instruments Toolbox™ is used to price European vanilla call options using different equity models.

The example compares call option prices using the Cox-Ross-Rubinstein model, the Leisen-Reimer model and the Black-Scholes closed formula.

Define the Call Instrument

Consider a European call option, with an exercise price of $30 on January 1, 2010. The option expires on Sep 1, 2010. Assume that the underlying stock provides no dividends. The stock is trading at $25 and has a volatility of 35% per annum. The annualized continuously compounded risk-free rate is 1.11% per annum.

```
% Option
Settle = 'Jan-01-2010';
Maturity = 'Sep-01-2010';
Strike = 30;
OptSpec = 'call';

% Stock
AssetPrice = 25;
Sigma = .35;
```

Create the Interest Rate Term Structure

```
StartDates = '01 Jan 2010';
EndDates =   '01 Jan 2013';
Rates = 0.0111;
ValuationDate = '01 Jan 2010';
Compounding = -1;

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

Create the Stock Structure

Suppose we want to create two scenarios. The first one assumes that `AssetPrice` is currently $25, the option is out of the money (OTM). The second scenario assumes that the option is at the money (ATM), and therefore `AssetPriceATM = 30`.

```
AssetPriceATM = 30;

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

Price the Options Using the Black-Scholes Closed Formula

Use the function `optstockbybls` in the Financial Instruments Toolbox to compute the price of the European call options.

```
% Price the option with AssetPrice = 25
PriceBLS = optstockbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike);
```

```
% Price the option with AssetPrice = 30
PriceBLSATM = optstockbybls(RateSpec, StockSpecATM, Settle, Maturity, OptSpec, Strike);
```

Build the Cox-Ross-Rubinstein Tree

```
% Create the time specification of the tree
NumPeriods = 15;

CRRTimeSpec = crrtimespec(ValuationDate, Maturity, NumPeriods);

% Build the tree
CRRTree = crrtree(StockSpec, RateSpec, CRRTimeSpec);
CRRTreeATM = crrtree(StockSpecATM, RateSpec, CRRTimeSpec);
```

Build the Leisen-Reimer Tree

```
% Create the time specification of the tree
LRTimeSpec = lrtimespec(ValuationDate, Maturity, NumPeriods);

% Use the default method 'PP1' (Peizer-Pratt method 1 inversion)to build
% the tree
LRTree = lrtree(StockSpec, RateSpec, LRTimeSpec, Strike);
LRTreeATM = lrtree(StockSpecATM, RateSpec, LRTimeSpec, Strike);
```

Price the Options Using the Cox-Ross-Rubinstein (CRR) Model

```
PriceCRR = optstockbycrr(CRRTree, OptSpec, Strike, Settle, Maturity);
PriceCRRATM = optstockbycrr(CRRTreeATM, OptSpec, Strike, Settle, Maturity);
```

Price the Options Using the Leisen-Reimer (LR) Model

```
PriceLR = optstockbylr(LRTree, OptSpec, Strike, Settle, Maturity);
PriceLRATM = optstockbylr(LRTreeATM, OptSpec, Strike, Settle, Maturity);
```

Compare BLS, CRR and LR Results

```
sprintf('PriceBLS: \t%f\nPriceCRR: \t%f\nPriceLR:\t%f\n', PriceBLS, ...
    PriceCRR, PriceLR)

ans =
    'PriceBLS:     1.275075
     PriceCRR:     1.294979
     PriceLR:    1.275838
     '



sprintf('\t== ATM ==\nPriceBLS ATM: \t%f\nPriceCRR ATM: \t%f\nPriceLR ATM:\t%f\n', PriceBLSATM,
    PriceCRRATM,   PriceLRATM)

ans =
    '    == ATM ==
     PriceBLS ATM:     3.497891
     PriceCRR ATM:     3.553938
     PriceLR ATM:    3.498571
     '
```

Convergence of CRR and LR Models to a BLS Solution

The following tables compare call option prices using the CRR and LR models against the results obtained with the Black-Scholes formula.

While the CRR binomial model and the Black-Scholes model converge as the number of time steps gets large and the length of each step gets small, this convergence, except for at the money options, is anything but smooth or uniform.

The tables below show that the Leisen-Reimer model reduces the size of the error with even as few steps of 45.

```
Strike = 30, Asset Price = 30

-------------------------------------

#Steps LR CRR

15 3.4986 3.5539

25 3.4981 3.5314

45 3.4980 3.5165

65 3.4979 3.5108

85 3.4979 3.5077

105 3.4979 3.5058

201 3.4979 3.5020

501 3.4979 3.4996

999 3.4979 3.4987

Strike = 30, Asset Price = 25

-------------------------------------

#Steps LR CRR

15 1.2758 1.2950

25 1.2754 1.2627

45 1.2751 1.2851

65 1.2751 1.2692

85 1.2751 1.2812

105 1.2751 1.2766

201 1.2751 1.2723

501 1.2751 1.2759
```

```
999 1.2751 1.2756
```

Analyze the Effect of the Number of Periods on the Price of the Options

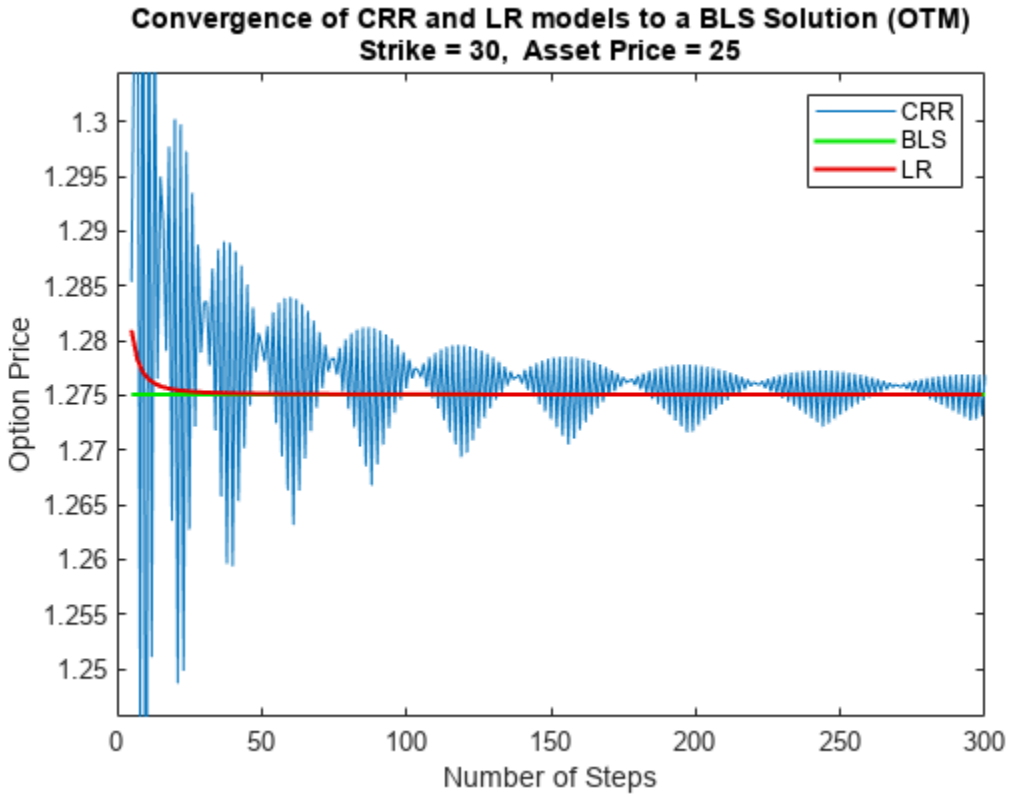
The following graphs show how convergence changes as the number of steps in the binomial calculation increases, as well as, the impact on convergence to changes to the stock price. Observe that the Leisen-Reimer model removes the oscillation and produces estimates close to the Black-Scholes model using only a small number of steps.

```
NPoints = 300;

% Cox-Ross-Rubinstein
NumPeriodCRR  = 5 : 1 : NPoints;
NbStepCRR     = length(NumPeriodCRR);
PriceCRR = nan(NbStepCRR, 1);
PriceCRRATM = PriceCRR;

for i = 1 : NbStepCRR
    CRRTimeSpec = crrtimespec(ValuationDate, Maturity, NumPeriodCRR(i));
    CRRT = crrtree(StockSpec, RateSpec, CRRTimeSpec);
    PriceCRR(i) = optstockbycrr(CRRT, OptSpec, Strike,ValuationDate, Maturity) ;

    CRRTATM = crrtree(StockSpecATM, RateSpec, CRRTimeSpec);
    PriceCRRATM(i) = optstockbycrr(CRRTATM, OptSpec, Strike,ValuationDate, Maturity) ;
end


% Now with Leisen-Reimer
NumPeriodLR  = 5 : 2 : NPoints;
NbStepLR     = length(NumPeriodLR);
PriceLR = nan(NbStepLR, 1);
PriceLRATM = PriceLR;

for i = 1 : NbStepLR
    LRTimeSpec = lrtimespec(ValuationDate, Maturity, NumPeriodLR(i));
    LRT = lrtree(StockSpec, RateSpec, LRTimeSpec, Strike);
    PriceLR(i) = optstockbylr(LRT, OptSpec, Strike,ValuationDate, Maturity) ;

    LRTATM = lrtree(StockSpecATM, RateSpec, LRTimeSpec, Strike);
    PriceLRATM(i) = optstockbylr(LRTATM, OptSpec, Strike,ValuationDate, Maturity) ;
end
```

First scenario: Out of the Money call option

```
% For Cox-Ross-Rubinstein
plot(NumPeriodCRR, PriceCRR);
hold on;
plot(NumPeriodCRR, PriceBLS*ones(NbStepCRR,1),'Color',[0 0.9 0], 'linewidth', 1.5);

% For Leisen-Reimer
plot(NumPeriodLR, PriceLR, 'Color',[0.9 0 0], 'linewidth', 1.5);

% Concentrate in the area of interest by clipping on the Y axis at 5x the
% LR Price:
YLimDelta = 5*abs(PriceLR(1) - PriceBLS);
ax = gca;
ax.YLim = [PriceBLS-YLimDelta PriceBLS+YLimDelta];
```

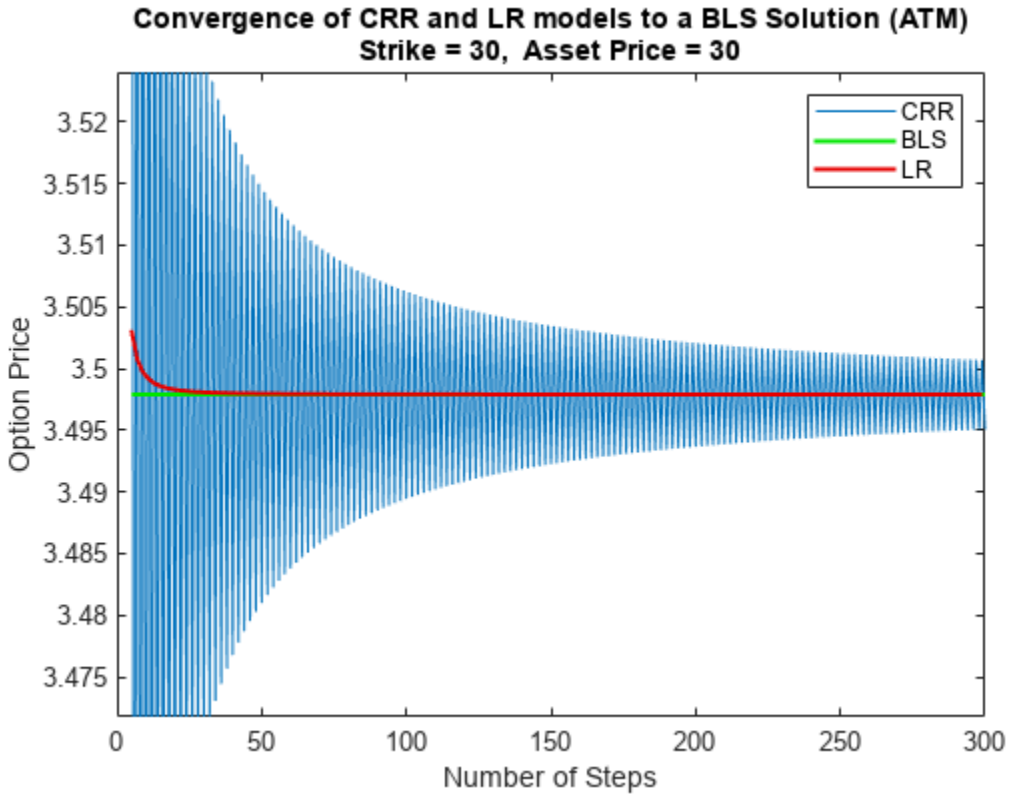
```
% Annotate Plot
titleString = sprintf('\nConvergence of CRR and LR models to a BLS Solution (OTM)\nStrike = %d,
title(titleString)
ylabel('Option Price')
xlabel('Number of Steps')
legend('CRR', 'BLS', 'LR', 'Location', 'NorthEast')
```



Second scenario: At the Money call option

```
% For Cox-Ross-Rubinstein
figure;
plot(NumPeriodCRR, PriceCRRATM);
hold on;
plot(NumPeriodCRR, PriceBLSATM*ones(NbStepCRR,1),'Color',[0 0.9 0], 'linewidth', 1.5);

% For Leisen-Reimer
plot(NumPeriodLR, PriceLRATM, 'Color',[0.9 0 0], 'linewidth', 1.5);

% Concentrate in the area of interest by clipping on the Y axis at 5x the
% LR Price:
YLimDelta = 5*abs(PriceLRATM(1) - PriceBLSATM);
ax = gca;
ax.YLim = [PriceBLSATM-YLimDelta PriceBLSATM+YLimDelta];
% Annotate Plot
titleString = sprintf('\nConvergence of CRR and LR models to a BLS Solution (ATM)\nStrike = %d,
title(titleString)
ylabel('Option Price')
```

```
xlabel('Number of Steps')
legend('CRR', 'BLS', 'LR', 'Location', 'NorthEast')
```



See Also

assetbybls | assetsensbybls | cashbybls | cashsensbybls | chooserbybls | gapbybls | gapsensbybls | impvbybls | optstockbybls | optstocksensbybls | supersharebybls | supersharesensbybls | impvbyblk | optstockbyblk | optstocksensbyblk | impvbyrgw | optstockbyrgw | optstocksensbyrgw | impvbybjs | optstockbybjs | optstocksensbybjs | spreadbybjs | spreadsensbybjs | basketbyju | basketsensbyju | basketstockspec | maxassetbystulz | maxassetsensbystulz | minassetbystulz | minassetsensbystulz | spreadbykirk | spreadsensbykirk | asianbykv | asiansensbykv | asianbylevy | asiansensbylevy | lookbackbycvgsg | lookbacksensbycvgsg | basketbyls | basketsensbyls | basketstockspec | asianbyls | asiansensbyls | lookbackbyls | lookbacksensbyls | spreadbyls | spreadsensbyls | optstockbyls | optstocksensbyls | optpricebysim | optstocksensbybaw

Related Examples

- “Equity Derivatives Using Closed-Form Solutions” on page 3-79
- “Compute the Option Price on a Future” on page 3-95
- “Pricing Asian Options” on page 3-110

More About

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Compute the Option Price on a Future

Consider a call European option on the Crude Oil Brent futures. The option expires on December 1, 2014 with an exercise price of $120. Assume that on April 1, 2014 futures price is at $105, the annualized continuously compounded risk-free rate is 3.5% per annum and volatility is 22% per annum. Using this data, compute the price of the option.

Define the `RateSpec`.

```
ValuationDate = datetime(2014,1,1);
EndDates = datetime(2015,1,1);
Rates = 0.035;
Compounding = -1;
Basis = 1;
RateSpec  = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate,...
'EndDates', EndDates, 'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis')
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9656
            Rates: 0.0350
         EndTimes: 1
       StartTimes: 0
         EndDates: 735965
       StartDates: 735600
    ValuationDate: 735600
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec`.

```
AssetPrice = 105;
Sigma = 0.22;
StockSpec  = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2200
         AssetPrice: 105
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the option.

```
Settle = datetime(2014,4,1);
Maturity = datetime(2014,12,1);
Strike = 120;
OptSpec = {'call'};
```

Price the futures call option.

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

```
Price = 2.5847
```

See Also

assetbybls | assetsensbybls | cashbybls | cashsensbybls | chooserbybls | gapbybls | gapsensbybls | impvbybls | optstockbybls | optstocksensbybls | supersharebybls | supersharesensbybls | impvbyblk | optstockbyblk | optstocksensbyblk | impvbyrgw | optstockbyrgw | optstocksensbyrgw | impvbybjs | optstockbybjs | optstocksensbybjs | spreadbybjs | spreadsensbybjs | basketbyju | basketsensbyju | basketstockspec | maxassetbystulz | maxassetsensbystulz | minassetbystulz | minassetsensbystulz | spreadbykirk | spreadsensbykirk | asianbykv | asiansensbykv | asianbylevy | asiansensbylevy | lookbackbycvgsg | lookbacksensbycvgsg | basketbyls | basketsensbyls | basketstockspec | asianbyls | asiansensbyls | lookbackbyls | lookbacksensbyls | spreadbyls | spreadsensbyls | optstockbyls | optstocksensbyls | optpricebysim | optstockbybaw | optstocksensbybaw

Related Examples

- “Equity Derivatives Using Closed-Form Solutions” on page 3-79
- “Pricing European Call Options Using Different Equity Models” on page 3-88
- “Pricing Asian Options” on page 3-110

More About

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Pricing European and American Spread Options

This example shows how to price and calculate sensitivities for European and American spread options using various techniques. First, the price and sensitivities for a European spread option is calculated using closed form solutions. Then, price and sensitivities for an American spread option is calculated using finite difference and Monte Carlo simulations. Finally, further analysis is conducted on spread options with a different range of inputs.

Spread options are options on the difference of two underlying asset prices. For example, a call option on the spread between two assets has the following payoff at maturity:

$$max(X_1 - X_2 - K, 0)$$

where X_1 is the price of the first underlying asset, X_2 is the price of the second underlying asset, and K is the strike price. At maturity, if the spread $X_1 - X_2$ is greater than the strike price K, the option holder exercises the option and gains the difference between the spread and the strike price. If the spread is less than 0, the option holder does not exercise the option, and the payoff is 0. Spread options are frequently traded in the energy market. Two examples are:

- *Crack spreads*: Options on the spread between refined petroleum products and crude oil. The spread represents the refinement margin made by the oil refinery by "cracking" the crude oil into a refined petroleum product.
- *Spark spreads*: Options on the spread between electricity and some type of fuel. The spread represents the margin of the power plant, which takes fuel to run its generator to produce electricity.

Overview of the Pricing Methods

There are several methods to price spread options, as discussed in [1 on page 3-107]. This example uses the closed form, finite difference, and Monte Carlo simulations to price spread options. The advantages and disadvantages of each method are discussed below:

- Closed form solutions and approximations of partial differential equations (PDE) are advantageous because they are very fast, and extend well to computing sensitivities (Greeks). However, closed form solutions are not always available, for example for American spread options.
- The finite difference method is a numerical procedure to solve PDEs by discretizing the price and time variables into a grid. A detailed analysis of this method can be found in [2 on page 3-107]. It can handle cases where closed form solutions are not available. Also, finite difference extends well to calculating sensitivities because it outputs a grid of option prices for a range of underlying prices and times. However, it is slower than the closed form solutions.
- Monte Carlo simulation uses random sampling to simulate movements of the underlying asset prices. It handles cases where closed solutions do not exist. However, it usually takes a long time to run, especially if sensitivities are calculated.

Pricing a European Spread Option

The following example demonstrates the pricing of a crack spread option.

A refiner is concerned about its upcoming maintenance schedule and needs to protect against decreasing crude oil prices and increasing heating oil prices. During the maintenance the refiner needs to continue providing customers with heating oil to meet their demands. The refiner's strategy is to use spread options to manage its hedge.

On January 2013, the refiner buys a 1:1 crack spread option by purchasing heating oil futures and selling crude oil futures. CLF14 WTI crude oil futures is at $100 per barrel and HOF14 heating oil futures contract is at $2.6190 per gallon.

```
clear;

% Price, volatility, and dividend of heating oil
Price1gallon = 2.6190;       % $/gallon
Price1 = Price1gallon*42;    % $/barrel
Vol1 = 0.10;
Div1 = 0.03;

% Price, volatility, and dividend of WTI crude oil
Price2 = 100;     % $/barrel
Vol2 = 0.15;
Div2 = 0.02;

% Correlation of underlying prices
Corr = 0.3;

% Option type
OptSpec = 'call';

% Strike
Strike = 5;

% Settlement date
Settle = '01-Jan-2013';

% Maturity
Maturity = '01-Jan-2014';

% Risk free rate
RiskFreeRate = 0.05;
```

The pricing functions take an interest-rate term structure and stock structure as inputs. Also, you need to specify which outputs are of interest.

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

% Define StockSpec for the two assets
StockSpec1 = stockspec(Vol1, Price1, 'Continuous', Div1);
StockSpec2 = stockspec(Vol2, Price2, 'Continuous', Div2);

% Specify price and sensitivity outputs
OutSpec = {'Price', 'Delta', 'Gamma'};
```

The Financial Instruments Toolbox™ contains two types of closed form approximations for calculating price and sensitivities of European spread options: the Kirk's approximation (`spreadbykirk`, `spreadsensbykirk`) and the Bjerksund and Stensland model (`spreadbybjs`, `spreadsensbybjs`) [3 on page 3-107].

The function `spreadsensbykirk` calculates prices and sensitivities for a European spread option using the Kirk's approximation.

```
% Kirk's approximation
[PriceKirk, DeltaKirk, GammaKirk] = ...
    spreadsensbykirk(RateSpec, StockSpec1, StockSpec2, Settle, ...
    Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec)
```

```
PriceKirk = 8.3636

DeltaKirk = 1×2

    0.6108   -0.5590


GammaKirk = 1×2

    0.0225    0.0249
```

The function `spreadsensbybjs` calculates the prices and sensitivities for a European spread option using the Bjerksund and Stensland model. In [3 on page 3-107], Bjerksund and Stensland explains that the Kirk's approximation tends to underprice the spread option when the strike is close to zero, and overprice when the strike is further away from zero. In comparison, the model by Bjerksund and Stensland has higher precision.

```
% Bjerksund and Stensland model
[PriceBJS, DeltaBJS, GammaBJS] = ...
    spreadsensbybjs(RateSpec, StockSpec1, StockSpec2, Settle, ...
    Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec)
```

```
PriceBJS = 8.3662

DeltaBJS = 1×2

    0.6115   -0.5597


GammaBJS = 1×2

    0.0225    0.0248
```

A comparison of the calculated prices show that the two closed form models produce similar results for price and sensitivities. In addition to delta and gamma, the functions can also calculate theta, vega, lambda, and rho.

```
displayComparison('Kirk', 'BJS', PriceKirk, PriceBJS, DeltaKirk, DeltaBJS, GammaKirk, GammaBJS)
```

```
Comparison of prices:

Kirk:    8.363641
BJS :    8.366158

Comparison of delta:

Kirk:    0.610790     -0.558959
BJS :    0.611469     -0.559670
```

```
Comparison of gamma:

Kirk:    0.022533      0.024850
BJS :    0.022495      0.024819
```

Pricing an American Spread Option

Although the closed form approximations are fast and well suited for pricing European spread options, they cannot price American spread options. Using the finite difference method and the Monte Carlo method, an American spread option can be priced. In this example, an American spread option is priced with the same attributes as the above crack spread option.

The finite difference method numerically solves a PDE by discretizing the underlying price and time variables into a grid. The Financial Instrument Toolbox™ contains the functions `spreadbyfd` and `spreadsensbyfd`, which calculate prices and sensitivities for European and American spread options using the finite difference method. For the finite difference method, the composition of the grid has a large impact on the quality of the output and the execution time. Generally, a finely discretized grid will result in outputs that are closer to the theoretical value, but it comes at the cost of longer execution times. The composition of the grid is controlled using optional parameters `PriceGridSize`, `TimeGridSize`, `AssetPriceMin` and `AssetPriceMax`.

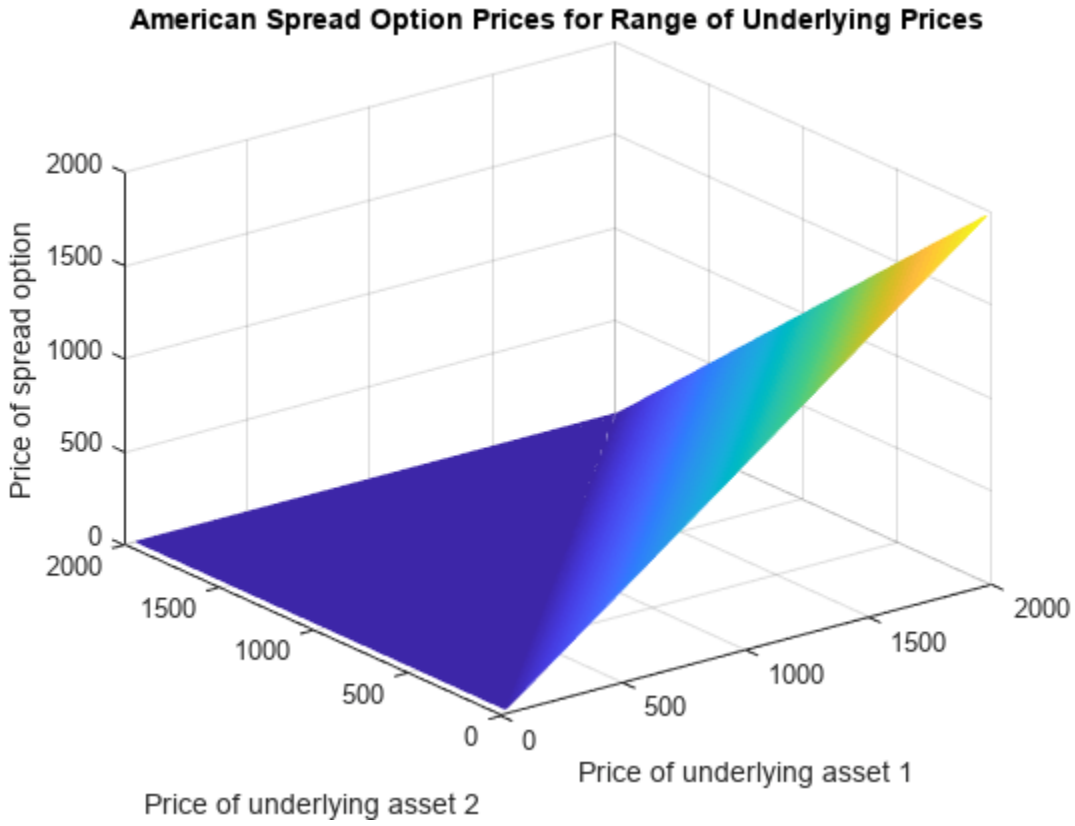
To indicate pricing an American option, add an optional input of `AmericanOpt` with a value of `1` to the argument of the function.

```
% Finite difference method for American spread option
[PriceFD, DeltaFD, GammaFD, PriceGrid, AssetPrice1, ...
    AssetPrice2] = ...
    spreadsensbyfd(RateSpec, StockSpec1, StockSpec2, Settle, ...
    Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec, ...
    'PriceGridSize', [500 500], 'TimeGridSize', 100, ...
    'AssetPriceMin', [0 0], 'AssetPriceMax', [2000 2000], ...
    'AmericanOpt', 1);

% Display price and sensitivities
PriceFD

PriceFD = 8.5463

DeltaFD

DeltaFD = 1×2

    0.6306   -0.5777


GammaFD

GammaFD = 1×2

    0.0233    0.0259
```

The function `spreadsensbyfd` also returns a grid that contains the option prices for a range of underlying prices and times. The grid of option prices at time zero, which is the option prices at the settle date, can be plotted for a range of underlying prices.

```
% Plot option prices
figure;
```

```
mesh(AssetPrice1, AssetPrice2, PriceGrid(:, :, 1)');
title('American Spread Option Prices for Range of Underlying Prices');
xlabel('Price of underlying asset 1');
ylabel('Price of underlying asset 2');
zlabel('Price of spread option');
```



An American style option can be priced by Monte Carlo methods using the least square method of Longstaff and Schwartz [4 on page 3-107]. The Financial Instruments Toolbox™ contains the functions `spreadbyls` and `spreadsensbyls`, that calculate prices and sensitivities of European and American options using simulations. The Monte Carlo simulation method in `spreadsensbyls` generates multiple paths of simulations according to a geometric Brownian motion (GBM) for the two underlying asset prices. Similar to the finite difference method where the granularity of the grid determined the quality of the output and the execution time, the quality of output and execution time of the Monte Carlo simulation depends on the number of paths (`NumTrials`) and the number of time periods per path (`NumPeriods`). Also, the results obtained by Monte Carlo simulations are not deterministic. Each run will have different results depending on the simulation outcomes.

```
% To indicate that we are pricing an American option using the Longstaff
% and Schwartz method, add an optional input of |AmericanOpt| with a value
% of |1| to the argument of the function.

% Monte Carlo method for American spread option
[PriceMC, DeltaMC, GammaMC] = ...
    spreadsensbyls(RateSpec, StockSpec1, StockSpec2, Settle, ...
    Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec, ...
    'NumTrials', 1000, 'Antithetic', true, 'AmericanOpt', 1)
```

```
PriceMC = 8.4999

DeltaMC = 1×2

    0.6325   -0.5931


GammaMC = 1×2

   -0.0873    0.0391
```

The results of the two models are compared. The prices and sensitivities calculated by the Longstaff and Schwartz method will vary at each run, depending on the outcome of the simulations. It is important to note that the quality of the results from the finite difference method and the Monte Carlo simulation depend on the optional input parameters. For example, increasing the number of paths (`NumTrials`) for the `spreadsensbyls` function will result in more precise results at the cost of longer execution times.

```
displayComparison('Finite Difference', 'Monte Carlo', PriceFD, PriceMC, DeltaFD, DeltaMC, GammaFD
```

```
Comparison of prices:

Finite Difference:    8.546285
Monte Carlo      :    8.499894

Comparison of delta:

Finite Difference:    0.630606     -0.577686
Monte Carlo      :    0.632549     -0.593106

Comparison of gamma:

Finite Difference:    0.023273      0.025852
Monte Carlo      :   -0.087340      0.039120
```

Comparing Results for a Range of Strike Prices

As discussed earlier, the Kirk's approximation tends to overprice spread options when the strike is further away from zero. To confirm this, a spread option is priced with the same attributes as before, but for a range of strike prices.

```
% Specify outputs
OutSpec = {'Price', 'Delta'};

% Range of strike prices
Strike = [-25; -15; -5; 0; 5; 15; 25];
```

The results from the Kirk's approximation and the Bjerksund and Stensland model are compared against the numerical approximation from the finite difference method. Since `spreadsensbyfd` can only price one option at a time, it is called in a loop for each strike value. The Monte Carlo simulation (`spreadsensbyls`) with a large number of trial paths can also be used as a benchmark, but the finite difference is used for this example.

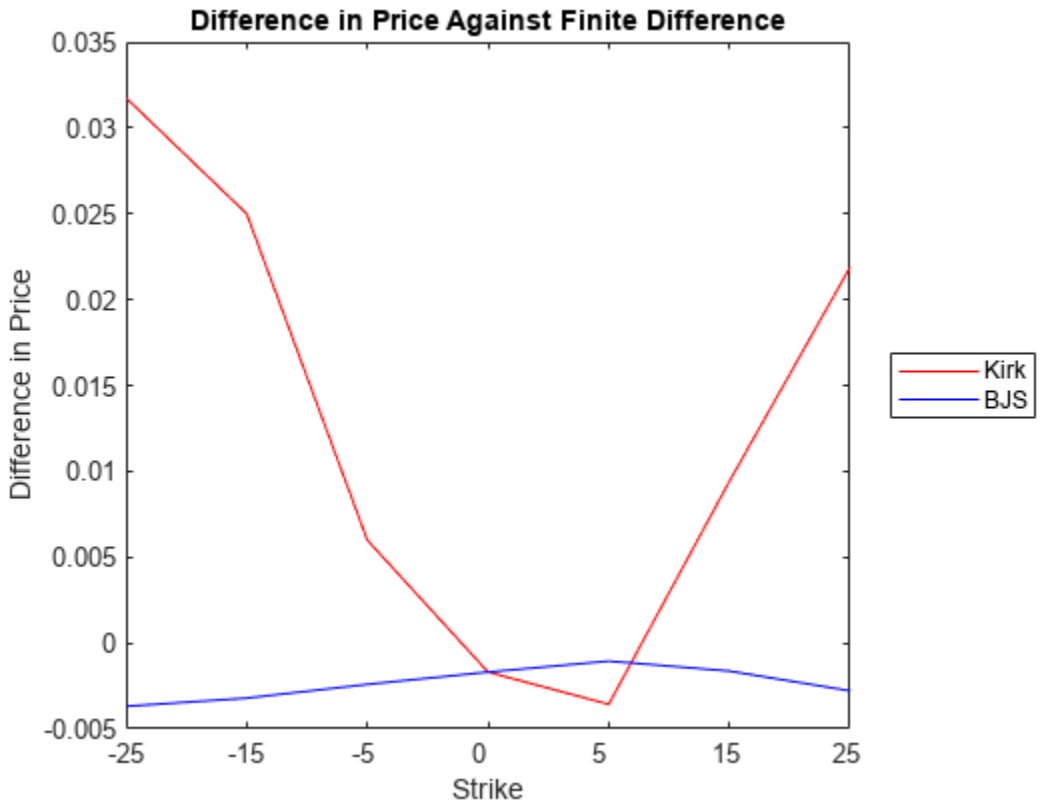
```
% Kirk's approximation
[PriceKirk, DeltaKirk] = ...
    spreadsensbykirk(RateSpec, StockSpec1, StockSpec2, Settle, ...
    Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec);
```

```
% Bjerksund and Stensland model
[PriceBJS, DeltaBJS] = ...
    spreadsensbybjs(RateSpec, StockSpec1, StockSpec2, Settle, ...
    Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec);

% Finite difference
PriceFD = zeros(numel(Strike), 1);
DeltaFD = zeros(numel(Strike), 2);
for i = 1:numel(Strike)
    [PriceFD(i), DeltaFD(i,:)] = ...
    spreadsensbyfd(RateSpec, StockSpec1, StockSpec2, Settle, ...
    Maturity, OptSpec, Strike(i), Corr, 'OutSpec', OutSpec, ...
    'PriceGridSize', [500 500], 'TimeGridSize', 100, ...
    'AssetPriceMin', [0 0], 'AssetPriceMax', [2000 2000]);
end

displayComparisonPrices(PriceKirk, PriceBJS, PriceFD, Strike)
```

```
Prices for range of strikes:

Kirk         BJS          FD
32.707787    32.672353    32.676040
23.605307    23.577099    23.580307
15.236908    15.228510    15.230919
11.560332    11.560332    11.562023
8.363641    8.366158    8.367212
3.689909    3.678862    3.680493
1.243753    1.219079    1.221866
```

The difference in prices between the closed form and finite difference method is plotted below. It is clear that as the strike moves further away from 0, the difference between the Kirk's approximation and finite difference (red line) increases, while the difference between the Bjerksund and Stensland model and finite difference (blue line) stays at the same level. As stated in [3 on page 3-107], the Kirk's approximation is overpricing the spread option when the strike is far away from 0.

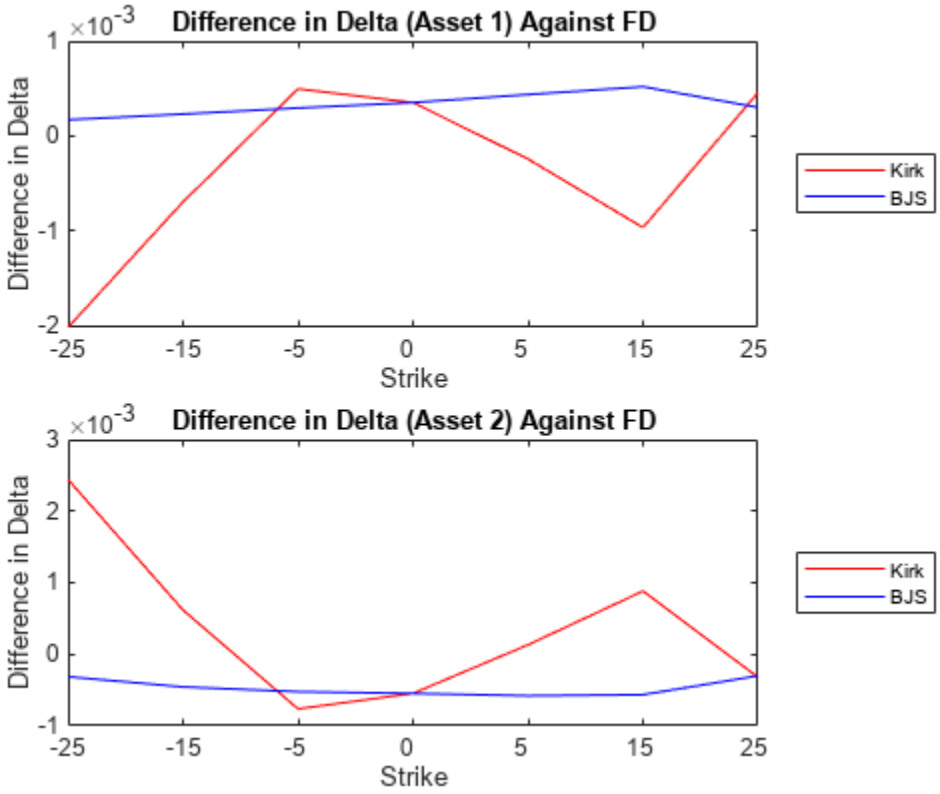
```
% Plot of difference in price against the benchmark
figure;
plot(PriceKirk-PriceFD, 'Color', 'red');
hold on;
plot(PriceBJS-PriceFD, 'Color', 'blue');
hold off;
title('Difference in Price Against Finite Difference');
legend('Kirk', 'BJS', 'Location', 'EastOutside');
xlabel('Strike');
ax = gca;
ax.XTickLabel = Strike;
ylabel('Difference in Price');
```



Next, the difference in delta between the closed form models and finite difference is plotted. The top plot shows the difference in delta for the first asset, and the bottom plot shows the difference in delta for the second asset. As seen from the small increments in the y-axis of the order 10e-3, it can be seen that all three models (Kirk, BJS, finite difference) produce similar values for delta.

```
% Plot of difference in delta of first asset against the benchmark
figure;
subplot(2, 1, 1);
plot(DeltaKirk(:,1)-DeltaFD(:,1), 'Color', 'red');
hold on;
plot(DeltaBJS(:,1)-DeltaFD(:,1), 'Color', 'blue');
hold off;
title('Difference in Delta (Asset 1) Against FD');
legend('Kirk', 'BJS', 'Location', 'EastOutside');
xlabel('Strike');
ax = gca;
ax.XTickLabel = Strike;
ylabel('Difference in Delta');

% Plot of difference in delta of second asset against the benchmark
subplot(2, 1, 2);
plot(DeltaKirk(:,2)-DeltaFD(:,2), 'Color', 'red');
hold on;
plot(DeltaBJS(:,2)-DeltaFD(:,2), 'Color', 'blue');
hold off;
title('Difference in Delta (Asset 2) Against FD');
legend('Kirk', 'BJS', 'Location', 'EastOutside');
```

```
xlabel('Strike');
ax = gca;
ax.XTickLabel = Strike;
ylabel('Difference in Delta');
```



Analyzing Prices and Vega at Different Levels of Volatility

To further show the type of analysis that can be conducted using these models, the above spread option is priced at different levels of volatility for the first asset. The price and vega are compared at three levels of volatility for the first asset: `0.1`, `0.3`, and `0.5`. The Bjerksund and Stensland model is used for this analysis.

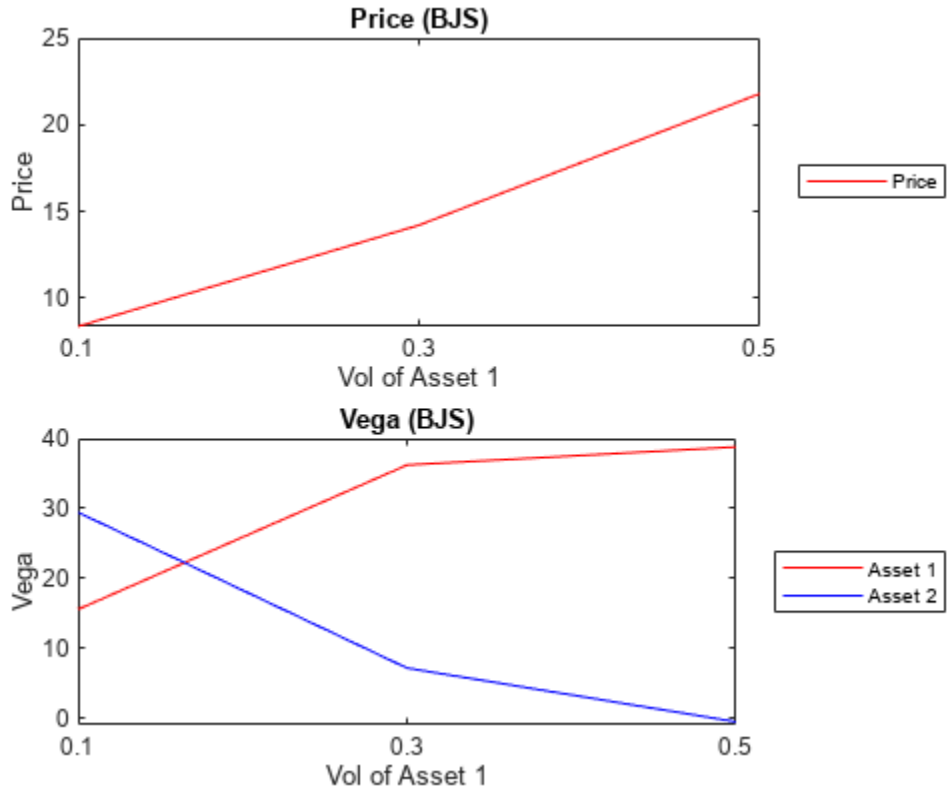
```
% Strike
Strike = 5;

% Specify output
OutSpec = {'Price', 'Vega'};

% Different levels of volatility for asset 1
Vol1 = [0.1, 0.3, 0.5];

StockSpec1 = stockspec(Vol1, Price1, 'Continuous', Div1);

% Bjerksund and Stensland model
[PriceBJS, VegaBJS] = ...
    spreadsensbybjs(RateSpec, StockSpec1, StockSpec2, Settle, ...
    Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec);
```

```
displaySummary(Vol1, PriceBJS, VegaBJS)
```

```
Prices for different vol levels in asset 1:

8.366158
14.209112
21.795746

Asset 1 vega for different vol levels in asset 1:

15.534849
36.212192
38.794348

Asset 2 vega for different vol levels in asset 1:

29.437036
7.133657
-0.557852
```

The change in the price and vega with respect to the volatility of the first asset is plotted below. You can observe that as the volatility of the first asset increases, the price of the spread option also increases. Also, the changes in vega indicate that the price of the spread option becomes more sensitive to the volatility of the first asset and less sensitive to the volatility of the second asset.

```
figure;

% Plot price for BJS model
subplot(2, 1, 1);
plot(PriceBJS, 'Color', 'red');
title('Price (BJS)');
legend('Price', 'Location', 'EastOutside');
xlabel('Vol of Asset 1');
ax = gca;
ax.XTick = 1:3;
ax.XTickLabel = Vol1;
ylabel('Price');

% Plot of vega for BJS model
subplot(2, 1, 2);
plot(VegaBJS(:,1), 'Color', 'red');
hold on;
plot(VegaBJS(:,2), 'Color', 'blue');
hold off;
title('Vega (BJS)');
legend('Asset 1', 'Asset 2', 'Location', 'EastOutside');
xlabel('Vol of Asset 1');
ax = gca;
ax.XTick = 1:3;
ax.XTickLabel = Vol1;
ax.YLim = [-1 40];
ylabel('Vega');
```



Summary

In this example, European and American spread options are priced and analyzed using various techniques. The Financial Instruments Toolbox™ provides functions for two types of closed form solutions (Kirk, BJS), the finite difference method, and the Monte Carlo simulation method. The closed form solutions are well suited for pricing and sensitivity calculation of European spread options because they are fast. However, they cannot price American spread options. The finite difference method and Monte Carlo method can price both European and American options. However, they are not as fast in pricing European spread options as compared to closed form solutions.

References

[1] Carmona, Rene, Durrleman, Valdo. "Pricing and Hedging Spread Options." *SIAM Review*. Vol. 45, No. 4, 2003, pp. 627-685.

[2] Wilmott, Paul, Dewynne, Jeff, Howison, Sam. *Option Pricing*. Oxford Financial Press, 1993.

[3] Bjerksund, Petter, Stensland, Gunnar. "Closed form spread option valuation." Department of Finance, NHH, 2006.

[4] Longstaff, Francis A, Schwartz, Eduardo S. "Valuing American Options by Simulation: A Simple Least-Squares Approach." Anderson Graduate School of Management, UC Los Angeles, 2001.

Utility Functions

```
function displayComparison(model1, model2, price1, price2, delta1, delta2, gamma1, gamma2)
% Pad the model name with additional spaces
additionalSpaces = numel(model1) - numel(model2);
if additionalSpaces > 0
    model2 = [model2 repmat(' ', 1, additionalSpaces)];
else
    model1 = [model1 repmat(' ', 1, abs(additionalSpaces))];
end

% Comparison of calculated prices
fprintf('Comparison of prices:\n');
fprintf('\n');
fprintf('%s:   % f\n', model1, price1);
fprintf('%s:   % f\n', model2, price2);
fprintf('\n');

% Comparison of Delta
fprintf('Comparison of delta:\n');
fprintf('\n');
fprintf('%s:   % f     % f\n', model1, delta1(1), delta1(2));
fprintf('%s:   % f     % f\n', model2, delta2(1), delta2(2));
fprintf('\n');

% Comparison of Gamma
fprintf('Comparison of gamma:\n');
fprintf('\n');
fprintf('%s:   % f     % f\n', model1, gamma1(1), gamma1(2));
fprintf('%s:   % f     % f\n', model2, gamma2(1), gamma2(2));
fprintf('\n');
end

function displayComparisonPrices(PriceKirk, PriceBJS, PriceFD, Strike)
% Comparison of calculated prices
fprintf('Prices for range of strikes:\n');
fprintf('\n')
fprintf('Kirk     \tBJS      \tFD    \n');
for i = 1:numel(Strike)
    fprintf('%f\t%f\t%f\n', PriceKirk(i), PriceBJS(i), PriceFD(i));
end
end

function displaySummary(Vol1, PriceBJS, VegaBJS)
% Display price
fprintf('Prices for different vol levels in asset 1:\n');
fprintf('\n');
for i = 1:numel(Vol1)
    fprintf('%f\n', PriceBJS(i));
end
fprintf('\n');

% Display vega for first asset
fprintf('Asset 1 vega for different vol levels in asset 1:\n');
fprintf('\n');
for i = 1:numel(Vol1)
    fprintf('%f\n', VegaBJS(i,1));
end
```

```
fprintf('\n');

% Display vega for second asset
fprintf('Asset 2 vega for different vol levels in asset 1:\n');
fprintf('\n');
for i = 1:numel(Vol1)
    fprintf('%f\n', VegaBJS(i,2));
end
end
```

See Also

More About

- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Pricing Asian Options

This example shows how to price a European Asian option using six methods in the Financial Instruments Toolbox™. This example demonstrates four closed form approximations (Kemna-Vorst, Levy, Turnbull-Wakeman, and Haug-Haug-Margrabe), a lattice model (Cox-Ross-Rubinstein), and Monte Carlo simulation. All these methods involve some tradeoffs between numerical accuracy and computational efficiency. This example also demonstrates how variations in spot prices, volatility, and strike prices affect option prices on European Vanilla and Asian options.

Overview of Asian Options

Asian options are securities with payoffs that depend on the average value of an underlying asset over a specific period of time. Underlying assets can be stocks, commodities, or financial indices.

Two types of Asian options are found in the market: average price options and average strike options. Average price options have a fixed strike value and the average used is the asset price. Average strike options have a strike equal to the average value of the underlying asset.

The payoff at maturity of an average price European Asian option is:

$max(0, Savg - K)$ for a call

$max(0, K - Savg)$ for a put

The payoff at maturity of an average strike European Asian option is:

$max(0, St - Savg)$ for a call

$max(0, Savg - St)$ for a put

where $Savg$ is the average price of underlying asset, St is the price at maturity of underlying asset, and K is the strike price.

The average can be arithmetic or geometric.

Pricing Asian Options Using Closed Form Approximations

The Financial Instruments Toolbox™ supports four closed form approximations for European Average Price options. The Kemna-Vorst method is based on the geometric mean of the price of the underlying during the life of the option [1]. The Levy and Turnbull-Wakeman models provide a closed form pricing solution to continuous arithmetic averaging options [2,3 on page 3-120]. The Haug-Haug-Margrabe approximation is used for pricing discrete arithmetic averaging options [4].

All the pricing functions `asianbykv`, `asianbylevy`, `asianbytw`, and `asianbyhhm` take an interest-rate term structure and stock structure as inputs.

Consider the following example:

```
% Create RateSpec from the interest rate term structure
StartDates = '12-March-2014';
EndDates = '12-March-2020';
Rates = 0.035;
Compounding = -1;
Basis = 1;
```

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

% Define StockSpec with the underlying asset information
Sigma = 0.20;
AssetPrice = 100;

StockSpec = stockspec(Sigma, AssetPrice);

% Define the Asian option
Settle = '12-March-2014';
ExerciseDates = '12-March-2015';
Strike = 90;
OptSpec = 'call';

% Kemna-Vorst closed form model
PriceKV = asianbykv(RateSpec, StockSpec, OptSpec, Strike, Settle,...
    ExerciseDates);

% Levy model approximation
PriceLevy = asianbylevy(RateSpec, StockSpec, OptSpec, Strike, Settle,...
    ExerciseDates);

% Turnbull-Wakeman approximation
PriceTW = asianbytw(RateSpec, StockSpec, OptSpec, Strike, Settle,...
    ExerciseDates);

% Haug-Haug-Margrabe approximation
PriceHHM = asianbyhhm(RateSpec, StockSpec, OptSpec, Strike, Settle,...
    ExerciseDates);

% Comparison of calculated prices for the geometric and arithmetic options
% using different closed form algorithms.
displayPricesClosedForm(PriceKV, PriceLevy, PriceTW, PriceHHM)

Comparison of Asian Arithmetic and Geometric Prices:

Kemna-Vorst:        11.862580
Levy:               12.164734
Turnbull-Wakeman:   12.164734
Haug-Haug-Margrabe: 12.108746
```

Computing Asian Options Prices Using the Cox-Ross-Rubinstein Model

In addition to closed form approximations, the Financial Instruments Toolbox™ supports pricing European Average Price options using CRR trees via the function `asianbycrr`.

The lattice pricing function `asianbycrr` takes an interest-rate tree (`CRRTree`) and stock structure as inputs. You can price the previous options by building a `CRRTree` using the interest-rate term structure and stock specification from the example above.

```
% Create the time specification of the tree
NPeriods = 20;
TreeValuationDate = '12-March-2014';
TreeMaturity = '12-March-2024';
TimeSpec = crrtimespec(TreeValuationDate, TreeMaturity, NPeriods);
```

```
% Build the tree
CRRTree =  crrtree(StockSpec, RateSpec, TimeSpec);

% Price the European Asian option using the CRR lattice model.
% The function 'asianbycrr' computes prices of arithmetic and geometric
% Asian options.
AvgType = {'arithmetic';'geometric'};
AmericanOpt = 0;
PriceCRR20 = asianbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates,...
                        AmericanOpt, AvgType);

% Increase the numbers of periods in the tree and compare results
NPeriods = 40;
TimeSpec = crrtimespec(TreeValuationDate, TreeMaturity, NPeriods);
CRRTree =  crrtree(StockSpec, RateSpec, TimeSpec);

PriceCRR40 = asianbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates,...
                        AmericanOpt, AvgType);

% Display prices
displayPricesCRR(PriceCRR20, PriceCRR40)

Asian Prices using the CRR lattice model:

PriceArithmetic(CRR20): 11.934380
PriceArithmetic(CRR40): 12.047243
PriceGeometric (CRR20): 11.620899
PriceGeometric (CRR40): 11.732037
```

The results above compare the findings from calculating both geometric and arithmetic Asian options, using CRR trees with 20 and 40 levels. As the number of levels increases, the results approach the closed form solutions.

Calculating Prices of Asian Options Using Monte Carlo Simulation

Another method to price European Average Price options with the Financial Instruments Toolbox™ is via Monte Carlo simulations.

The pricing function `asianbyls` takes an interest-rate term structure and stock structure as inputs. The output and execution time of the Monte Carlo simulation depends on the number of paths (`NumTrials`) and the number of time periods per path (`NumPeriods`).

You can price the same options of previous examples using Monte Carlo.

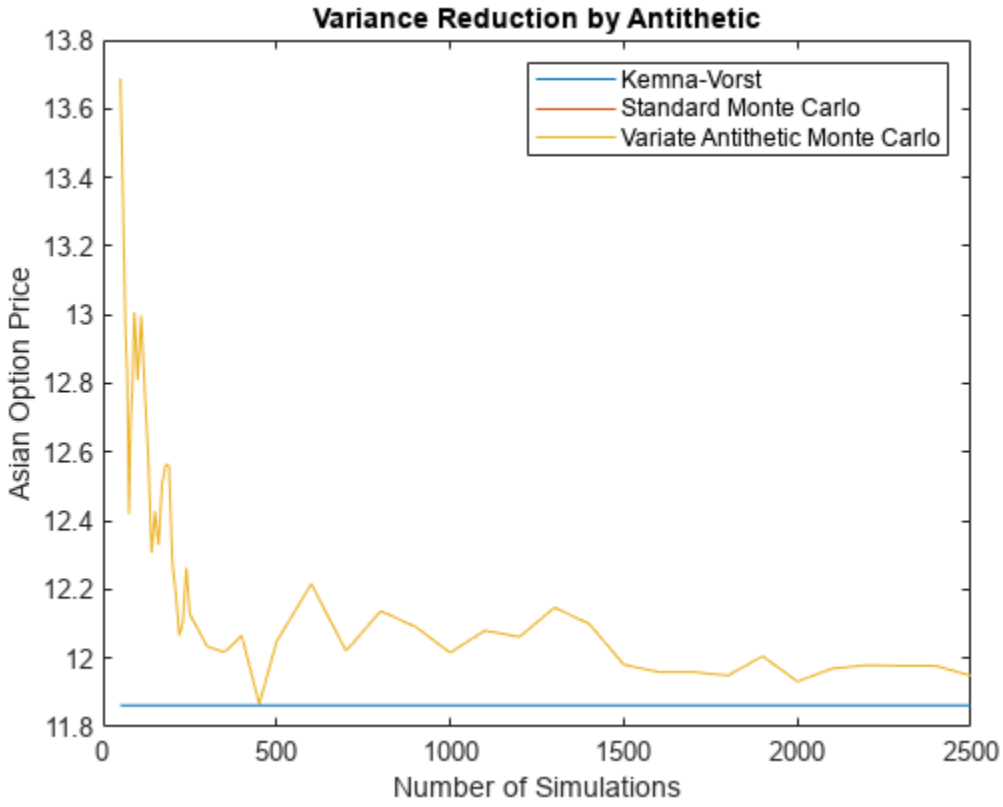
```
% Simulation Parameters
NumTrials = 500;
NumPeriods = 200;

% Price the arithmetic option
PriceAMC = asianbyls(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                     ExerciseDates,'NumTrials', NumTrials, ...
                     'NumPeriods', NumPeriods);

% Price the geometric option
PriceGMC = asianbyls(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                     ExerciseDates,'NumTrials', NumTrials, ...
                     'NumPeriods', NumPeriods, 'AvgType', AvgType(2));
```

```
% Use the antithetic variates method to value the options
Antithetic = true;
PriceAMCAntithetic = asianbyls(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                    ExerciseDates,'NumTrials', NumTrials, 'NumPeriods',...
                    NumPeriods, 'Antithetic', Antithetic);

PriceGMCAntithetic = asianbyls(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                    ExerciseDates,'NumTrials', NumTrials, 'NumPeriods',...
                    NumPeriods, 'Antithetic', Antithetic,'AvgType', AvgType(2));

% Display prices
displayPricesMonteCarlo(PriceAMC, PriceAMCAntithetic, PriceGMC, PriceGMCAntithetic)

Asian Prices using Monte Carlo Method:

Arithmetic Asian
Standard Monte Carlo:           12.304046
Variate Antithetic Monte Carlo: 12.304046

Geometric Asian
Standard Monte Carlo:           12.048434
Variate Antithetic Monte Carlo: 12.048434
```

The use of variate antithetic accelerates the conversion process by reducing the variance.

You can create a plot to display the difference between the geometric Asian price using the Kemna-Vorst model, standard Monte Carlo, and antithetic Monte Carlo.

```
nTrials = [50:5:100 110:10:250 300:50:500 600:100:2500]';
PriceKVVector = PriceKV * ones(size(nTrials));
PriceGMCVector = nan(size(nTrials));
PriceGMCAntitheticVector = nan(size(nTrials));
TimeGMCAntitheticVector = nan(length(nTrials),1);
TimeGMCVector = nan(length(nTrials),1);
idx = 1;
for iNumTrials = nTrials'
    PriceGMCVector(idx) = asianbyls(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                        ExerciseDates,'NumTrials', iNumTrials, 'NumPeriods',...
                        NumPeriods,'AvgType', AvgType(2));

    PriceGMCAntitheticVector(idx) = asianbyls(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                        ExerciseDates,'NumTrials', iNumTrials, 'NumPeriods',...
                        NumPeriods, 'Antithetic', Antithetic,'AvgType', AvgType(2));
    idx = idx+1;
end

figure('menubar', 'none', 'numbertitle', 'off')
plot(nTrials, [PriceKVVector PriceGMCVector PriceGMCAntitheticVector]);
title 'Variance Reduction by Antithetic'
xlabel 'Number of Simulations'
ylabel 'Asian Option Price'
legend('Kemna-Vorst', 'Standard Monte Carlo', 'Variate Antithetic Monte Carlo ', 'location', 'no
```



The graph above shows how oscillation in simulated price is reduced by using variate antithetic.

Compare Pricing Model Results

Prices calculated by the Monte Carlo method varies depending on the outcome of the simulations. Increase `NumTrials` and analyze the results.

```
NumTrials = 2000;

PriceAMCAntithetic2000 = asianbyls(RateSpec, StockSpec, OptSpec, Strike, Settle, ExerciseDates,.
         'NumTrials', NumTrials, 'NumPeriods', NumPeriods, 'Antithetic', Antithetic);

PriceGMCAntithetic2000 = asianbyls(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                    ExerciseDates,'NumTrials', NumTrials, 'NumPeriods',...
                    NumPeriods, 'Antithetic', Antithetic,'AvgType', AvgType(2));

% Comparison of calculated Asian call prices
displayComparisonAsianCallPrices(PriceLevy, PriceTW, PriceHHM, PriceCRR40, PriceAMCAntithetic, P

Comparison of Asian call prices:

Arithmetic Asian
Levy:                     12.164734
Turnbull-Wakeman:         12.164734
Haug-Haug-Margrabe:       12.108746
Cox-Ross-Rubinstein:      12.047243
Monte Carlo(500 trials):  12.304046
Monte Carlo(2000 trials): 12.196848
```

```
Geometric Asian
Kemna-Vorst:              11.862580
Cox-Ross-Rubinstein:      11.732037
Monte Carlo(500 trials):  12.048434
Monte Carlo(2000 trials): 11.932017
```

The table above contrasts the results from closed approximation models against price simulations implemented via CRR trees and Monte Carlo.

Asian and Vanilla Call Options

Asian options are popular instruments since they tend to be less expensive than comparable Vanilla calls and puts. This is because the volatility in the average value of an underlier tends to be lower than the volatility of the value of the underlier itself.

The Financial Instruments Toolbox™ supports several algorithms for pricing vanilla options. Let us compare the price of Asian options against their Vanilla counterpart.

First, compute the price of a European Vanilla Option using the Black Scholes model.

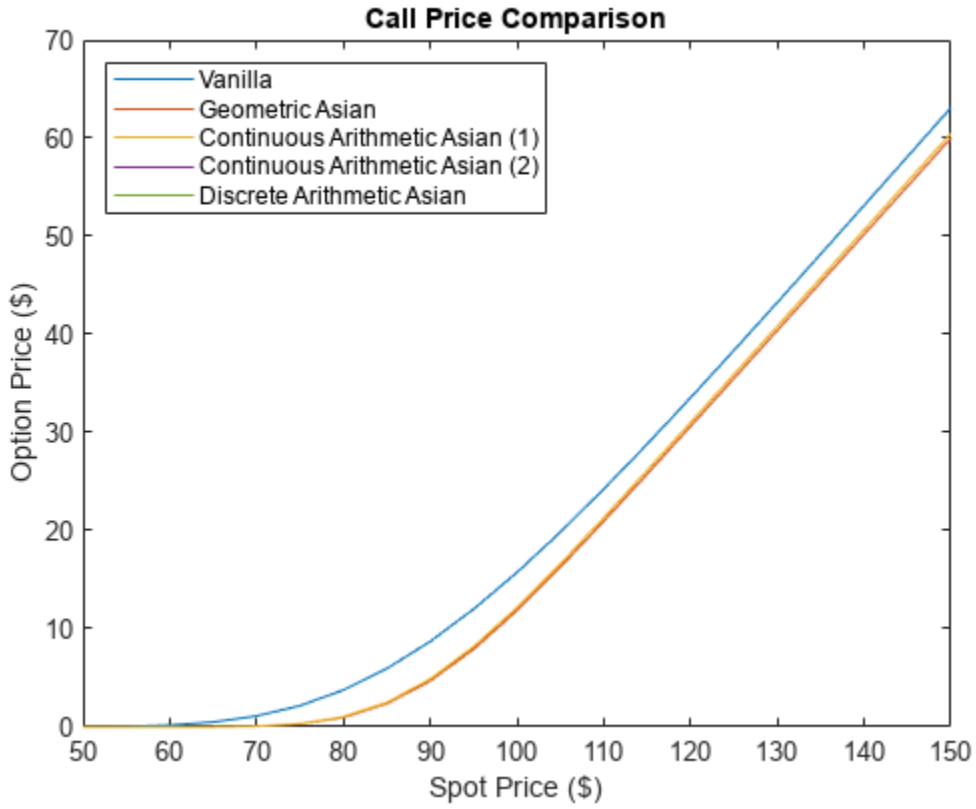
```
PriceBLS = optstockbybls(RateSpec, StockSpec, Settle, ExerciseDates,...
                         OptSpec, Strike);

% Comparison of calculated call prices.
displayComparisonVanillaAsian('Prices', PriceBLS, PriceKV, PriceLevy, PriceTW, PriceHHM)

Comparison of Vanilla and Asian Prices:

Vanilla BLS:              15.743809
Asian Kemna-Vorst:        11.862580
Asian Levy:               12.164734
Asian Turnbull-Wakeman:   12.164734
Asian Haug-Haug-Margrabe: 12.108746
```

Both geometric and arithmetic Asians price lower than their Vanilla counterpart.

You can analyze options prices at different levels of the underlying asset. Using the Financial Instruments Toolbox™, it is possible to observe the effect of different parameters on the price of the options. Consider for example, the effect of variations in the price of the underlying asset.

```
StockPrices = (50:5:150)';
PriceBLS = nan(size(StockPrices));
PriceKV = nan(size(StockPrices));
PriceLevy = nan(size(StockPrices));
PriceTW = nan(size(StockPrices));
PriceHHM = nan(size(StockPrices));
idx = 1;
for So = StockPrices'
    SP = stockspec(Sigma, So);
    PriceBLS(idx) = optstockbybls(RateSpec, SP, Settle, ExerciseDates,...
                                  OptSpec, Strike);

    PriceKV(idx) = asianbykv(RateSpec, SP, OptSpec, Strike, Settle,...
                             ExerciseDates);

    PriceLevy(idx) = asianbylevy(RateSpec, SP, OptSpec, Strike, Settle,...
                                 ExerciseDates);
```

```
    PriceKV(idx) = asianbykv(RateSpec, SP, OptSpec, Strike, Settle,...
                             ExerciseDates);

    PriceKV(idx) = asianbykv(RateSpec, SP, OptSpec, Strike, Settle,...
                             ExerciseDates);

    idx = idx+1;
end

figure('menubar', 'none', 'numbertitle', 'off')
plot(StockPrices, [PriceBLS PriceKV PriceLevy PriceTW PriceHHM]);
xlabel 'Spot Price ($)'
ylabel 'Option Price ($)'
title 'Call Price Comparison'
legend('Vanilla', 'Geometric Asian', 'Continuous Arithmetic Asian (1)', 'Continuous Arithmetic As
```



It can be observed that the price of the Asian option is cheaper than the price of the Vanilla option.

Also, it is possible to observe the effect of changes in the volatility of the underlying asset. The table below shows what happens to Asian and Vanilla option prices when the constant volatility changes.

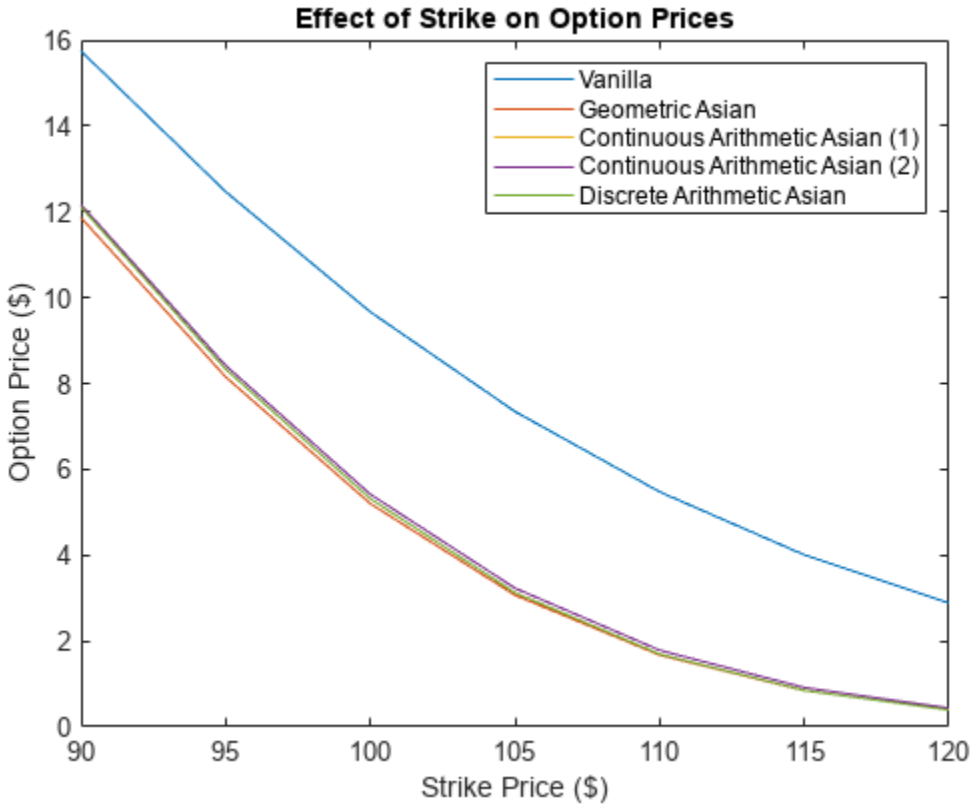
```
Call Option (ITM)

Strike = 90 AssetPrice = 100

------------------------------------------------------------------------------
--------
```

```
Volatility Haug-Haug-Margrabe Turnbull-Wakeman Levy Kemna-Vorst BLS

10% 11.3946 11.3987 11.3987 11.3121 13.4343

20% 12.1087 12.1647 12.1647 11.8626 15.7438

30% 13.5374 13.6512 13.6512 13.0338 18.8770

40% 15.2823 15.4464 15.4464 14.4086 22.2507
```

A comparison of the calculated prices show that Asian options are less sensitive to volatility changes, since averaging reduces the volatility of the value of the underlying asset. Also, Asian options that use arithmetic average are more expensive than those that use geometric average.

Now, examine the effect of strike on option prices.

```
Strikes = (90:5:120)';
NStrike = length(Strikes);
PriceBLS = nan(size(Strikes));
PriceKV = nan(size(Strikes));
PriceLevy = nan(size(Strikes));
PriceTW = nan(size(Strikes));
PriceHHM = nan(size(Strikes));
idx = 1;
for ST = Strikes'
    SP = stockspec(Sigma, AssetPrice);
    PriceBLS(idx) = optstockbybls(RateSpec, SP, Settle, ExerciseDates,...
                                  OptSpec, ST);

    PriceKV(idx) = asianbykv(RateSpec, SP, OptSpec, ST, Settle,...
                             ExerciseDates);

    PriceLevy(idx) = asianbylevy(RateSpec, SP, OptSpec, ST, Settle,...
                                 ExerciseDates);

    PriceTW(idx) = asianbytw(RateSpec, SP, OptSpec, ST, Settle,...
                             ExerciseDates);

    PriceHHM(idx) = asianbyhhm(RateSpec, SP, OptSpec, ST, Settle,...
                             ExerciseDates);

    idx = idx+1;
end

figure('menubar', 'none', 'numbertitle', 'off')
plot(Strikes, [PriceBLS PriceKV PriceLevy PriceTW PriceHHM]);
xlabel 'Strike Price ($)'
ylabel 'Option Price ($)'
title 'Effect of Strike on Option Prices'
legend('Vanilla', 'Geometric Asian', 'Continuous Arithmetic Asian (1)', 'Continuous Arithmetic As
```



The figure above displays the option price with respect to strike price. Since call option value decreases as strike price increases, the Asian call curve is under the Vanilla call curve. It can be observed that the Asian call option is less expensive than the Vanilla call.

Hedging

Hedging is an insurance to minimize exposure to market movements on the value of a position or portfolio. As the underlying changes, the proportions of the instruments forming the portfolio may need to be adjusted to keep the sensitivities within the desired range. `Delta` measures the option price sensitivity to changes in the price of the underlying.

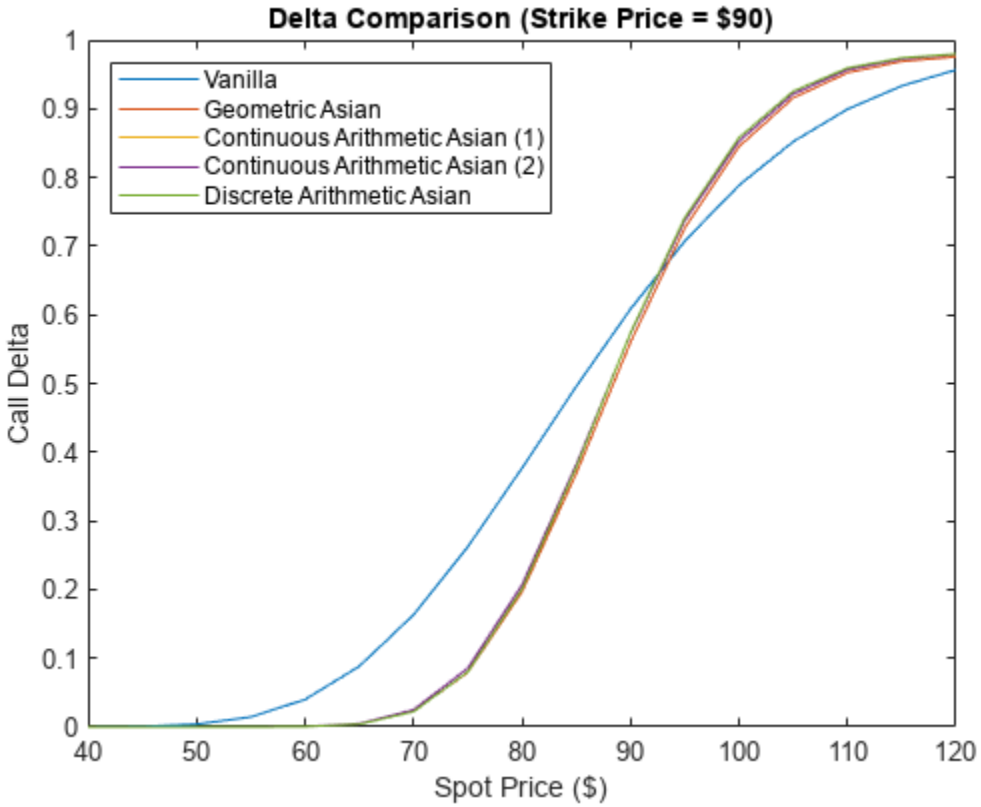
Assume that you have a portfolio of two options with the same strike and maturity. You can use the Financial Instruments Toolbox™ to compute `Delta` for the Vanilla and Average Price options.

```
OutSpec = 'Delta';

% Vanilla option using Black Scholes
DeltaBLS = optstocksensbybls(RateSpec, StockSpec, Settle, ExerciseDates,...
                             OptSpec, Strike, 'OutSpec', OutSpec);

% Asian option using Kemna-Vorst method
DeltaKV = asiansensbykv(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                        ExerciseDates,  'OutSpec', OutSpec);

% Asian option using Levy model
DeltaLevy = asiansensbylevy(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                            ExerciseDates,  'OutSpec', OutSpec);
```

```
% Asian option using Turnbull-Wakeman model
DeltaTW = asiansensbytw(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                        ExerciseDates,  'OutSpec', OutSpec);

% Asian option using Haug-Haug-Margrabe model
DeltaHHM = asiansensbyhhm(RateSpec, StockSpec, OptSpec, Strike, Settle,...
                        ExerciseDates,  'OutSpec', OutSpec);

% Delta Comparison
displayComparisonVanillaAsian('Delta', DeltaBLS, DeltaKV, DeltaLevy, DeltaTW, DeltaHHM)

Comparison of Vanilla and Asian Delta:

Vanilla BLS:              0.788666
Asian Kemna-Vorst:        0.844986
Asian Levy:               0.852806
Asian Turnbull-Wakeman:   0.852806
Asian Haug-Haug-Margrabe: 0.857864
```

The following graph demonstrates the behavior of `Delta` for the Vanilla and Asian options as a function of the underlying price.

```
StockPrices = (40:5:120)';
NStockPrices = length(StockPrices);
DeltaBLS = nan(size(StockPrices));
DeltaKV = nan(size(StockPrices));
DeltaLevy = nan(size(StockPrices));
DeltaTW = nan(size(StockPrices));
DeltaHHM = nan(size(StockPrices));

idx = 1;
for SPrices = StockPrices'
    SP = stockspec(Sigma, SPrices);
    DeltaBLS(idx) = optstocksensbybls(RateSpec, SP, Settle, ...
                    ExerciseDates, OptSpec, Strike, 'OutSpec', OutSpec);

    DeltaKV(idx) = asiansensbykv(RateSpec, SP, OptSpec, Strike, ...
                   Settle, ExerciseDates,'OutSpec', OutSpec);

    DeltaLevy(idx) = asiansensbylevy(RateSpec, SP, OptSpec, Strike,...
                     Settle, ExerciseDates, 'OutSpec', OutSpec);

    DeltaTW(idx) = asiansensbytw(RateSpec, SP, OptSpec, Strike, ...
                   Settle, ExerciseDates,'OutSpec', OutSpec);

    DeltaHHM(idx) = asiansensbyhhm(RateSpec, SP, OptSpec, Strike, ...
                   Settle, ExerciseDates,'OutSpec', OutSpec);

    idx = idx+1;
end

figure('menubar', 'none', 'numbertitle', 'off')
plot(StockPrices, [DeltaBLS DeltaKV DeltaLevy DeltaTW DeltaHHM]);
xlabel 'Spot Price ($)'
ylabel 'Call Delta'
title 'Delta Comparison (Strike Price = $90)'
legend('Vanilla', 'Geometric Asian', 'Continuous Arithmetic Asian (1)', 'Continuous Arithmetic As
```



A Vanilla, or Asian, in the money (ITM) call option is more sensitive to price movements than an out of the money (OTM) option. If the asset price is deep in the money, then it is more likely to be exercised. The opposite occurs for an out of the money option. Asian delta is lower for out of the money options and is higher for in the money options than its Vanilla European counterpart. The geometric Asian delta is lower than the arithmetic Asian delta.

References

[1] Kemna, A. and Vorst, A. "A Pricing Method for Options Based on Average Asset Values." *Journal of Banking and Finance.* Vol. 14, 1990, pp. 113-129.

[2] Levy, E. "Pricing European Average Rate Currency Options." *Journal of International Money and Finance*. Vol. 11,1992, pp. 474-491.

[3] Turnbull, S. M., and Wakeman, L.M. "A Quick Algorithm for Pricing European Average Options." *Journal of Financial and Quantitative Analysis.* Vol. 26(3), 1991, pp. 377-389.

[4] Haug, E. G. *The Complete Guide to Option Pricing Formulas*. McGraw-Hill, New York, 2007.

Utility Functions

```
function displayPricesClosedForm(PriceKV, PriceLevy, PriceTW, PriceHHM)
fprintf('Comparison of Asian Arithmetic and Geometric Prices:\n');
fprintf('\n');
fprintf('Kemna-Vorst:        %f\n', PriceKV);
fprintf('Levy:               %f\n', PriceLevy);
fprintf('Turnbull-Wakeman:   %f\n', PriceTW);
```

```
fprintf('Haug-Haug-Margrabe: %f\n', PriceHHM);
end

function displayPricesCRR(PriceCRR20, PriceCRR40)
fprintf('Asian Prices using the CRR lattice model:\n');
fprintf('\n');
fprintf('PriceArithmetic(CRR20): %f\n', PriceCRR20(1));
fprintf('PriceArithmetic(CRR40): %f\n', PriceCRR40(1));
fprintf('PriceGeometric (CRR20): %f\n', PriceCRR20(2));
fprintf('PriceGeometric (CRR40): %f\n', PriceCRR40(2));
end

function displayPricesMonteCarlo(PriceAMC, PriceAMCAntithetic, PriceGMC, PriceGMCAntithetic)
fprintf('Asian Prices using Monte Carlo Method:\n');
fprintf('\n');
fprintf('Arithmetic Asian\n');
fprintf('Standard Monte Carlo:           %f\n', PriceAMC);
fprintf('Variate Antithetic Monte Carlo: %f\n\n', PriceAMCAntithetic);
fprintf('Geometric Asian\n');
fprintf('Standard Monte Carlo:           %f\n', PriceGMC);
fprintf('Variate Antithetic Monte Carlo: %f\n', PriceGMCAntithetic);
end

function displayComparisonAsianCallPrices(PriceLevy, PriceTW, PriceHHM, PriceCRR40, PriceAMCAnti
fprintf('Comparison of Asian call prices:\n');
fprintf('\n');
fprintf('Arithmetic Asian\n');
fprintf('Levy:                     %f\n', PriceLevy);
fprintf('Turnbull-Wakeman:         %f\n', PriceTW);
fprintf('Haug-Haug-Margrabe:       %f\n', PriceHHM);
fprintf('Cox-Ross-Rubinstein:      %f\n', PriceCRR40(1));
fprintf('Monte Carlo(500 trials):  %f\n', PriceAMCAntithetic);
fprintf('Monte Carlo(2000 trials): %f\n', PriceAMCAntithetic2000);
fprintf('\n');
fprintf('Geometric Asian\n');
fprintf('Kemna-Vorst:              %f\n', PriceKV);
fprintf('Cox-Ross-Rubinstein:      %f\n', PriceCRR40(2));
fprintf('Monte Carlo(500 trials):  %f\n', PriceGMCAntithetic);
fprintf('Monte Carlo(2000 trials): %f\n', PriceGMCAntithetic2000);
end

function displayComparisonVanillaAsian(type, BLS, KV, Levy, TW, HHM)
fprintf('Comparison of Vanilla and Asian %s:\n', type);
fprintf('\n');
fprintf('Vanilla BLS:              %f\n', BLS);
fprintf('Asian Kemna-Vorst:        %f\n', KV);
fprintf('Asian Levy:               %f\n', Levy);
fprintf('Asian Turnbull-Wakeman:   %f\n', TW);
fprintf('Asian Haug-Haug-Margrabe: %f\n', HHM);
end
```

See Also

asianbykv | asiansensbykv | asianbylevy | asiansensbylevy | asianbyhhm | asiansensbyhhm | asianbytw | asiansensbytw | asianbyls | asiansensbyls | asianbystt | asianbyitt | asianbyeqp | asianbycrr

Related Examples

- “Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135

More About

- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

- How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

Price Spread Instrument for a Commodity Using Black-Scholes Model and Analytic Pricers

This example shows the workflow to price a commodity `Spread` instrument when you use a `BlackScholes` model and `Kirk` and `BjerksundStensland` analytic pricing methods.

Understanding Crack Spread Options

In the petroleum industry, refiners are concerned about the difference between their input costs (crude oil) and output prices (refined products — gasoline, heating oil, diesel fuel, and so on). The differential between these two underlying commodities is referred to as a *crack spread*. It represents the profit margin between crude oil and the refined products.

A *spread option* is an option on the spread where the holder has the right, but not the obligation, to enter into a spot or forward spread contract. Crack spread options are often used to protect against declines in the crack spread or to monetize volatility or price expectations on the spread.

Define the Commodity

Assume that current gasoline prices are strong, and you want to model a crack spread option strategy to protect the gasoline margin. A crack spread option strategy is used to maintain profits for the following season. The WTI crude oil futures are at $93.20 per barrel and RBOB gasoline futures contract are at $2.85 per gallon.

```
Strike = 20;
Rate = 0.05;

Settle = datetime(2020,1,1);
Maturity = datemnth(Settle,3);

% Price and volatility of RBOB gasoline
PriceGallon1 = 2.85;          % Dollars per gallon
Price1 = PriceGallon1 * 42;   % Dollars per barrel
Vol1 = 0.29;

% Price and volatility of WTI crude oil
Price2 = 93.20;         % Dollars per barrel
Vol2 = 0.36;

% Correlation between the prices of the commodities
Corr = 0.42;
```

Create Spread Instrument Object

Use `fininstrument` to create a `Spread` instrument object.

```
SpreadOpt = fininstrument("Spread", 'ExerciseDate', Maturity, 'Strike', Strike,'ExerciseStyle',"e
```

```
SpreadOpt =
  Spread with properties:

       OptionType: "call"
           Strike: 20
    ExerciseStyle: "european"
     ExerciseDate: 01-Apr-2020
```

```
            Name: "spread_instrument"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes", 'Volatility', [Vol1,Vol2], 'Correlation', [1 Corr; C
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
ZeroCurve = ratecurve('zero', Settle, Maturity, Rate, 'Basis', 1);
```

Create BjerksundStensland Pricer Object

Use `finpricer` to create a `BjerksundStensland` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
BJSPricer = finpricer("Analytic", 'Model', BlackScholesModel, 'SpotPrice', [Price1 , Price2], 'Di
```

Create Kirk Pricer Object

Use `finpricer` to create a `Kirk` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
KirkPricer = finpricer("Analytic", 'Model', BlackScholesModel,'SpotPrice', [Price1 , Price2], 'Di
```

Price Spread Instrument Using BjerksundStensland and Kirk Analytic Pricing Methods

Use `price` to compute the price and sensitivities for the commodity `Spread` instrument.

```
[PriceKirk, outPR_Kirk] = price(KirkPricer, SpreadOpt, "all");
[PriceBJS,  outPR_BJS]  = price(BJSPricer,  SpreadOpt, "all");

[outPR_Kirk.Results; outPR_BJS.Results]
```

```
ans=2×7 table
    Price           Delta                Gamma                Lambda               Vega
    _____    ____________________    ____________________    _________________    ____________

    11.19    0.67224    -0.60665    0.019081    0.021662    7.1907    -6.4891    11.299    9.8869
     11.2    0.67371    -0.60816    0.018992    0.021572    7.2003    -6.4997    11.198    9.9878
```

Price Vanilla Instrument Using Heston Model and Multiple Different Pricers

This example shows the workflow to price a `Vanilla` instrument when you use a `Heston` model and various pricing methods.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle,6);
Strike = 80;
VanillaOpt = fininstrument('Vanilla','ExerciseDate',Maturity,'Strike',Strike,'Name',"vanilla_opti
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 29-Dec-2017
           Strike: 80
             Name: "vanilla_option"
```

Create `Heston` Model Object

Use `finmodel` to create a `Heston` model object.

```
V0 = 0.04;
ThetaV = 0.05;
Kappa = 1.0;
SigmaV = 0.2;
RhoSV = -0.7;

HestonModel = finmodel("Heston",'V0',V0,'ThetaV',ThetaV,'Kappa',Kappa,'SigmaV',SigmaV,'RhoSV',Rho
```

```
HestonModel =
  Heston with properties:

         V0: 0.0400
    ThetaV: 0.0500
     Kappa: 1
    SigmaV: 0.2000
     RhoSV: -0.7000
```

Create `ratecurve` object

Create a `ratecurve` object using `ratecurve`.

```
Rate = 0.03;
ZeroCurve = ratecurve('zero',Settle,Maturity,Rate);
```

Create NumericalIntegration, FFT, and FiniteDifference Pricer Objects

Use `finpricer` to create a `NumericalIntegration`, `FFT`, and `FiniteDifference` pricer objects and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
SpotPrice = 80;
Strike = 80;
DividendYield = 0.02;

NIPricer = finpricer("NumericalIntegration",'Model', HestonModel,'SpotPrice',SpotPrice,'DiscountC
```

```
NIPricer =
  NumericalIntegration with properties:

                Model: [1x1 finmodel.Heston]
        DiscountCurve: [1x1 ratecurve]
            SpotPrice: 80
         DividendType: "continuous"
        DividendValue: 0.0200
               AbsTol: 1.0000e-10
               RelTol: 1.0000e-10
     IntegrationRange: [1.0000e-09 Inf]
    CharacteristicFcn: @characteristicFcnHeston
            Framework: "heston1993"
       VolRiskPremium: 0
           LittleTrap: 1
```

```
FFTPricer = finpricer("FFT",'Model',HestonModel, ...
    'SpotPrice',SpotPrice,'DiscountCurve',ZeroCurve, ...
    'DividendValue',DividendYield,'NumFFT',8192)
```

```
FFTPricer =
  FFT with properties:

                    Model: [1x1 finmodel.Heston]
            DiscountCurve: [1x1 ratecurve]
                SpotPrice: 80
             DividendType: "continuous"
            DividendValue: 0.0200
                   NumFFT: 8192
    CharacteristicFcnStep: 0.0100
            LogStrikeStep: 0.0767
        CharacteristicFcn: @characteristicFcnHeston
            DampingFactor: 1.5000
               Quadrature: "simpson"
           VolRiskPremium: 0
               LittleTrap: 1
```

```
FDPricer = finpricer("FiniteDifference",'Model',HestonModel,'SpotPrice',SpotPrice,'DiscountCurve
```

```
FDPricer =
  FiniteDifference with properties:

     DiscountCurve: [1x1 ratecurve]
             Model: [1x1 finmodel.Heston]
         SpotPrice: 80
    GridProperties: [1x1 struct]
```

```
      DividendType: "continuous"
     DividendValue: 0.0200
```

Price `Vanilla` Instrument

Use the following sensitivities when pricing the `Vanilla` instrument.

```
InpSensitivity = ["delta", "gamma", "theta", "rho", "vega", "vegalt"];
```

Use `price` to compute the price and sensitivities for the `Vanilla` instrument that uses the `NumericalIntegration` pricer.

```
[PriceNI,  outPR_NI]  = price(NIPricer,VanillaOpt,InpSensitivity)
```

```
PriceNI = 4.7007

outPR_NI =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

Use `price` to compute the price and sensitivities for the `Vanilla` instrument that uses the `FFT` pricer.

```
[PriceFFT, outPR_FFT] = price(FFTPricer,VanillaOpt,InpSensitivity)
```

```
PriceFFT = 4.7007

outPR_FFT =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

Use `price` to compute the price and sensitivities for the `Vanilla` instrument that uses the `FiniteDifference` pricer.

```
[PriceFD,  outPR_FD]  = price(FDPricer,VanillaOpt,InpSensitivity)
```

```
PriceFD = 4.7003

outPR_FD =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

Aggregate the price results.

```
[outPR_NI.Results;outPR_FFT.Results;outPR_FD.Results]
```

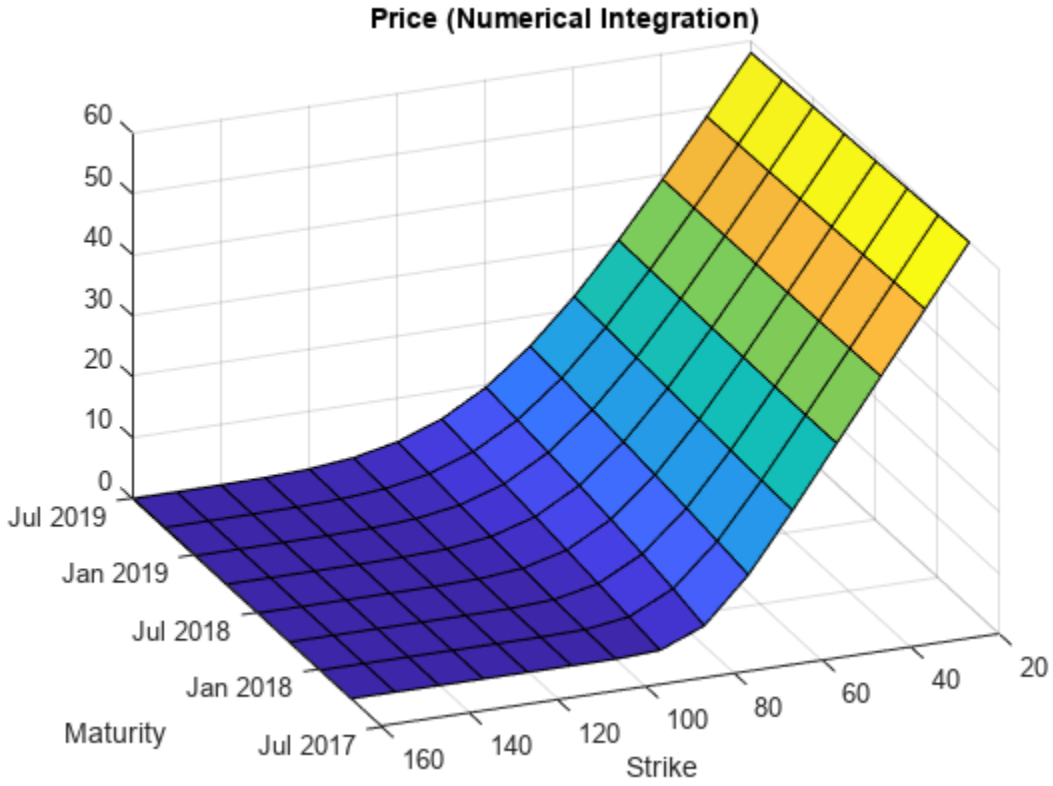
```
ans=3×7 table
    Price      Delta      Gamma       Theta      Rho       Vega      VegaLT
    ______    _______    ________    _______    ______    ______    ______
```

```
    4.7007    0.57747     0.03392    -4.8474    20.805    17.028    5.2394
    4.7007    0.57747     0.03392    -4.8474    20.805    17.028    5.2394
    4.7003    0.57722    0.035254    -4.8483    20.801    17.046    5.2422
```

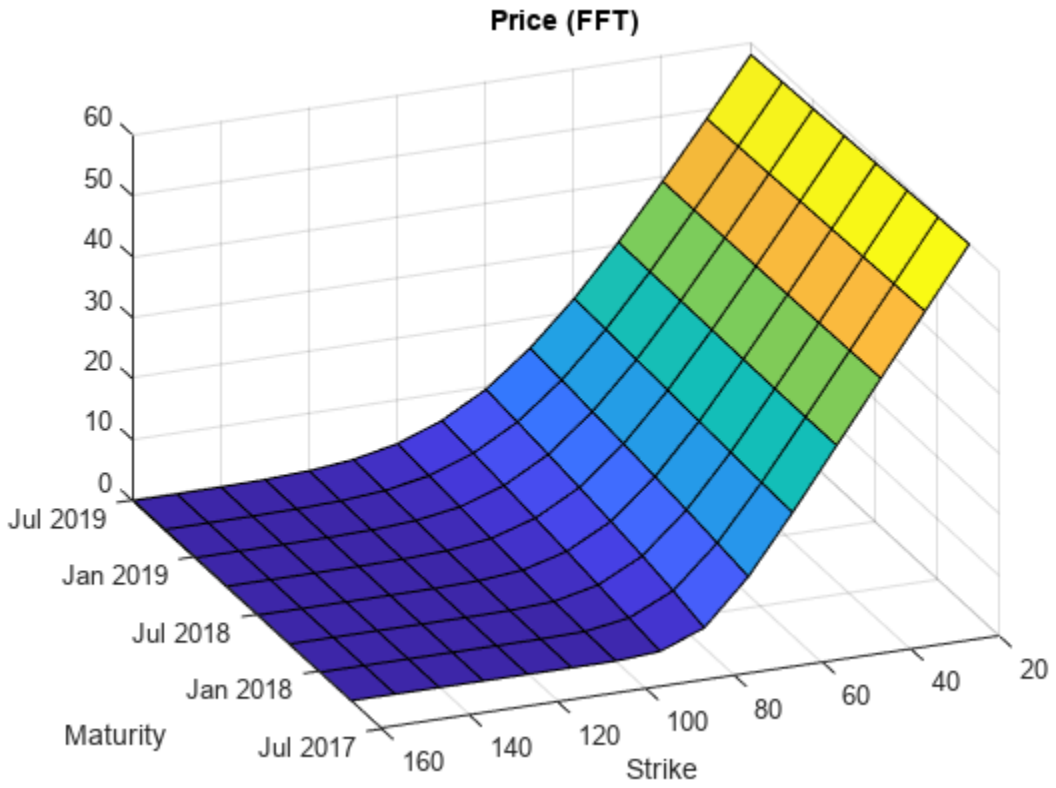
Compute Option Price Surfaces

Use the `price` function for the `NumericalIntegration` pricer and the `price` function for the `FFT` pricer to compute the prices for a range of `Vanilla` instruments.

```
Maturities = datemnth(Settle,(3:3:24)');
NumMaturities = length(Maturities);
Strikes = (20:10:160)';
NumStrikes = length(Strikes);

[Maturities_Full,Strikes_Full] = meshgrid(Maturities,Strikes);

NumInst = numel(Strikes_Full);
VanillaOptions(NumInst, 1) = fininstrument("vanilla",...
    "ExerciseDate", Maturities_Full(1), "Strike", Strikes_Full(1));
for instidx=1:NumInst
    VanillaOptions(instidx) = fininstrument("vanilla",...
        "ExerciseDate", Maturities_Full(instidx), "Strike", Strikes_Full(instidx));
end

Prices_NI = price(NIPricer, VanillaOptions);
Prices_FFT = price(FFTPricer, VanillaOptions);

figure;
surf(Maturities_Full,Strikes_Full,reshape(Prices_NI,[NumStrikes,NumMaturities]));
title('Price (Numerical Integration)');
view(-112,34);
xlabel('Maturity')
ylabel('Strike')
```




```
figure;
surf(Maturities_Full,Strikes_Full,reshape(Prices_FFT,[NumStrikes,NumMaturities]));
title('Price (FFT)');
view(-112,34);
xlabel('Maturity')
ylabel('Strike')
```


Price (FFT)
60
50
40
30
20
10
0
Jul 2019
Jan 2019
Jul 2018
Jan 2018
Jul 2017
Maturity
160
140
120
100
80
60
40
20
Strike

Create and Price Portfolio of Instruments

Use `finportfolio` and `pricePortfolio` to create and price a portfolio of interest-rate and equity instruments. The portfolio contains a vanilla `FixedBond`, an `OptionEmbeddedFixedBond`, a `Vanilla` European call option, a `Vanilla` American call option, and an `Asian` call option.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates);
```

Create the Instrument Objects

Use `fininstrument` to create the instrument objects.

```
% Vanilla FixedBond
CouponRate = 0.0325;
Maturity = datetime(2038,3,15);
Period = 1;
VanillaBond = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',CouponRate,...
    'Period',Period,'Name',"VanillaBond")
```

```
VanillaBond =
  FixedBond with properties:

                 CouponRate: 0.0325
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Mar-2038
                        Name: "VanillaBond"
```

```
% OptionEmbeddedBond
Maturity = datetime(2024,9,15);
CouponRate = 0.035;
Strike = 100;
ExerciseDates = datetime(2023,9,15);
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
Period = 1;
CallableBond = fininstrument("OptionEmbeddedFixedBond", "Maturity",Maturity,...
    'CouponRate',CouponRate,'Period',Period, ...
    'CallSchedule',CallSchedule,...
```

```
    'Name',"CallableBond");

% Vanilla European call option
ExerciseDate = datetime(2022,1,1);
Strike = 96;
OptionType = 'call';
CallOpt = fininstrument("Vanilla",'ExerciseDate',ExerciseDate,'Strike',Strike,...
    'OptionType',OptionType, 'Name',"EuropeanCallOption")

CallOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 01-Jan-2022
           Strike: 96
             Name: "EuropeanCallOption"



% Vanilla American call option
ExerciseDate = datetime(2023,1,1);
Strike = 97;
OptionType = 'call';
CallOpt_American = fininstrument("Vanilla",'ExerciseDate',ExerciseDate,'Strike',Strike,...
    'OptionType',OptionType, 'ExerciseStyle', "american", ...
    'Name',"AmericanCallOption")

CallOpt_American =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2023
           Strike: 97
             Name: "AmericanCallOption"



% Asian call option
ExerciseDate = datetime(2023,1,1);
Strike = 102;
OptionType = 'call';
CallOpt_Asian = fininstrument("Asian",'ExerciseDate',ExerciseDate,'Strike',Strike,...
    'OptionType',OptionType,'Name',"AsianCall")

CallOpt_Asian =
  Asian with properties:

         OptionType: "call"
             Strike: 102
        AverageType: "arithmetic"
       AveragePrice: 0
   AverageStartDate: NaT
      ExerciseStyle: "european"
       ExerciseDate: 01-Jan-2023
               Name: "AsianCall"
```

Create Model Objects

Use `finmodel` to create `HullWhite` and `BlackScholes` model objects.

```
% Create Hull-White model
Vol = 0.01;
Alpha = 0.1;
HWModel = finmodel("hullwhite",'alpha',Alpha,'sigma',Vol);

% Create Black-Scholes model
Vol = .1;
SpotPrice = 95;
BlackScholesModel = finmodel("BlackScholes",'Volatility',Vol);
```

Create Pricer Objects

Use `finpricer` to create `Discount`, `IRTree`, `BlackScholes`, `Levy`, and `BjerksundStensland` pricer objects and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
% Create Discount pricer
DiscPricer = finpricer("Discount","DiscountCurve",ZeroCurve);

% Create Hull-White tree pricer
TreeDates = Settle + calyears(1:30);
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,...
    'TreeDates',TreeDates');

% Create BlackScholes, Levy, and BjerksundStensland pricers
BLSPricer = finpricer("analytic",'DiscountCurve',ZeroCurve,'Model',BlackScholesModel,'SpotPrice'
LevyPricer = finpricer("analytic",'DiscountCurve',ZeroCurve,'Model',BlackScholesModel,...
                       'SpotPrice',SpotPrice,'PricingMethod',"Levy");
BJSpricer = finpricer("analytic",'DiscountCurve',ZeroCurve,'Model',BlackScholesModel,...
                       'SpotPrice',SpotPrice,'PricingMethod',"BjerksundStensland");
```

Create `finportfolio` Object

Create a `finportfolio` object that contains all of the instrument and pricer objects using `finportfolio`.

```
myPort = finportfolio([VanillaBond CallableBond CallOpt CallOpt_American CallOpt_Asian]',...
                           [DiscPricer HWTreePricer BLSPricer BJSpricer LevyPricer]')
```

```
myPort =
  finportfolio with properties:

    Instruments: [5x1 fininstrument.FinInstrument]
        Pricers: [5x1 finpricer.FinPricer]
    PricerIndex: [5x1 double]
       Quantity: [5x1 double]
```

Price Portfolio

Use `pricePortfolio` to compute the price and sensitivities for the portfolio and the instruments in the portfolio.

```
[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(myPort)
```

```
PortPrice = 237.3275
```

```
InstPrice = 5×1

  107.4220
  110.8389
    7.5838
    8.8705
    2.6123


PortSens=1×8 table
    Price      Delta     Gamma    Lambda     Vega      Theta      Rho       DV01
    _____    _______    _____    ______    _____    _______    _____    ______

    237.33    -546.39    2840     26.354    124.28    -4.0673    418.68    0.1579


InstSens=5×8 table
                          Price      Delta      Gamma      Lambda     Vega       Theta       Rho
                          _____    _______    ________    ______    _____    ________    _____

    VanillaBond           107.42       NaN         NaN       NaN       NaN         NaN       Na
    CallableBond          110.84    -547.9      2839.9       NaN    -62.532         NaN       Na
    EuropeanCallOption    7.5838    0.57026    0.022762    7.1435    67.763     -1.3962    153.6
    AmericanCallOption    8.8705     0.5845    0.019797    6.2597    76.808     -1.8677    200.6
    AsianCall             2.6123    0.35611    0.032053     12.95    42.238    -0.80342    64.3
```

Use Black-Scholes Model to Price Asian Options with Several Equity Pricers

This example shows how to compare arithmetic and geometric Asian option prices using the `BlackScholes` model and various pricing methods. The pricing methods are: the Kemna-Vorst, Levy, Turnbull-Wakeman, and Cox-Ross-Rubinstein methods and Monte Carlo simulation. This example also demonstrates how variations in spot prices affect option and delta sensitivity values on European vanilla and Asian options.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,01,01);
Maturity = datetime(2025,01,01);
Rate = 0.035;
Compounding = -1;
Basis = 1;
ZeroCurve = ratecurve('zero',Settle,Maturity,Rate,'Compounding',Compounding,'Basis', Basis)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2025
               Rates: 0.0350
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
Volatility = .20;
BSModel = finmodel("BlackScholes",'Volatility',Volatility)
```

```
BSModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `Asian` Instrument Objects

Use `fininstrument` to create two `Asian` instrument objects, one using an arithmetic average and the other using a geometric average.

```
ExerciseDates = datetime(2020,01,01);
Strike = 90;
OptionType = 'call';
```

```
AverageType = 'geometric';

AsianOptArith = fininstrument("Asian",'ExerciseDate',ExerciseDates,'Strike',Strike,...
                              'OptionType',OptionType,'Name',"CallAsianArith")

AsianOptArith =
  Asian with properties:

          OptionType: "call"
              Strike: 90
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 01-Jan-2020
                Name: "CallAsianArith"


AsianOptGeo = fininstrument("Asian",'ExerciseDate',ExerciseDates,'Strike',Strike,...
                            'OptionType',OptionType,'AverageType', AverageType,'Name',"CallAsian

AsianOptGeo =
  Asian with properties:

          OptionType: "call"
              Strike: 90
         AverageType: "geometric"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 01-Jan-2020
                Name: "CallAsianGeo"
```

Create `Analytic`, `AssetTree`, and `AssetMonteCarlo` Pricer Objects

Use `finpricer` to create `BlackScholes`, `AssetTree`, and `AssetMonteCarlo` pricer objects and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
SpotPrice = 100;
InpSensitivity = "delta";

% Analytic Pricers
LevyPricer = finpricer('Analytic', 'Model', BSModel, 'SpotPrice', SpotPrice, ...
                       'DiscountCurve', ZeroCurve, 'PricingMethod', "Levy");

TWPricer = finpricer('Analytic', 'Model', BSModel, 'SpotPrice', SpotPrice, ...
                     'DiscountCurve', ZeroCurve, 'PricingMethod', "TurnbullWakeman");

KVPricer = finpricer('Analytic', 'Model', BSModel, 'SpotPrice', SpotPrice, ...
                       'DiscountCurve', ZeroCurve, 'PricingMethod', "KemnaVorst");

% AssetTree Pricer
% Define the number of levels of the tree
NumPeriods = 50;
CRRPricer = finpricer("AssetTree",'DiscountCurve',ZeroCurve,'Model',BSModel, 'SpotPrice',SpotPric
                      'PricingMethod',"CoxRossRubinstein",'NumPeriods', NumPeriods,...
                      'Maturity', ExerciseDates);
```

```
% AssetMonteCarlo Pricer
% Define the number of simulation trials
NumTrials = 2000;
SimDates =[Settle:days(2):ExerciseDates ExerciseDates];

MCPricer = finpricer("AssetMonteCarlo", 'Model', BSModel, 'SpotPrice', SpotPrice, 'DiscountCurve
                     'SimulationDates',  SimDates, 'NumTrials', NumTrials);
```

Calculate the Price of the Arithmetic and Geometric Asian Options Using Different Pricers

Calculate the Asian option prices using the `price` function for the `Analytic`, `AssetTree`, and `AssetMonetCarlo` pricing methods.

```
% Analytic
[LevyPrice,LevyoutPR] = price(LevyPricer, AsianOptArith,InpSensitivity);

[TWPrice, TWoutPR] = price(TWPricer, AsianOptArith, InpSensitivity);

[KVPrice, KVoutPR] = price(KVPricer, AsianOptGeo, InpSensitivity);

% Cox-Ross-Rubinstein
[CRRArithPrice, CRRArithoutPR] = price(CRRPricer, AsianOptArith, InpSensitivity);

[CRRGeoPrice, CRRGeooutPR] = price(CRRPricer, AsianOptGeo, InpSensitivity);

% Monte Carlo
[MCArithPrice, MCArithoutPR] = price(MCPricer, AsianOptArith, InpSensitivity);

[MCGeoPrice, MCGeooutPR] = price(MCPricer, AsianOptGeo, InpSensitivity);
```

Compare Asian Option Prices

Compare the Asian option call prices using the `displayPricesAsianCallOption` function defined in Local Functions on page 3-141.

```
displayPricesAsianCallOption(KVPrice,LevyPrice,TWPrice,CRRArithPrice,CRRGeoPrice,MCArithPrice,MC
```

```
Comparison of Asian prices:

Arithmetic Asian
Levy:                    12.164734
Turnbull-Wakeman:        12.164734
Cox-Ross-Rubinstein:     12.126509
Monte Carlo:             12.102669

Geometric Asian
Kemna-Vorst:             11.862580
Cox-Ross-Rubinstein:     11.852462
Monte Carlo:             11.988051
```

The table contrasts the results from closed approximation models against price simulations implemented using the Cox-Ross-Rubinstein binomial tree and Monte Carlo pricing methods. Observe that arithmetic average Asian options are more expensive than their geometric average counterparts.

Compare Asian and Vanilla Options

Asian options are popular instruments since they tend to be less expensive than comparable vanilla calls and puts. This is because the volatility in the average value of an underlier tends to be lower

than the volatility of the value of the underlier itself. You can compare the price and delta sensitivity values of Asian options against their vanilla counterparts.

Create Vanilla Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object with same Maturity and Strike as the two Asian options.

```
EuropeanCallOption = fininstrument("Vanilla",'ExerciseDate',ExerciseDates,'Strike',Strike,...
                                   'OptionType',OptionType,'Name',"CallVanilla")
```

```
EuropeanCallOption =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 01-Jan-2020
           Strike: 90
             Name: "CallVanilla"
```

Create `Analytic` Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
BLSPricer = finpricer('Analytic', 'Model', BSModel, 'SpotPrice', SpotPrice, 'DiscountCurve', Zero
```

```
BLSPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0
     DividendType: "continuous"
```

Compute price and delta sensitivity.

```
[BLSPrice, BLSoutPR] = price(BLSPricer, EuropeanCallOption, InpSensitivity);
```

Compare Prices for Asian and Vanilla Options

Compare option prices using the `displayVanillaAsianComparison` function defined in Local Functions on page 3-141.

```
displayVanillaAsianComparison('Prices', BLSPrice, KVPrice,LevyPrice,TWPrice)
```

```
Comparison of Vanilla and Asian Option Prices:

Vanilla BLS:              15.743809
Asian Kemna-Vorst:        11.862580
Asian Levy:               12.164734
Asian Turnbull-Wakeman:   12.164734
```

Observe that both the geometric and arithmetic Asian option prices are lower than their vanilla counterparts.

Compare Delta Sensitivity for Asian and Vanilla Options

The delata value measures the option price sensitivity to changes in the price of the underlying asset. As the underlier changes, the proportions of the instruments forming the portfolio might need to be adjusted to keep the sensitivities within the desired range.

```
displayVanillaAsianComparison('Delta', BLSoutPR.Results.Delta, KVoutPR.Results.Delta, LevyoutPR.R
```

```
Comparison of Vanilla and Asian Option Delta:

Vanilla BLS:              0.788666
Asian Kemna-Vorst:        0.844986
Asian Levy:               0.852806
Asian Turnbull-Wakeman:   0.852806
```

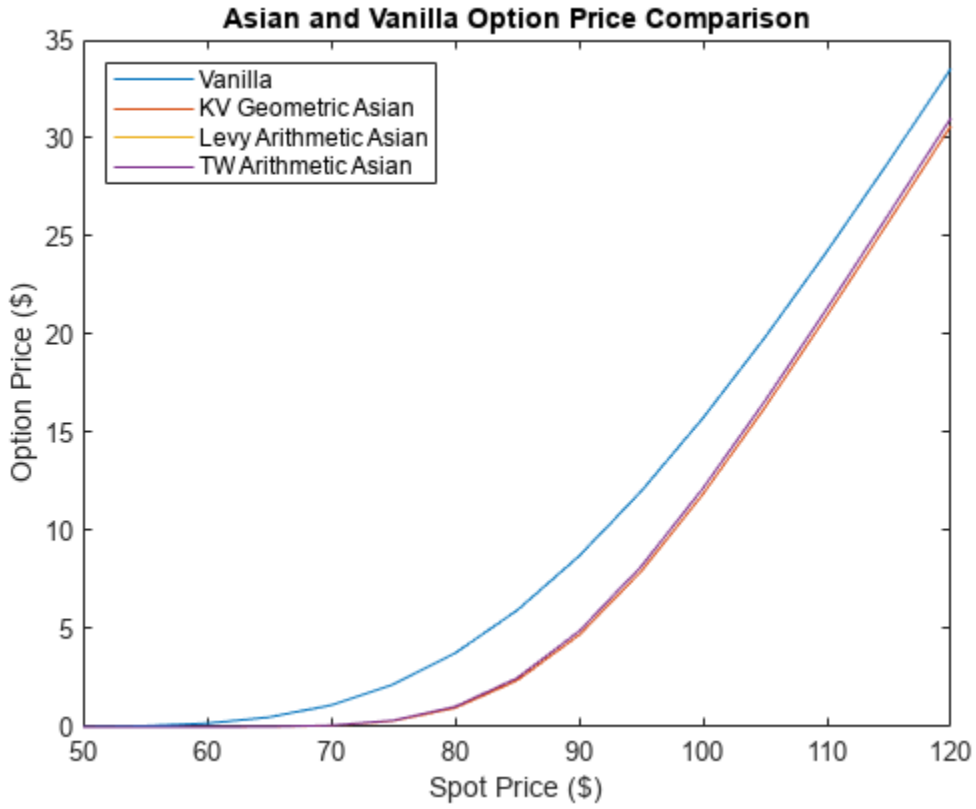
The table shows the delta values for both the vanilla and arithmetic and geometric Asian options. Observe that the geometric Asian delta value is lower than the delta value for the arithmetic Asian option.

Analyze Effect of Variations of Underlying Asset on Option Prices

Examine the effect of changes of underlying asset prices. Create a plot to show the effect of variations in the price of the underlying asset on the vanilla and Asian option prices.

```
StockPrices = (50:5:120)';
PriceBLS = nan(size(StockPrices));
PriceKV = PriceBLS;
PriceLevy = PriceBLS;
PriceTW = PriceBLS;
DeltaBLS = PriceBLS;
DeltaLevy = PriceBLS;
DeltaTW = PriceBLS;
DeltaKV = PriceBLS;
InpSensitivity = "delta";
idx = 1;
for AssetPrice = StockPrices'

    PricerBLS = finpricer('Analytic', 'Model', BSModel, 'SpotPrice', AssetPrice,...
                          'DiscountCurve', ZeroCurve);
    [PriceBLS(idx), outPRBLS] = price(PricerBLS, EuropeanCallOption, InpSensitivity);
    DeltaBLS(idx) = outPRBLS.Results.Delta;

    PricerLevy = finpricer('Analytic', 'Model', BSModel, 'SpotPrice', AssetPrice, ...
                           'DiscountCurve', ZeroCurve, 'PricingMethod', "Levy");
    [PriceLevy(idx), outPRLevy] = price(PricerLevy, AsianOptArith, InpSensitivity);
    DeltaLevy(idx) = outPRLevy.Results.Delta;

    PricerTW = finpricer('Analytic', 'Model', BSModel, 'SpotPrice', AssetPrice, ...
        'DiscountCurve', ZeroCurve, 'PricingMethod', "TurnbullWakeman");
    [PriceTW(idx), outPRBTW] = price(PricerTW, AsianOptArith, InpSensitivity);
    DeltaTW(idx) = outPRBTW.Results.Delta;

    PricerKV = finpricer('Analytic', 'Model', BSModel, 'SpotPrice', AssetPrice, ...
        'DiscountCurve', ZeroCurve, 'PricingMethod', "KemnaVorst");
    [PriceKV(idx), outPRKV] = price(PricerKV, AsianOptGeo, InpSensitivity);
    DeltaKV(idx) = outPRKV.Results.Delta;

    idx = idx+1;
```

```
end

figure('menubar', 'none', 'numbertitle', 'off')
plot(StockPrices, [PriceBLS PriceKV PriceLevy PriceTW]);
xlabel 'Spot Price ($)'
ylabel 'Option Price ($)'
title 'Asian and Vanilla Option Price Comparison'
legend('Vanilla', 'KV Geometric Asian', 'Levy Arithmetic Asian', 'TW Arithmetic Asian', 'locatio
```

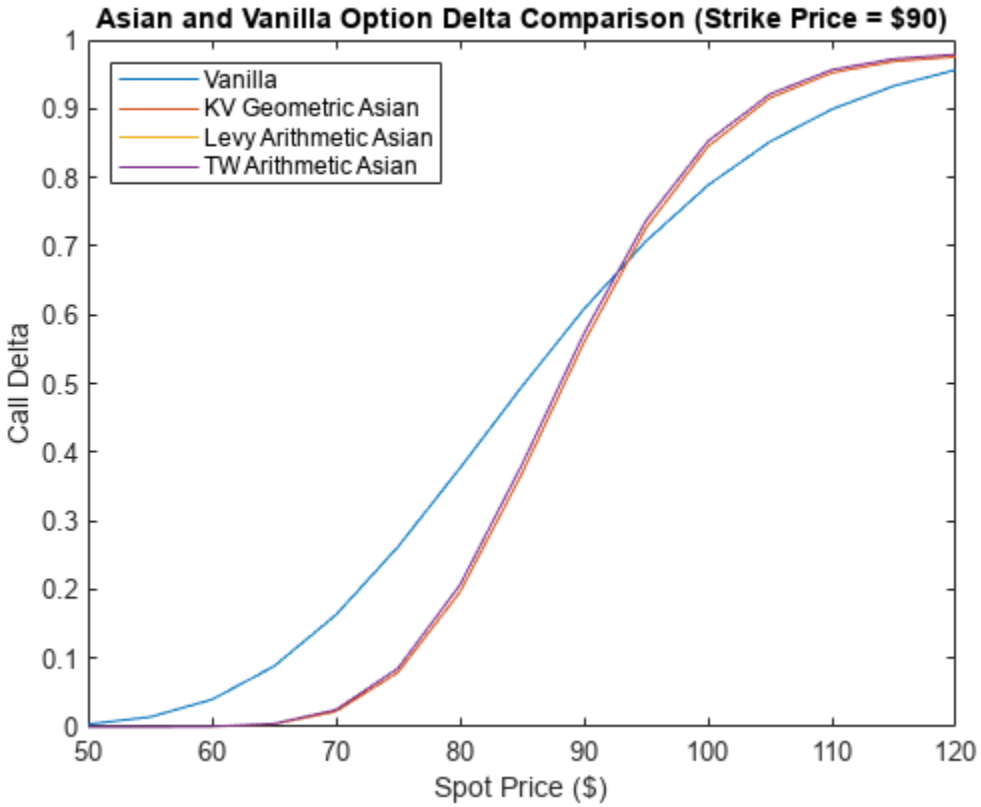


The plot displays vanilla and Asian option prices with respect to the underlying asset price. Observe that the price of the Asian options is cheaper than the price of the vanilla option.

Analyze Effect of Variations of the Underlying Asset on the Options Delta

Examine the effect of changes of underlying asset prices on delta sensitivity. The following plot demonstrates the behavior of the delta value for the Vanilla and Asian options as a function of the underlying price.

```
figure('menubar', 'none', 'numbertitle', 'off')
plot(StockPrices, [DeltaBLS DeltaKV DeltaLevy DeltaTW]);
xlabel 'Spot Price ($)'
ylabel 'Call Delta'
title 'Asian and Vanilla Option Delta Comparison (Strike Price = $90)'
legend('Vanilla', 'KV Geometric Asian', 'Levy Arithmetic Asian', 'TW Arithmetic Asian', 'locatio
```



The plot displays the vanilla and Asian delta sensitivity values with respect to the underlying asset price. A vanilla or Asian in-the-money (ITM) call option is more sensitive to price movements than an out-of-the-money (OTM) option. If the asset price is deep in the money, then it is more likely to be exercised. The opposite is the case for an out-of-the-money option. Observe that the Asian delta value is lower for out-of-the-money options and higher for in-the-money options compared with the vanilla European counterpart.

Local Functions

```
function displayPricesAsianCallOption(KVPrice,LevyPrice,TWPrice,CRRArithPrice,CRRGeoPrice,MCArith
fprintf('Comparison of Asian prices:\n');
fprintf('\n');
fprintf('Arithmetic Asian\n');
fprintf('Levy:                    %f\n', LevyPrice);
fprintf('Turnbull-Wakeman:        %f\n', TWPrice);
fprintf('Cox-Ross-Rubinstein:     %f\n', CRRArithPrice);
fprintf('Monte Carlo:             %f\n', MCArithPrice);
fprintf('\n');
fprintf('Geometric Asian\n');
fprintf('Kemna-Vorst:             %f\n', KVPrice);
fprintf('Cox-Ross-Rubinstein:     %f\n', CRRGeoPrice);
fprintf('Monte Carlo:             %f\n', MCGeoPrice);
end

function displayVanillaAsianComparison(type, BLS, KV, Levy, TW)
fprintf('Comparison of Vanilla and Asian Option %s:\n', type);
fprintf('\n');
```

```
fprintf('Vanilla BLS:              %f\n', BLS);
fprintf('Asian Kemna-Vorst:        %f\n', KV);
fprintf('Asian Levy:               %f\n', Levy);
fprintf('Asian Turnbull-Wakeman:   %f\n', TW);
end
```

See Also

Related Examples

- “Pricing Asian Options” on page 3-110

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53
- “Supported Exercise Styles” on page 1-62

External Websites

- How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

Calibrate Option Pricing Model Using Heston Model

This example shows how to use the **Calibrate Pricing Model** Live Editor task to calibrate a `Heston` pricing model to call option prices from the market. After calibration, use the Financial Instruments Toolbox™ object-based workflow to price an American option for a `Barrier` instrument using the calibrated parameter values for the `Heston` model with an `AssetMonteCarlo` pricing method.

Define Pricing Data

Define the data.

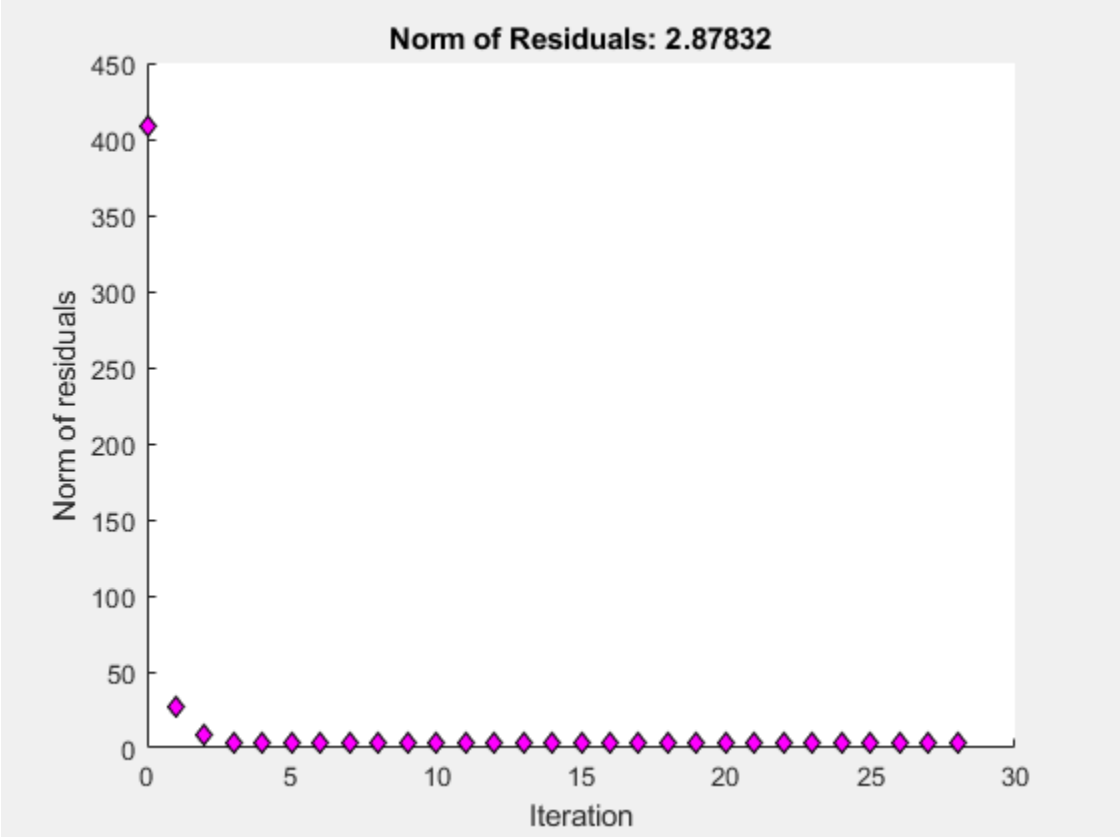
```
Settle = datetime(2015,7,10);
SpotPrice = 123.28;
Rate = -0.001;
MaturityDates = datetime([2015,8,21; 2015,9,18; 2015,12,18; 2016,4,15; 2016,6,17; 2017,1,20]);

Strikes = [115 120 125 130 135 140 145]';

Prices = [9.95 10.63 12.84 15.10 15.95 20.00; ...
    6.30 7.20 9.90 12.30 13.57 17.50; ...
    3.60 4.55 7.30 9.70 11.15 15.20; ...
    1.82 2.68 5.30 7.70 9.00 13.20; ...
    0.82 1.45 3.70 5.85 7.20 11.27; ...
    0.36 0.77 2.50 4.48 5.76 9.65; ...
    0.15 0.38 1.70 3.44 4.54 8.10];

ZeroCurve = ratecurve("zero", Settle, MaturityDates(end), Rate)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: 20-Jan-2017
             Rates: -1.0000e-03
            Settle: 10-Jul-2015
      InterpMethod: "linear"
  ShortExtrapMethod: "next"
   LongExtrapMethod: "previous"
```

Use the Calibrate Pricing Model live task to interactively select the data, model, parameter constraints, and the optimization and solver options to generate a volatility surface plot.

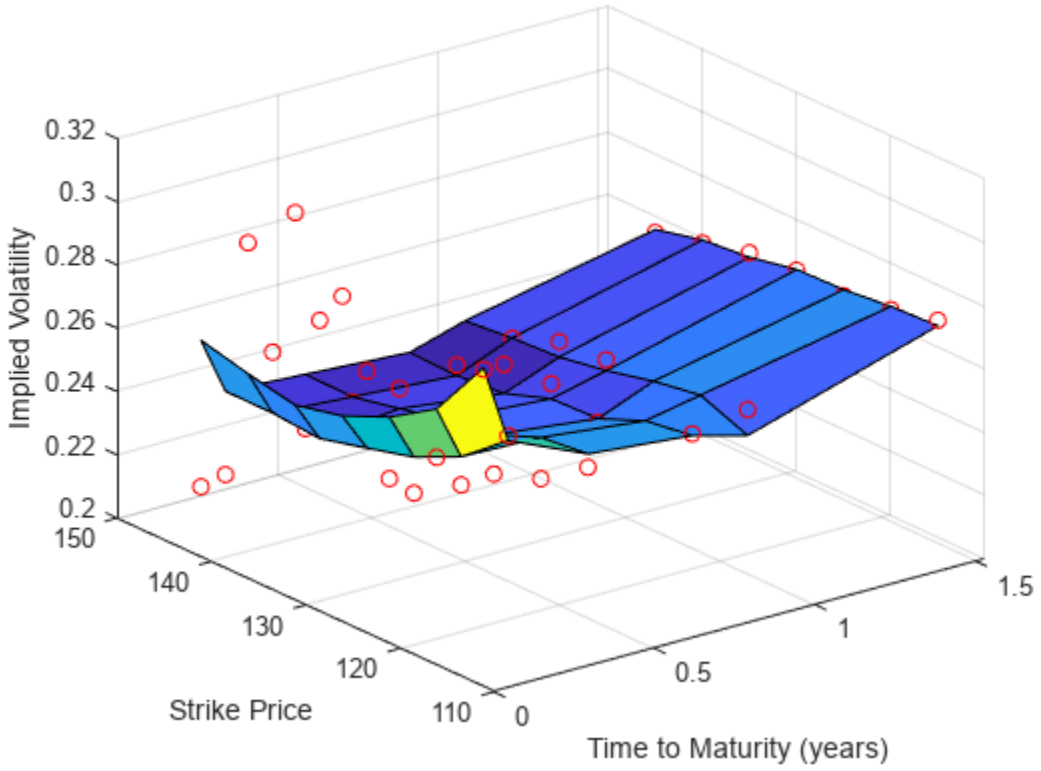
```
% Create fininstrument objects
[MAT, STR] = meshgrid(MaturityDates', Strikes);
Inst = fininstrument('Vanilla', 'ExerciseDate', MAT(:), ...
    'Strike', STR(:), 'OptionType', 'Call');

% Construct objective function
objectiveFcn = @(Param) Prices(:) - price(finpricer('FFT', 'Model', ...
    finmodel('Heston', 'V0', Param(1), 'ThetaV', Param(2), ...
    'Kappa', Param(3), 'SigmaV', Param(4), 'RhoSV', Param(5)), ...
    'SpotPrice', 123.28, 'DiscountCurve', ZeroCurve), Inst);
```

```
% Estimate model parameters
options = optimoptions('lsqnonlin', 'FunctionTolerance', 0.0001, ...
    'Display', 'final', 'PlotFcn', 'optimplotresnorm');
Param = lsqnonlin(objectiveFcn, [0.1 0.4 0.2 0.6 -0.1], [0 0 0 0 -1], ...
    [1 1 10 2 1], options);
```




```
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.


HestonModel = finmodel('Heston', 'V0', Param(1), 'ThetaV', Param(2), ...
    'Kappa', Param(3), 'SigmaV', Param(4), 'RhoSV', Param(5))

HestonModel =
  Heston with properties:

        V0: 0.0448
    ThetaV: 0.1625
     Kappa: 0.3317
    SigmaV: 0.0776
     RhoSV: -0.8401


% Calculate implied volatilities for market and model prices
TMAT = yearfrac(ZeroCurve.Settle, MAT);
```

```
MKTVOL = blsimpv(123.28, STR, ZeroCurve.Rates(1), TMAT, Prices);

ModelPrice = price(finpricer('FFT', 'Model', HestonModel, 'SpotPrice', 123.28, ...
    'DiscountCurve', ZeroCurve), Inst);
ModelVol = blsimpv(123.28, STR(:), ZeroCurve.Rates(1), TMAT(:), ModelPrice);

% Plot implied volatility surface
figure
surf(TMAT, STR, MKTVOL)
hold on
scatter3(TMAT(:), STR(:), ModelVol, 'ro')
xlabel('Time to Maturity (years)')
ylabel('Strike Price')
zlabel('Implied Volatility')
hold off
grid on
```




```
clearvars TMAT MKTVOL ModelPrice ModelVol
clearvars STR MAT Inst Param options objectiveFcn
```

Continue with this workflow to price an American option for a `Barrier` instrument using the calibrated parameter values for a `Heston` model with an `AssetMonteCarlo` pricing method.

Create `Barrier` Instrument Object

Use `fininstrument` to create a `Barrier` instrument object.

```
ExerciseDate = datetime(2016,1,1);
BarrierOpt = fininstrument("Barrier",Strike=90,ExerciseDate=ExerciseDate,OptionType="call",Exerci

BarrierOpt =
  Barrier with properties:

       OptionType: "call"
           Strike: 90
      BarrierType: "do"
     BarrierValue: 40
           Rebate: 0
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2016
             Name: "barrier_option"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object `ZeroCurve` for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",DiscountCurve=ZeroCurve,Model=HestonModel,SpotPrice=100,S

outPricer =
  HestonMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 100
    SimulationDates: [11-Jul-2015    16-Jul-2015    21-Jul-2015    ...    ]
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Heston]
       DividendType: "continuous"
      DividendValue: 0
```

Price `Barrier` Instrument

Use `price` to compute the price and sensitivities for the `Barrier` instrument.

```
[Price,outPR] = price(outPricer,BarrierOpt,"all")

Price = 12.4688

outPR =
  priceresult with properties:

       Results: [1x8 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×8 table
    Price      Delta      Gamma      Lambda      Rho       Theta       Vega      VegaLT
    ______    _______    ________    ______    ______    ________    ______    ______
```

```
    12.469    0.94837    -0.24723    7.6059    297.84    -46.735    17.591    2.6804
```

See Also

Functions
Heston | Bates | Merton | ratecurve | AssetMonteCarlo | Barrier | lsqnonlin | simulannealbnd

Related Examples

- “Use Deep Learning to Approximate Barrier Option Prices with Heston Model” on page 3-148

Use Deep Learning to Approximate Barrier Option Prices with Heston Model

This example shows how to use Deep Learning Toolbox™ to train a network and obtain predictions on barrier option prices with a Heston model.

Barrier Option

The barrier option is an option where the payoff depends on whether the underlying asset crosses the predetermined trigger value (barrier level) during the life of the option. Barrier options are attractive because they are less expensive than the corresponding vanilla options.

Heston Model

The Heston model is an extension of the Black-Scholes model, where the volatility (square root of variance) is no longer assumed to be constant, and the variance follows a stochastic (CIR) process. The Heston model allows modeling the implied volatility smiles observed in the market where options with identical expiration dates show increasing volatility as the options become more in-the-money (ITM) or out-of-the-money (OTM).

The stochastic differential equation is:

$$dS_t = rS_t dt + \sqrt{v_t} S_t dW_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$

where

r — Continuous risk-free rate

S_t — Asset price at time t

v_t — Asset price variance at time t

v_0 — Initial variance of the asset price at $t = 0$

θ — Long-term variance level

κ — Mean reversion speed for the variance

σ_v — Volatility of the variance

ρ — Correlation between the Wiener processes W_t and W_t^v

Barrier option prices are usually computed using Monte Carlo simulation in the Heston setting since there is no closed-form solution available. However, a Monte Carlo simulation is computationally expensive, and when pricing instruments for financial markets, pricing speed is crucial. This example demonstrates using a vanilla neural network to speed up the barrier option pricing by learning the results from a Monte Carlo simulation. The neural network provides a highly efficient approximation technique. Although the off-line training is time consuming, the on-line pricing is fast.

Define Parameters

Focusing on a barrier-up, knock-out call option, start by deciding on the ranges for the pricing parameters. Consider a scaled spot price (moneyness) instead of two separate parameters S_0 (asset spot price) and K(strike). The barrier level is also scaled by strike value.

```
% Option parameter ranges.
% The first value defines the lower bound
% and the second value is the upper bound.
moneyness = [0.6 1.2]; % S0/K
maturity = [0.05 2];
upBarrier = [0.6 1.3]; % Up barrier/K

% Model parameter ranges
rate = [0 0.05];
kappa = [0.3 2];
theta = [0.05 0.2];
sigma = [0.05 0.5];
v0 = [0.05 0.2];
rho = [-0.9 -0.1];

% Call option parameters
optSpec = "call";
exerciseStyle = "european";
barriertype = "uo";

% Simulation parameter
nTrials = 1000;

% Set the random generator seed for reproducibility.
%rng('default')
```

Gather Data

Sample the parameter combinations by using a quasi-Monte Carlo sampling method (`sobolset`) that is based on Sobol sequences which possess good uniformity properties. A Sobol sequence uses a base of 2 to form successively finer uniform partitions of the unit interval, and then reorders the coordinates in each dimension.

```
Quasi = sobolset(9,'Skip',1024);
Quasi = scramble(Quasi,'MatousekAffineOwen');
inputs = Quasi(1:24e4, :); % Initial 240000 samples

% Column number for each parameter in the inputs array.
iMoneyness = 1;iTime = 2;iRate = 3;iCorr = 4;iKappa = 5;iTheta = 6;iSigma = 7;iV0 = 8;iBarrier =

inputs(:,iMoneyness) = inputs(:,iMoneyness)*(moneyness(2)-moneyness(1))+moneyness(1);   % Moneyne
inputs(:,iTime)      = inputs(:,iTime)*(maturity(2)-maturity(1))+maturity(1);           % Maturi
inputs(:,iRate)      = inputs(:,iRate)*(rate(2)-rate(1))+rate(1);                       % rate
inputs(:,iCorr)      = inputs(:,iCorr)*(rho(2)-rho(1))+rho(1);                          % Correla
inputs(:,iKappa)     = inputs(:,iKappa)*(kappa(2)-kappa(1))+kappa(1);                   % Mean re
inputs(:,iTheta)     = inputs(:,iTheta)*(theta(2)-theta(1))+theta(1);                   % Long-te
inputs(:,iSigma)     = inputs(:,iSigma)*(sigma(2)-sigma(1))+sigma(1);                   % Volati
inputs(:,iV0)        = inputs(:,iV0)*(v0(2)-v0(1))+v0(1);                               % Initia
inputs(:,iBarrier)   = inputs(:,iBarrier)*(upBarrier(2)-upBarrier(1))+upBarrier(1);     % UPbarri
```

Remove the parameter combinations where the barrier levels are not greater than the initial spot prices.

```
% Barrier level should be higher than the initial spot price.
invalid = inputs(:,iBarrier)<=inputs(:,iMoneyness);
inputs(invalid,:) = [];
```

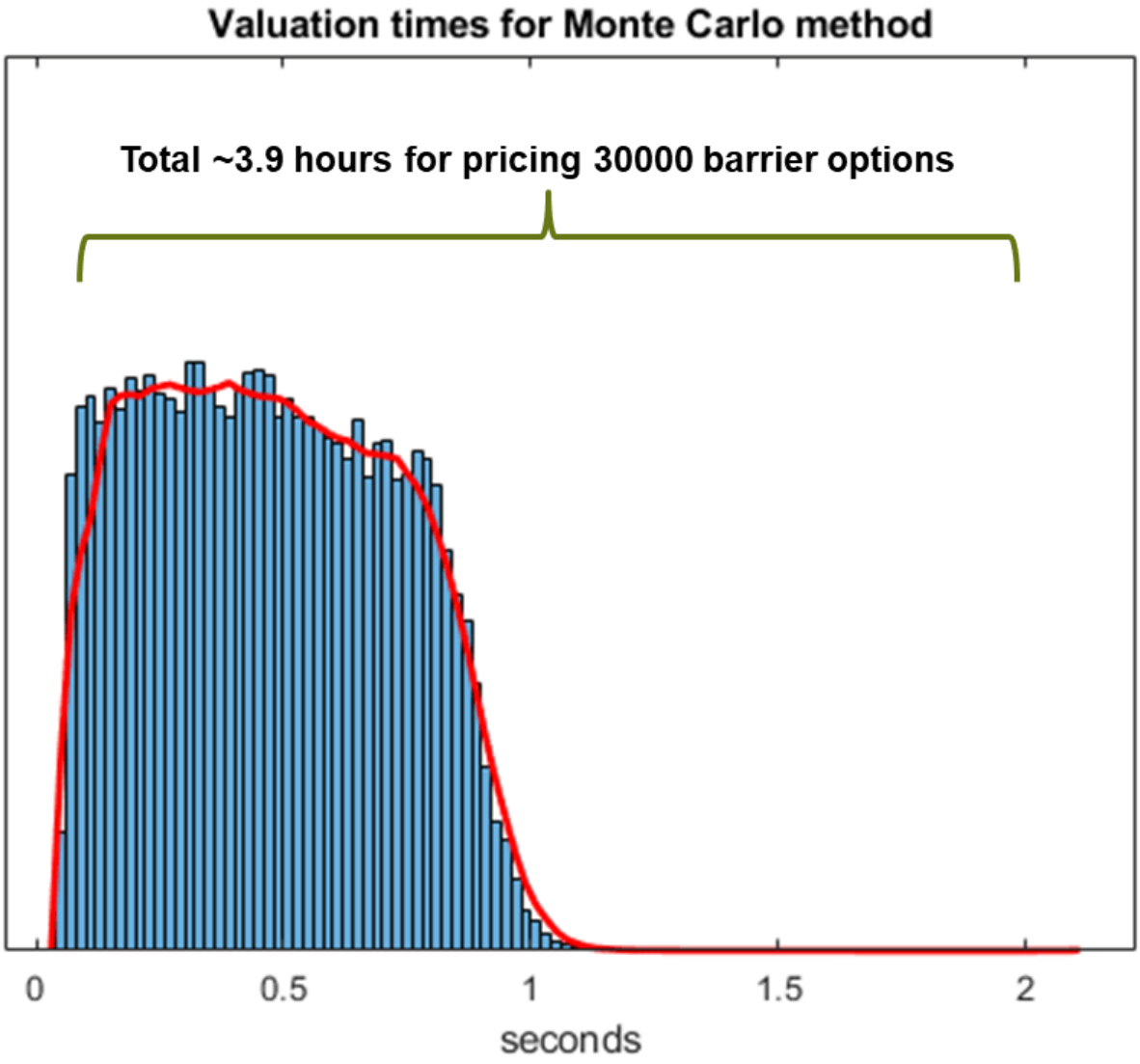
Calculate Barrier Option Prices Using Monte Carlo Simulation

After you create the parameter space, calculate the prices of the `Barrier` option by Monte Carlo simulation using the object-based pricing framework in Financial Instrument Toolbox™. Specifically, use `ratecurve`, `Heston`, `Barrier`, and `AssetMonteCarlo` to create the objects required to price the `Barrier` option. To avoid waiting for the Monte Carlo simulation, load the calculated prices for the example by setting the `doMCPricing` flag to `false`.

```
doMCPricing = false;

if doMCPricing
    % Calculate the prices using the AssetMonteCarlo pricer.
    Settle = datetime(2021,2,1);
    Price = nan(size(inputs,1),1);

    for i = 1:size(inputs,1)
        AssetPrice = inputs(i,iMoneyness);
        Strike = 1;
        Barrier = inputs(i,iBarrier);

        V0 = inputs(i,iV0);
        ThetaV = inputs(i,iTheta);
        Kappa = inputs(i,iKappa);
        SigmaV = inputs(i,iSigma);
        RhoSV = inputs(i,iCorr);

        Rates = inputs(i,iRate);
        ExerciseDate = daysadd(Settle,round(inputs(i,iTime)*365),0);
        ZeroCurve = ratecurve('zero',Settle,ExerciseDate,Rates);

        hestonModel = finmodel("Heston",V0=V0,ThetaV=ThetaV,Kappa=Kappa,SigmaV=SigmaV,RhoSV=RhoS

        MCPricer = finpricer("AssetMonteCarlo",DiscountCurve=ZeroCurve,Model=hestonModel,SpotPri
            SimulationDates=[Settle:days(2):ExerciseDate, ExerciseDate],numTrials=nTrials);

        CallBarrier = fininstrument("Barrier",ExerciseDate=ExerciseDate,Strike=Strike,OptionType=
            Barriertype=barriertype,Barriervalue=Barrier,ExerciseStyle=exerciseStyle);

        Price(i) = price(MCPricer,CallBarrier);

    end

else
    % Load the calculated prices for the example.
    load('DeepLearningBarrierHeston.mat','inputs','Price')
end
```

If you do not use the calculated prices for the example by setting the `doMCPricing` flag to `false`, the following histogram shows the distribution of valuation times for *each individual* barrier option using a Monte Carlo method. This histogram demonstrates that it takes approximately 0.46 seconds to compute a single barrier option price. In this example, `nTrials` is set to `1e3` which is the number of trials in the Monte Carlo simulation to compute a single option price. The total time to compute

prices for the sample size of 30,000 barrier options is approximately 3.9 hours, depending on the processor speed.



Define Neural Network

Different network architectures can help with the task of pricing barrier options using a Heston model. Choosing a neural network architecture requires balancing computation time against accuracy. This example uses multiple, fully-connected layers and Leaky ReLU activations.

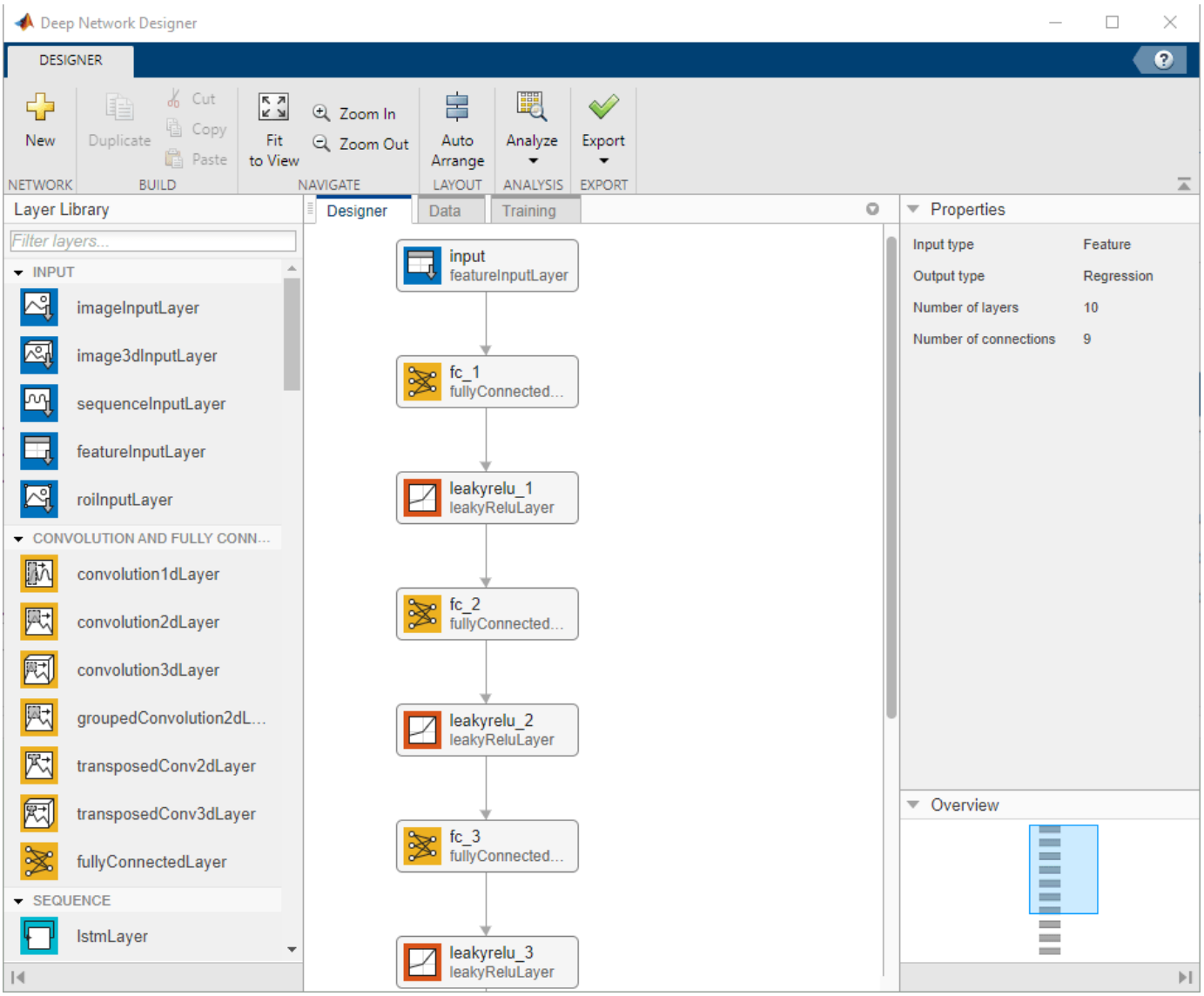
```
numFeatures = size(inputs,2);
layers = [
    featureInputLayer(numFeatures,Normalization='zscore')
    fullyConnectedLayer(32,WeightsInitializer='he')
    leakyReluLayer
    fullyConnectedLayer(32,WeightsInitializer='he')
    leakyReluLayer
    fullyConnectedLayer(32,WeightsInitializer='he')
    leakyReluLayer
```

```
    fullyConnectedLayer(1,WeightsInitializer='he')
    leakyReluLayer
    regressionLayer];
```

Visualize Network

You can visualize the network using the Deep Network Designer (Deep Learning Toolbox) app or the `analyzeNetwork` (Deep Learning Toolbox) function.

```
deepNetworkDesigner(layerGraph(layers))
```



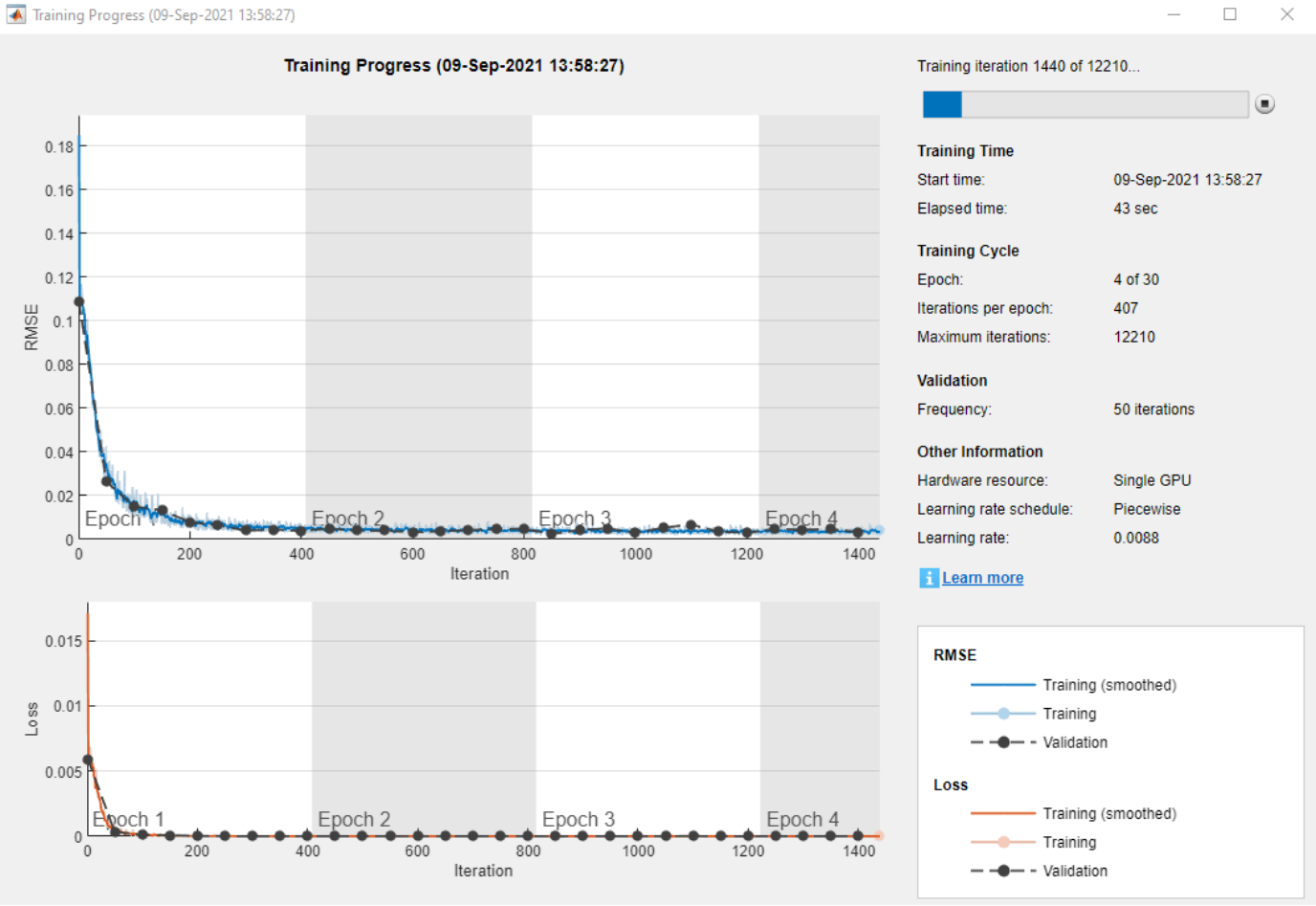
Train Network

Train the neural network by using the `trainNetwork` (Deep Learning Toolbox) function. The function creates a hold-out set to test the trained network and allocates a validation set to monitor the overfitting during the training. By default, `trainNetwork` uses a GPU if one is available; otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported

GPU device. For information, see “Deep Learning with MATLAB on Multiple GPUs” (Deep Learning Toolbox).

```
n = size(Price,1);
c = cvpartition(n,Holdout=1/5);     % Hold out 1/5 of the data set for testing
XTrain = inputs(training(c),:);     % 4/5 of the input for training
YTrain = Price(training(c),:);      % 4/5 of the target for training
XTest = inputs(test(c),:);          % 1/5 of the input for testing
YTest = Price(test(c),:);           % 1/5 of the target for testing

nTrain = size(XTrain,1);
idx = randperm(nTrain,floor(nTrain*0.1)); % 10% validation data
XValidation = XTrain(idx,:);
XTrain(idx,:) = [];
YValidation = YTrain(idx,:);
YTrain(idx,:) = [];

opts = trainingOptions('adam', ...
    MaxEpochs=30, ...
    Shuffle='every-epoch', ...
    Plots='none', ...
    Verbose=false, ...
    VerboseFrequency=50, ...
    MiniBatchSize=265, ...
    ValidationData={XValidation,YValidation}, ...
    ValidationFrequency=50, ...
    ValidationPatience=Inf, ...
    L2Regularization=1.9e-7, ...
    InitialLearnRate=8.8e-3, ...
    LearnRateSchedule='piecewise', ...
    LearnRateDropPeriod=4, ...
    LearnRateDropFactor=0.128, ...
    SquaredGradientDecayFactor=0.55, ...
    GradientDecayFactor=0.62);
%opts.ExecutionEnvironment = "gpu";  % When using GPU

doTraining = false;
if doTraining
    % Train the network.
    net = trainNetwork(XTrain,YTrain,layers,opts);
else
    % Load the pretrained network for the example.
    load('DeepLearningBarrierHeston.mat','net')
end
```

To avoid waiting for the training, load the pretrained network by setting the `doTraining` flag to `false`. To train the networks using `analyzeNetwork` (Deep Learning Toolbox), set the `doTraining` flag to `true`. The **Training Progress** window displays progress when `Plots` in `trainingOptions` is set as `training-progress`.



Test Network

After training the network model, you can use the `predict` (Deep Learning Toolbox) function to evaluate the test data set containing 30,000 barrier options on this trained network. Compared to the histogram in Calculate Barrier Option Prices Using MonteCarlo Simulation on page 3-150 where Monte Carlo simulation takes 0.46 seconds to price each barrier option (3.9 hours to price 30,000 barrier options), after the network model is trained, a data set containing 30,000 barrier option is evaluated in seconds.

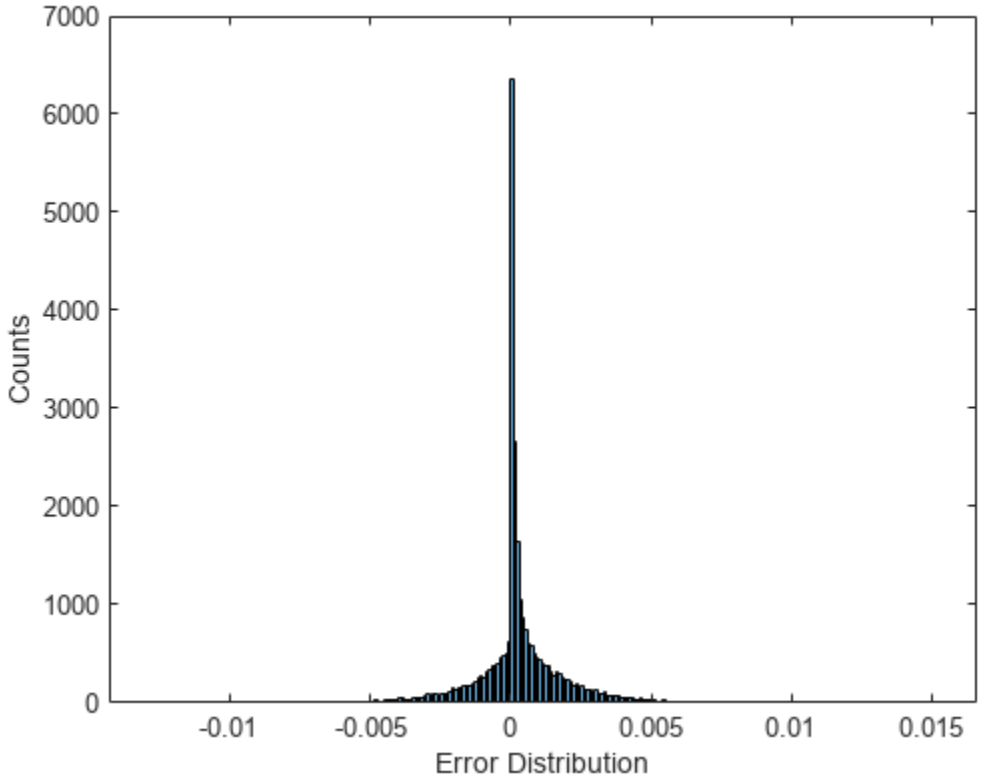
```
% If testing on a GPU, then convert data to a gpuArray.
if opts.ExecutionEnvironment == "gpu" && canUseGPU
    XTest = gpuArray(XTest);
end

YPred = predict(net,XTest);
```

To assess the performance of the network, calculate the mean-squared error (MSE) value.

```
mseTest = mean((YTest - YPred).^2)

mseTest = single
    2.9402e-06
```

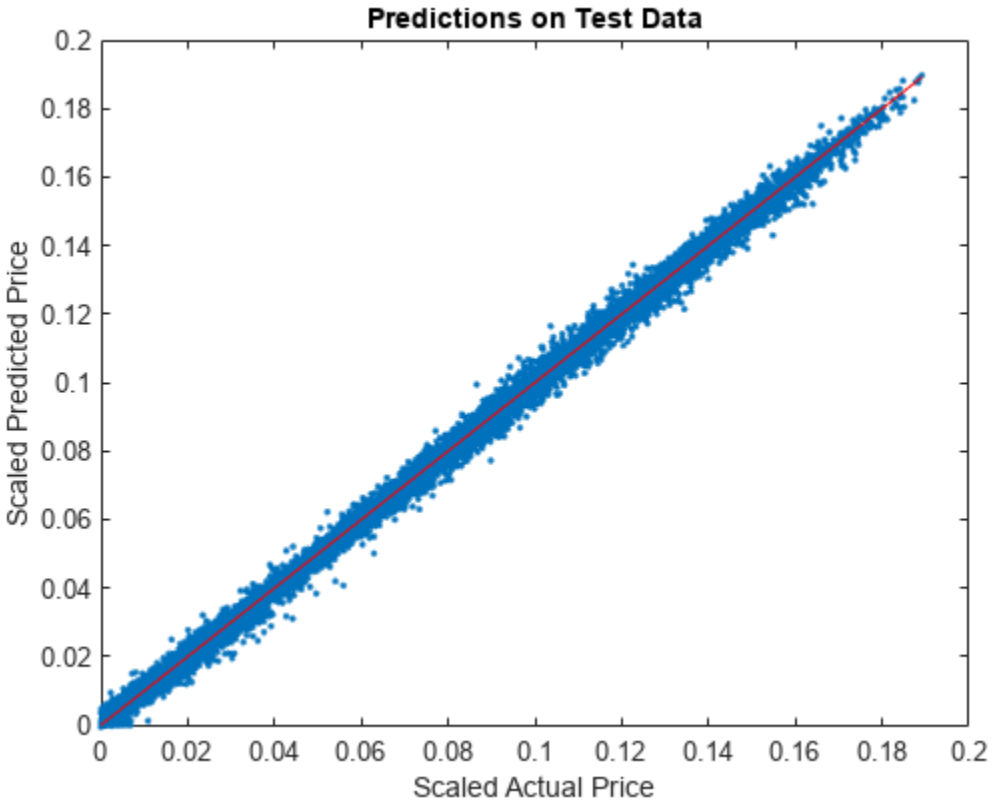
The following histogram shows the error distribution for the predicted barrier option price using Deep Learning Toolbox™ with respect to the calculated barrier option price using Financial Instrument Toolbox™.

```
figure
histogram(YTest - YPred, Binwidth=1e-4)
xlabel('Error Distribution')
ylabel('Counts')
```



A plot of the calculated prices and predicted prices shows the performance of the network for the Heston model using the test data.

```
figure
plot(YTest,YPred,'.',[min(YTest),max(YTest)],[min(YTest),max(YTest)],'r')
xlabel('Scaled Actual Price')
ylabel('Scaled Predicted Price')
title('Predictions on Test Data')
```



References

[1] Goodfellow, I., Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[2] Niederreiter, H. "Random Number Generation and Quasi-Monte Carlo Methods." *Society for Industrial and Applied Mathematics*, 1992.

See Also

`Barrier` | `Heston` | `AssetMonteCarlo` | **Deep Network Designer** | `trainNetwork` | `trainingOptions`

Related Examples

- “Hedge Options Using Reinforcement Learning Toolbox™” (Deep Learning Toolbox)

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.

See Also

`hedgeopt` | `hedgeslf`

Related Examples

- “Portfolio Creation Using Functions” on page 1-6
- “Adding Instruments to an Existing Portfolio Using Functions” on page 1-8
- “Instrument Constructors” on page 1-15
- “Creating Instruments or Properties” on page 1-16
- “Searching or Subsetting a Portfolio” on page 1-17
- “Hedging Functions” on page 4-3
- “Hedging with hedgeopt” on page 4-4
- “Self-Financing Hedges with hedgeslf” on page 4-9
- “Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
- “Pricing and Hedging a Portfolio Using the Black-Karasinski Model” on page 4-13
- “Specifying Constraints with ConSet” on page 4-24
- “Portfolio Rebalancing” on page 4-26
- “Hedging with Constrained Portfolios” on page 4-28

More About

- “Instrument Constructors” on page 1-15

Hedging Functions

In this section...

Introduction

Hedging is an investment to reduce the risk of adverse price movements in an asset. Financial Instruments Toolbox offers two functions for assessing the fundamental hedging tradeoff, `hedgeopt` and `hedgeslf`.

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

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

`hedgeopt` allows investors to modify portfolio allocations among instruments according to either of the goals. The problem is cast as a constrained linear least-squares problem. For additional information about `hedgeopt`, see “Hedging with hedgeopt” on page 4-4.

The second function, `hedgeslf`, attempts to allocate a self-financing hedge among a portfolio of instruments. In particular, `hedgeslf` attempts to maintain a constant portfolio value consistent with reduced portfolio sensitivities (that is, the rebalanced portfolio is hedged against market moves and is closest to being self-financing). If `hedgeslf` cannot find a self-financing hedge, it rebalances the portfolio to minimize overall portfolio sensitivities. For additional information on `hedgeslf`, see “Self-Financing Hedges with hedgeslf” on page 4-9.

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. So, 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`, `ittsens`, or `sttsens` 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 2-63. Likewise, for examples showing the computation of sensitivities for equity exotic options, see “Computing Equity Instrument Sensitivities” on page 3-75.

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.

```
FixedInd = [1 2 3 4 5 6 7 8];
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,Holdings, FixedInd, [], [], TargetSens)
```

```
Holdings =

        100.00
         50.00
        -50.00
         80.00
          8.00
         30.00
         40.00
         10.00


Sens =

     -61910.22     788946.21       4852.91


Cost =

             0


Quantity =

  Columns 1 through 6

       100.00         50.00        -50.00         80.00          8.00         30.00

  Columns 7 through 8

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

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

Then, specify the instruments to be fixed.

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

Finally, call `hedgeopt`

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

and again examine the results.

```
Sens =

     -23000.00      -3300.00       3000.00

Cost =

      19174.02

Quantity' =

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

Recompute `Value1`, the portfolio value after rebalancing.

```
Value1 = Quantity * Price

Value1 =

      4500.60
```

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

Fully Hedged Portfolio

The example in “Partially Hedged Portfolio” on page 4-6 illustrates a partial hedge, but perhaps the most interesting case involves the cost associated with a fully hedged portfolio (simultaneous delta,

gamma, and vega neutrality). In this case, set the target sensitivity to a row vector of 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 4-6 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.

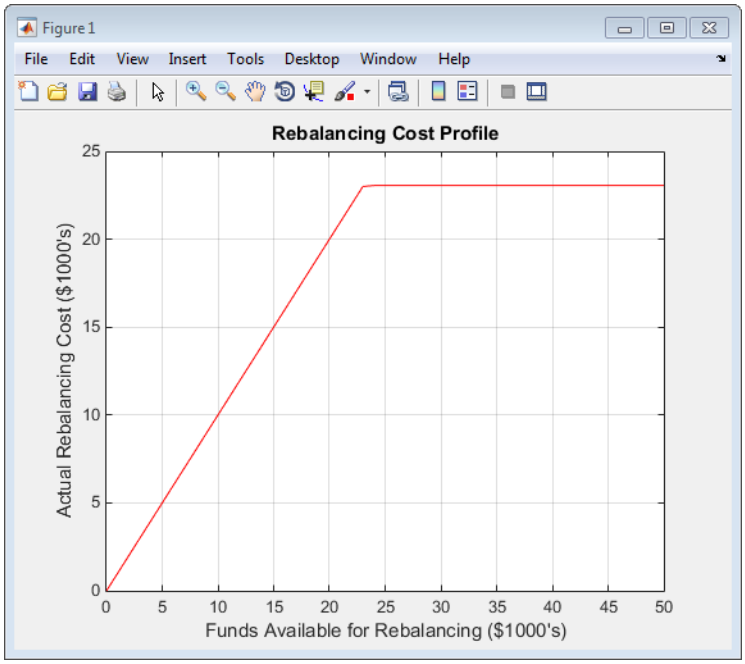
```
MaxCost = [0:1000:50000];
```

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)

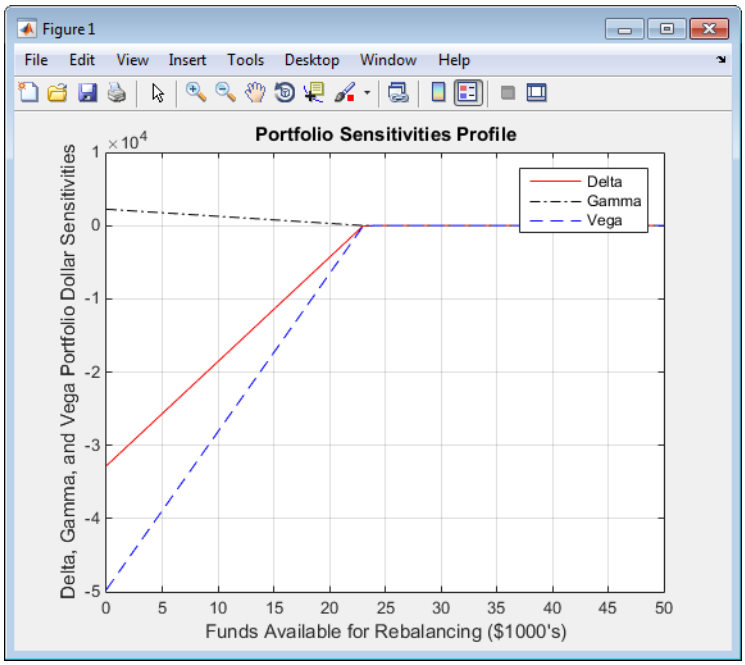
```
plot(MaxCost/1000, Cost/1000, 'red'), grid
xlabel('Funds Available for Rebalancing ($1000''s)')
ylabel('Actual Rebalancing Cost ($1000''s)')
title ('Rebalancing Cost Profile')
```



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 4-8 and “Funds Available for Rebalancing” on page 4-9 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
```

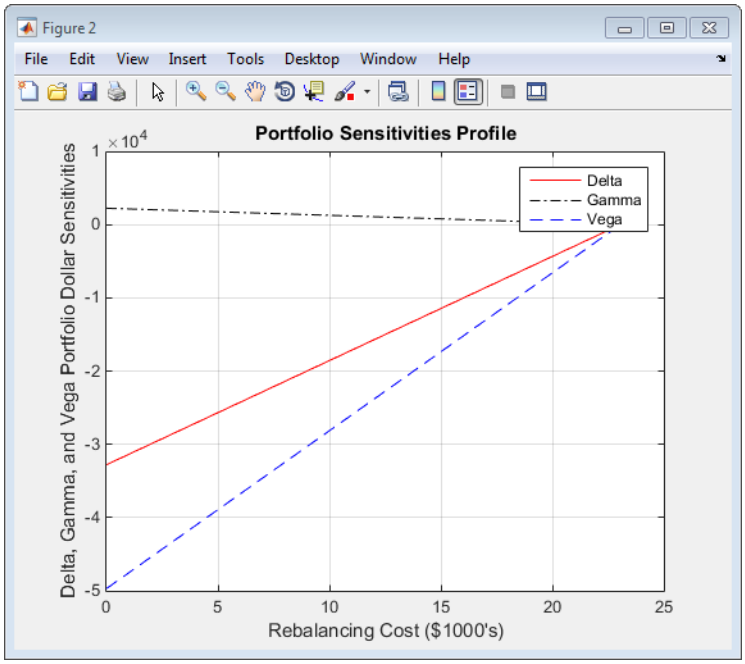
In this example, the portfolio is clearly not self-financing, so `hedgeslf` finds the best possible solution required to obtain zero sensitivities.

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.

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 4-28 for additional information.

See Also

`hedgeopt` | `hedgeslf`

Related Examples

- “Portfolio Creation Using Functions” on page 1-6
- “Adding Instruments to an Existing Portfolio Using Functions” on page 1-8
- “Instrument Constructors” on page 1-15
- “Creating Instruments or Properties” on page 1-16
- “Searching or Subsetting a Portfolio” on page 1-17
- “Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
- “Pricing and Hedging a Portfolio Using the Black-Karasinski Model” on page 4-13
- “Specifying Constraints with ConSet” on page 4-24
- “Portfolio Rebalancing” on page 4-26
- “Hedging with Constrained Portfolios” on page 4-28

More About

- “Instrument Constructors” on page 1-15
- “Hedging” on page 4-2

Pricing and Hedging a Portfolio Using the Black-Karasinski Model

This example illustrates how MATLAB® can be used to create a portfolio of interest-rate derivatives securities, and price it using the Black-Karasinski interest-rate model. The example also shows some hedging strategies to minimize exposure to market movements.

Create the Interest-Rate Term Structure Based on Reported Data

The structure `RateSpec` is an interest-rate term structure that defines the initial rate specification from which the tree rates are derived. Use the information of annualized zero coupon rates in the table below to populate the `RateSpec` structure.

```
  From             To           Rate
27 Feb 2007    27 Feb 2008      0.0493
27 Feb 2007    27 Feb 2009      0.0459
27 Feb 2007    27 Feb 2010      0.0450
27 Feb 2007    27 Feb 2012      0.0446
27 Feb 2007    27 Feb 2014      0.0445
27 Feb 2007    27 Feb 2017      0.0450
27 Feb 2007    27 Feb 2027      0.0473
```

This data could be retrieved from the Federal Reserve Statistical Release page by using the Datafeed Toolbox™. In this case, the Datafeed Toolbox™ will connect to FRED® and pull back the rates of the following treasury notes.

```
  Terms    Symbol
 =======   ======
    1   =  DGS1
    2   =  DGS2
    3   =  DGS3
    5   =  DGS5
    7   =  DGS7
    10  =  DGS10
    20  =  DGS20
```

Create the connection object:

```
  c = fred;
```

Create the symbol fetch list:

```
FredNames   = { ...
  'DGS1'; ...      % 1  Year
  'DGS2'; ...      % 2  Year
  'DGS3'; ...      % 3  Year
  'DGS5'; ...      % 5  Year
  'DGS7'; ...      % 7  Year
  'DGS10'; ...     % 10 Year
  'DGS20'};        % 20 Year
```

Define the Terms:

```
Terms = [ 1; ...      % 1  Year
          2; ...      % 2  Year
```

```
          3; ...      % 3  Year
          5; ...      % 5  Year
          7; ...      % 7  Year
         10; ...      % 10 Year
         20];         % 20 Year
```

Set the `StartDate` to Feb 27, 2007:

```
StartDate = datenum('Feb-27-2007');

FredRet = fetch(c,FredNames,StartDate);
```

Set the `ValuationDate` based on the `StartDate`:

```
ValuationDate = StartDate;

EndDates = [];

Rates =[];
```

Create the `EndDates`:

```
for idx = 1:length(FredRet)

 %Pull the rates associated with Feb 27, 2007. All the Fred Rates come
 %back as percents
 Rates = [Rates; ...
     FredRet(idx).Data(1,2) / 100];

  %Determine the EndDates by adding the Term to the year of the
  %StartDate
  EndDates = [EndDates; ...
     round(datenum(...
         year(StartDate)+ Terms(idx,1), ...
         month(StartDate),...
         day(StartDate)))];

end
```

Use the function `intenvset` to create the `RateSpec` with the following data:

```
Compounding = 1;
StartDate = datetime(2007,2,27);
Rates = [0.0493; 0.0459; 0.0450; 0.0446; 0.0446; 0.0450; 0.0473];
EndDates = [datetime(2008,2,27); datetime(2009,2,27) ; datetime(2010,2,27) ; datetime(2012,2,27)
ValuationDate = StartDate;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [7x1 double]
            Rates: [7x1 double]
         EndTimes: [7x1 double]
       StartTimes: [7x1 double]
         EndDates: [7x1 double]
       StartDates: 733100
    ValuationDate: 733100
```

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

Specify the Volatility Model

Create the structure `VolSpec` that specifies the volatility process with the following data.

```
Volatility = [0.011892; 0.01563; 0.02021; 0.02125; 0.02165; 0.02065; 0.01803];
Alpha = [0.0001];
VolSpec = bkvolspec(ValuationDate, EndDates, Volatility, EndDates(end), Alpha)
```

```
VolSpec = struct with fields:
             FinObj: 'BKVolSpec'
      ValuationDate: 733100
           VolDates: [7x1 double]
           VolCurve: [7x1 double]
         AlphaCurve: 1.0000e-04
         AlphaDates: 740405
    VolInterpMethod: 'linear'
```

Specify the Time Structure of the Tree

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 = bktimespec(ValuationDate, EndDates)
```

```
TimeSpec = struct with fields:
           FinObj: 'BKTimeSpec'
    ValuationDate: 733100
         Maturity: [7x1 double]
      Compounding: -1
            Basis: 0
     EndMonthRule: 1
```

Create the BK Tree

Use the previously computed values for `RateSpec`, `VolSpec`, and `TimeSpec` to create the BK tree.

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

```
BKTree = struct with fields:
      FinObj: 'BKFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 5 7 10]
        dObs: [733100 733465 733831 734196 734926 735657 736753]
      CFlowT: {1x7 cell}
       Probs: {1x6 cell}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]  [2 3 3 4 5 5 6]  [2 3 4 5 6 7 8]  [2 2 ... ]}
     FwdTree: {1x7 cell}
```

Visualize the interest rate evolution along the tree by looking at the output structure `BKTree`. The function `bktree` returns an inverse discount tree, which you can convert into an interest rate tree with the `cvtree` function.

```
BKTreeR = cvtree(BKTree)
```

```
BKTreeR = struct with fields:
      FinObj: 'BKRateTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 5 7 10]
        dObs: [733100 733465 733831 734196 734926 735657 736753]
      CFlowT: {1x7 cell}
       Probs: {1x6 cell}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]  [2 3 3 4 5 5 6]  [2 3 4 5 6 7 8]  [2 2 ... ]}
    RateTree: {1x7 cell}
```

Look at the upper, middle and lower branch paths of the tree.

```
OldFormat = get(0, 'format');
format short

%Rate at root node:
RateRoot      = trintreepath(BKTreeR, 0)
```

```
RateRoot = 0.0481
```

```
%Rates along upper branch:
RatePathUp    = trintreepath(BKTreeR, [1 1 1 1 1 1])
```

```
RatePathUp = 7×1

    0.0481
    0.0425
    0.0446
    0.0478
    0.0510
    0.0555
    0.0620
```

```
%Rates along middle branch:
RatePathMiddle = trintreepath(BKTreeR, [2 2 2 2 2 2])
```

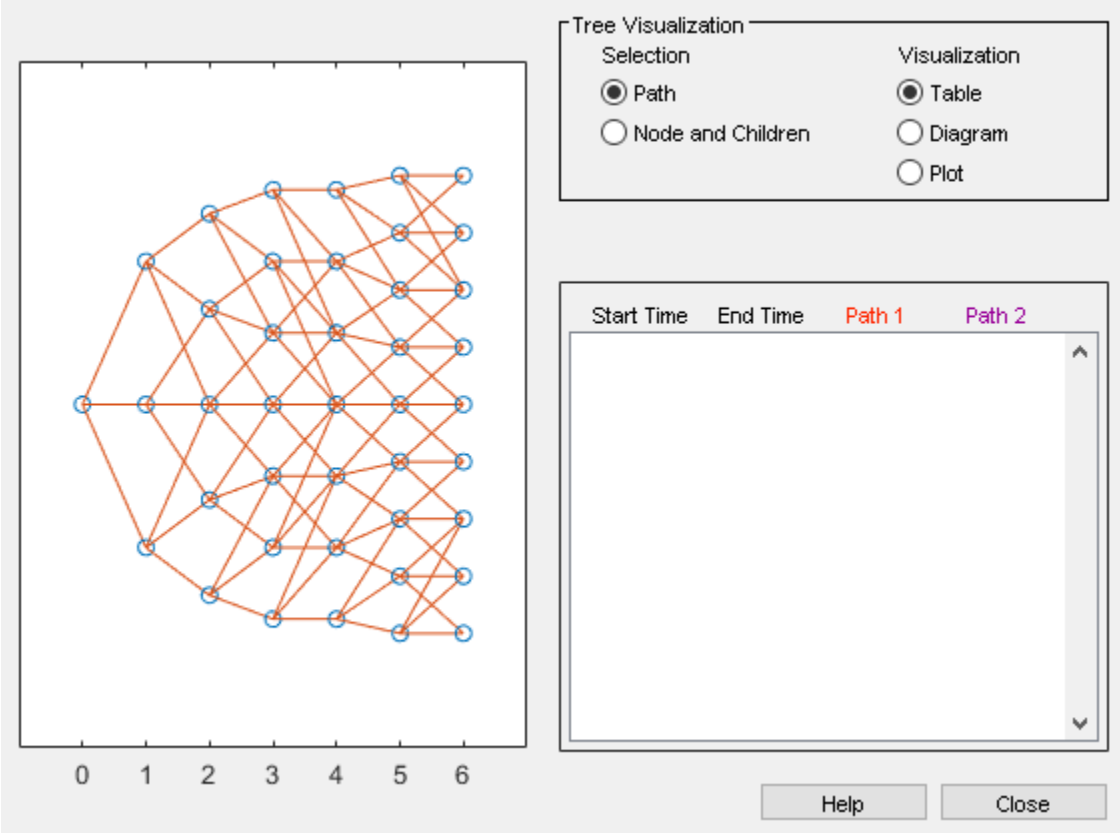
```
RatePathMiddle = 7×1

    0.0481
    0.0416
    0.0423
    0.0430
    0.0436
    0.0449
    0.0484
```

```
%Rates along lower branch:
RatePathDown = trintreepath(BKTreeR, [3 3 3 3 3 3])
```

```
RatePathDown = 7×1

    0.0481
    0.0408
    0.0401
    0.0388
    0.0373
    0.0363
    0.0378
```

You can also display a graphical representation of the tree to examine interactively the rates on the nodes of the tree until maturity. The function `treeviewer` displays the structure of the rate tree in the left window. The tree visualization in the right window is blank, but by selecting **Table/Diagram** and clicking on the nodes you can examine the rates along the paths.

```
treeviewer(BKTreeR);
```



Create an Instrument Portfolio

Create a portfolio consisting of two bonds instruments and an option on the 5% bond.

```
% Two Bonds
CouponRate = [0.04;0.05];
```

```
Settle = datetime(2007,2,27);
Maturity = [datetime(2009,2,27) ; datetime(2010,2,27) ];
Period = 1;

% American Option on the 5% Bond
OptSpec = {'call'};
Strike = 98;
ExerciseDates = datetime(2010,2,27);
AmericanOpt = 1;

InstSet = instadd('Bond', CouponRate, Settle,  Maturity, Period);
InstSet = instadd(InstSet,'OptBond', 2, OptSpec, Strike, ExerciseDates, AmericanOpt);

% Assign Names and Holdings
Holdings = [10; 15;3];
Names = {'4% Bond'; '5% Bond'; 'Option 98'};

InstSet = instsetfield(InstSet, 'Index',1:3, 'FieldName', {'Quantity'}, 'Data', Holdings );
InstSet = instsetfield(InstSet, 'Index',1:3, 'FieldName', {'Name'}, 'Data', Names );
```

Examine the set of instruments contained in the variable `InstSet`.

```
instdisp(InstSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.04       27-Feb-2007    27-Feb-2009    1      0     1            NaN       NaN
2     Bond 0.05       27-Feb-2007    27-Feb-2010    1      0     1            NaN       NaN

Index Type    UnderInd OptSpec Strike ExerciseDates  AmericanOpt Quantity Name
3     OptBond 2        call    98     27-Feb-2010    1           3        Option 98
```

Price the Portfolio Using the BK Model

Calculate the price of each instrument in the portfolio.

```
[Price, PTree] = bkprice(BKTree, InstSet)
```

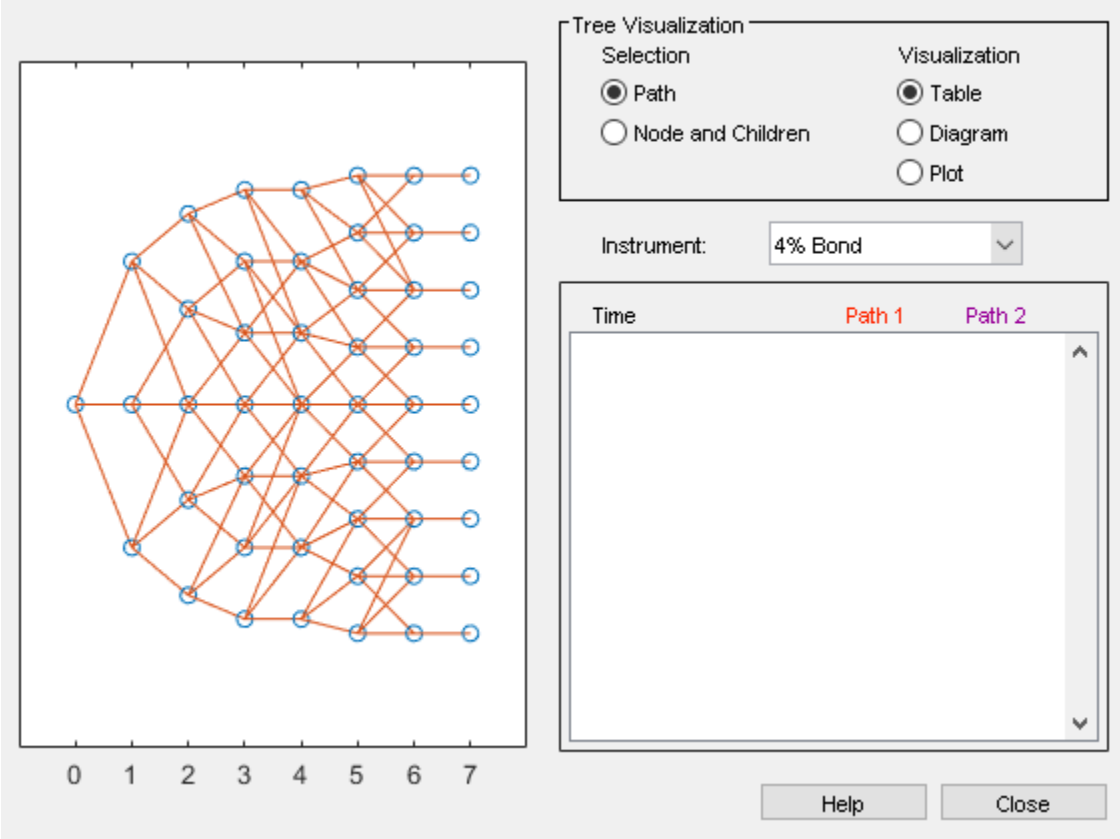
```
Price = 3×1

   98.8841
  101.3470
    3.3470


PTree = struct with fields:
     FinObj: 'BKPriceTree'
      PTree: {1x8 cell}
     AITree: {1x8 cell}
     ExTree: {1x8 cell}
       tObs: [0 1 2 3 5 7 10 20]
    Connect: {[2]  [2 3 4]  [2 3 4 5 6]  [2 3 3 4 5 5 6]  [2 3 4 5 6 7 8]  [2 2 3 ... ]}
      Probs: {1x6 cell}
```

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.

In the `Price` vector, the first element, 98.884, represents the price of the first instrument (4% Bond); the second element, 101.347, represents the price of the second instrument (5% Bond), and 3.347 represents the price of the American call option.

You can also display a graphical representation of the price tree to examine the prices on the nodes of the tree until maturity.

```
treeviewer(PTree,InstSet);
```



Add More Instruments to the Existing Portfolio

Add instruments to the existing portfolio: cap, floor, floating rate note, vanilla swap and a puttable and callable bond.

```
% Cap
StrikeC =0.035;
InstSet = instadd(InstSet,'Cap', StrikeC, Settle, datetime(2010,2,27));

% Floor
StrikeF =0.05;
InstSet = instadd(InstSet,'Floor', StrikeF, Settle, datetime(2009,2,27));

% Floating Rate Note
InstSet = instadd(InstSet,'Float', 30, Settle, datetime(2009,2,27));

% Vanilla Swap
 LegRate =[0.04 5];
```

```
 InstSet = instadd(InstSet,'Swap', LegRate, Settle, datetime(2010,2,27));

% Puttable and Callable Bonds
InstSet = instadd(InstSet,'OptEmBond', CouponRate, Settle,datetime(2010,2,27), {'put';'call'},..
                  Strike, datetime(2010,2,27),'AmericanOpt', 1, 'Period', 1);

% Process Names and Holdings
Holdings = [15 ;5 ;8; 7; 9; 4];
Names = {'3.5% Cap';'5% Floor';'30BP Float';'4%/5BP Swap'; 'PuttBond'; 'CallBond' };

InstSet = instsetfield(InstSet, 'Index',4:9, 'FieldName', {'Quantity'}, 'Data', Holdings );
InstSet = instsetfield(InstSet, 'Index',4:9, 'FieldName', {'Name'}, 'Data', Names );
```

Examine the set of instruments contained in the variable `InstSet`.

```
instdisp(InstSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.04       27-Feb-2007    27-Feb-2009    1      0     1            NaN       NaN
2     Bond 0.05       27-Feb-2007    27-Feb-2010    1      0     1            NaN       NaN

Index Type    UnderInd OptSpec Strike ExerciseDates  AmericanOpt Quantity Name
3     OptBond 2        call    98     27-Feb-2010    1           3        Option 98

Index Type Strike Settle         Maturity       CapReset Basis Principal Quantity Name
4     Cap  0.035  27-Feb-2007    27-Feb-2010    1        0     100       15       3.5% Cap

Index Type  Strike Settle         Maturity       FloorReset Basis Principal Quantity Name
5     Floor 0.05   27-Feb-2007    27-Feb-2009    1          0     100       5        5% Floor

Index Type  Spread Settle         Maturity       FloatReset Basis Principal EndMonthRule CapRate
6     Float 30     27-Feb-2007    27-Feb-2009    1          0     100       1            Inf

Index Type LegRate   Settle         Maturity       LegReset Basis Principal LegType EndMonthRule
7     Swap [0.04  5] 27-Feb-2007    27-Feb-2010    [NaN]    0     100       [NaN]   1

Index Type      CouponRate Settle         Maturity       OptSpec Strike ExerciseDates
8     OptEmBond 0.04       27-Feb-2007    27-Feb-2010    put     98     27-Feb-2007   27-Feb-201
9     OptEmBond 0.05       27-Feb-2007    27-Feb-2010    call    98     27-Feb-2007   27-Feb-201
```

Hedging

The idea behind hedging is to minimize exposure to market movements. As the underlying changes, the proportions of the instruments forming the portfolio may need to be adjusted to keep the sensitivities within the desired range.

Calculate sensitivities using the BK model.

```
[Delta, Gamma, Vega, Price] = bksens(BKTree, InstSet);
```

Get the current portfolio holdings.

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

Create 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.

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

```
         98.88         10.00       -185.47        528.47             0
        101.35         15.00       -277.51       1045.05             0
          3.35          3.00       -223.52      11843.32             0
          2.77         15.00        250.04       2921.11         -0.00
          0.75          5.00       -132.97      11566.69             0
        100.56          8.00         -0.80          2.02             0
         -1.53          7.00       -272.08       1027.85          0.00
         98.60          9.00       -168.92      21712.82             0
         98.00          4.00        -53.99     -10798.27             0
```

The first column above is the dollar unit price of each instrument, the second column is the number of contracts of each instrument, and the third, fourth, and fifth columns are the dollar delta, gamma, and vega sensitivities.

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

```
TargetSens  = Holdings' * Sensitivities
```

```
TargetSens = 1×3

      -7249.21     317573.92         -0.00
```

Obtain a Neutral Sensitivity Portfolio Using `hedgeslf`

Suppose you want to obtain a delta, gamma and vega neutral portfolio. The function `hedgeslf` finds the reallocation in a portfolio of financial instruments closest to being self-financing (maintaining constant portfolio value).

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

```
Sens = 3×1

         -0.00
         -0.00
         -0.00


Value1 =
       4637.54


Quantity = 9×1

         10.00
          5.26
         -5.11
          7.06
         -3.05
         12.45
         -7.36
          8.47
```

```
         10.37
```

The function `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, you can verify the cost by comparing the portfolio values.

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

```
Value0 =
       4637.54
```

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.

Adding Constraints to Hedge a Portfolio

Suppose that you want to place upper and lower bounds on the individual instruments in the portfolio. Let's say that you want to bound the position of all instruments to within +/- 11 contracts.

Applying these constraints disallows the current positions in the fifth and eighth instruments. All other instruments are currently within the upper/lower bounds.

```
% Specify the lower and upper bounds
LowerBounds = [-11  -11  -11  -11  -11  -11  -11  -11  -11];
UpperBounds = [ 11   11   11   11   11   11   11   11   11];

% Use the function portcons to build the constraints
ConSet = portcons('AssetLims', LowerBounds, UpperBounds);

% Apply the constraints to the portfolio
[Sens, Value, Quantity1] = hedgeslf(Sensitivities, Price, Holdings, [], ConSet)
```

```
Sens = 3×1

             0
             0
             0


Value =
             0


Quantity1 = 9×1

             0
             0
             0
             0
             0
             0
             0
             0
             0
```

Observe that the `hedgeslf` function enforces the bounds on the fifth and eighth instruments, and the portfolio continues to be fully hedged and self-financing.

```
set(0, 'format', OldFormat);
```

See Also

`hedgeopt` | `hedgeslf`

Related Examples

- “Portfolio Creation Using Functions” on page 1-6
- “Adding Instruments to an Existing Portfolio Using Functions” on page 1-8
- “Instrument Constructors” on page 1-15
- “Creating Instruments or Properties” on page 1-16
- “Searching or Subsetting a Portfolio” on page 1-17
- “Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
- “Specifying Constraints with ConSet” on page 4-24
- “Portfolio Rebalancing” on page 4-26
- “Hedging with Constrained Portfolios” on page 4-28

More About

- “Instrument Constructors” on page 1-15
- “Hedging” on page 4-2

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”.

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 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, start up `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 4-11.

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.

See Also

`hedgeopt` | `hedgeslf`

Related Examples

- “Portfolio Creation Using Functions” on page 1-6
- “Adding Instruments to an Existing Portfolio Using Functions” on page 1-8
- “Instrument Constructors” on page 1-15
- “Creating Instruments or Properties” on page 1-16
- “Searching or Subsetting a Portfolio” on page 1-17
- “Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
- “Pricing and Hedging a Portfolio Using the Black-Karasinski Model” on page 4-13
- “Hedging with Constrained Portfolios” on page 4-28

More About

- “Instrument Constructors” on page 1-15
- “Hedging” on page 4-2

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` for details.) In particular, `lsqlin` attempts to minimize the constrained linear least squares problem

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

where C, A, and Aeq are matrices, and d, b, beq, lb, and ub are vectors. For Financial Instruments 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 4-6 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 that 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 4-8, “Funds Available for Rebalancing” on page 4-9, and “Rebalancing Cost” on page 4-11.

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 function `portcons`, you can specify these constraints, along with various 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.

See Also

`hedgeopt` | `hedgeslf`

Related Examples

- “Portfolio Creation Using Functions” on page 1-6
- “Adding Instruments to an Existing Portfolio Using Functions” on page 1-8
- “Instrument Constructors” on page 1-15
- “Creating Instruments or Properties” on page 1-16
- “Searching or Subsetting a Portfolio” on page 1-17
- “Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
- “Pricing and Hedging a Portfolio Using the Black-Karasinski Model” on page 4-13
- “Specifying Constraints with ConSet” on page 4-24
- “Portfolio Rebalancing” on page 4-26

More About

- “Instrument Constructors” on page 1-15
- “Hedging” on page 4-2

Hedging Strategies Using Spread Options

This example shows different hedging strategies to minimize exposure in the Energy market using Crack Spread Options.

Understanding Crack Spread Options

In the petroleum industry, refiners are concerned about the difference between their input costs (crude oil) and output prices (refined products - gasoline, heating oil, diesel fuel, and so on). The differential between these two underlying commodities is referred to as a *Crack Spread*. It represents the profit margin between crude oil and the refined products.

A *Spread option* is an option on the spread where the holder has the right, but not the obligation, to enter into a spot or forward spread contract. Crack Spread Options are often used to protect against declines in the crack spread or to monetize volatility or price expectations on the spread.

Example 1: Protecting Margins using a 1:1 Crack Spread Option

A marketer is interested in protecting his gasoline margin since current prices are strong. A crack spread option strategy is used to maintain profits for the following season. In March the June WTI crude oil futures are at $91.10 per barrel and RBOB gasoline futures contract are at $2.72 per gallon. The marketer's strategy is a long crack call involving purchasing RBOB gasoline futures and selling crude oil futures.

```
OldFormat = get(0, 'format');
format bank

% Price and volatility of RBOB gasoline
Price1gallon = 2.72;          % $/gallon
Price1 = Price1gallon * 42;   % $/barrel
Vol1 = 0.39;

% Price and volatility of WTI crude oil
Price2 = 91.10;         % $/barrel
Vol2 = 0.34;

% Assume the following data
% Spread Option
Strike = 20;
OptSpec = 'call';
Settle =   '01-March-2013';
Maturity = '01-June-2013';
Corr = 0.45;     % Correlation of underlying commodities
```

Define the `RateSpec` and `StockSpec`.

```
% Define RateSpec
Rate = 0.035;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
    'EndDates', Maturity, 'Rates', Rate, 'Compounding', ...
    Compounding, 'Basis', Basis);

% Define StockSpec for the two assets
```

```
StockSpec1 = stockspec(Vol1, Price1);
StockSpec2 = stockspec(Vol2, Price2);
```

Price the Crack Spread Option

Use the function `spreadbybjs` in the Financial Instruments Toolbox™ to price the spread option using the Bjerksund and Stensland model.

```
Price = spreadbybjs(RateSpec, StockSpec1, StockSpec2, Settle, ...
                      Maturity, OptSpec, Strike, Corr)
```

```
Price =
          9.91
```

The 1:1 implied current crack spread between these two underlyings is $23.14 per barrel.

```
CrackSpread = Price1 - Price2    % $/barrel
```

```
CrackSpread =
         23.14
```

Suppose that by expiration day, June crude oil prices decrease to $90.34 per barrel and gasoline prices rise to $2.89 per gallon. The price changes cause the marketer's profit margin (the new implied crack spread) to increase from $23.14/barrel to $31.04/barrel:

```
NewCrackSpread = (2.89 * 42) - 90.34
```

```
NewCrackSpread =
         31.04
```

Since the marketer purchased a long crack call on the $20 call, the option is now in the money by $11.04.

```
(NewCrackSpread - Strike)
```

```
ans =
         11.04
```

The marketer paid $9.91 from the long crack call, this protects the margin by $1.13.

```
(NewCrackSpread - Strike - Price)
```

```
ans =
          1.13
```

This strategy provides the marketer protection during spread increase scenarios.

Example 2: Creating a Floor with Crack Spread Options

A refiner is interested in covering its fixed and operating costs, but still profit from a favorable move in the market. In March the May WTI crude oil futures are at $99.43 per barrel and RBOB gasoline futures contract are at $3.04 per gallon. The refiner believes that the spread between those commodities of $28.25 per barrel is favorable. Of this, $11 corresponds to operating and fixed costs,

and $17.25 is the net refining margin. The refiner's strategy is to sell the crack spread by selling 10 RBOB gasoline futures and buying 10 crude oil futures.

```
% Price and volatility of RBOB gasoline
Price1gallon = 3.04;          % $/gallon
Price1 = Price1gallon * 42;   % $/barrel
Vol1 = 0.35;
Div1 = 0.0783;

% Price and volatility of WTI crude oil
Price2 = 99.43;         % $/barrel
Vol2 = 0.38;
Div2 = 0.0571;
```

The refiner purchases 10 May RBOB gasoline crack spread puts with a strike price of $25.

```
% Spread Option
Strike = 25;
OptSpec = 'put';
Settle =   '01-March-2013';
Maturity = '01-May-2013';
Corr = 0.30;      % Correlation of underlying commodities
```

Define the `RateSpec` and `StockSpec`.

```
% Define RateSpec
Rate = 0.035;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
    'EndDates', Maturity, 'Rates', Rate, 'Compounding', ...
    Compounding, 'Basis', Basis);

% Define StockSpec for the two assets
StockSpec1 = stockspec(Vol1, Price1, 'Continuous', Div1);
StockSpec2 = stockspec(Vol2, Price2, 'Continuous', Div2);
```

Price the Crack Spread Option

Use the function `spreadbyfd` in the Financial Instruments Toolbox™ to price the American spread option using the finite difference method.

```
Price = spreadbyfd(RateSpec, StockSpec1, StockSpec2, Settle, ...
                   Maturity, OptSpec, Strike, Corr, 'AmericanOpt', 1)
```

```
Price =
          6.61
```

By expiration, if the option is exercised, the refiner would have hedged the cost of purchasing 10000 barrels of crude oil with the revenue of selling 10000 barrels of RBOB gasoline. The futures contract represents 1000 barrels of crude oil and 42000 gallons of gasoline.

```
CostOfHedge = Price * 10000   % Option premium
```

```
CostOfHedge =
      66122.24
```

The hedge cost is $66386 to implement and guarantee that neither a fall in RBOB gasoline prices or an increase in WTI crude oil prices will diminish the refining margin below $25.

```
ProfitMargin = 14 * 10000   %$
```

```
ProfitMargin =
     140000.00
```

```
CrackingMargin = ProfitMargin - CostOfHedge
```

```
CrackingMargin =
      73877.76
```

This strategy allows a cracking margin of $73613.

Another strategy for the refiner could be to buy the $22 puts at a price of $5.38.

```
StrikeNew = 22;

PriceNew = spreadbyfd(RateSpec, StockSpec1, StockSpec2, Settle, ...
                      Maturity, OptSpec, StrikeNew, Corr, 'AmericanOpt', 1)
```

```
PriceNew =
          5.36
```

This time the hedge would have cost $53823, but it also guarantees a $11 per barrel or a $56176 cracking margin.

```
NewCostOfHedge = PriceNew * 10000   % Option premium
```

```
NewCostOfHedge =
      53570.97
```

```
NewProfitMargin = 11 * 10000
```

```
NewProfitMargin =
     110000.00
```

```
CrackingMargin = NewProfitMargin - NewCostOfHedge
```

```
CrackingMargin =
      56429.03
```

Example 3: Using Collars to Reduce the Cost of Hedging

A refiner is concerned about its cost of hedging and decides to use a collar strategy. In April the crack spread is trading at $4.23 per barrel. The refiner is not convinced to lock in this margin, but also

wants to protect against price changes causing the refinery margin to decrease less than $4 per barrel.

```
% Price and volatility of heating oil
Price1gallon = 2.52;          % $/gallon
Price1 = Price1gallon * 42;   % $/barrel
Vol1 = 0.38;
Div1 = 0.0762;

% Price and volatility of WTI crude oil
Price2 = 101.61;         % $/barrel
Vol2 = 0.34;
Div2 = 0.1169;
```

To accomplish the collar strategy the refiner sells a call spread option with a strike of $4.50 and uses the premium income to offset the cost of purchasing a put spread option with a strike of $4. This allows the refiner to benefit if market prices move up, and protects it if market prices move down.

```
% Assume the following data
Strike = [4.50;4];
OptSpec = {'call';'put'};
Settle =   '01-April-2013';
Maturity = '01-June-2013';
Corr = 0.35;       % Correlation of underlying commodities
```

Define the `RateSpec` and `StockSpec`.

```
% Define RateSpec
Rate = 0.035;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
    'EndDates', Maturity, 'Rates', Rate, 'Compounding', ...
    Compounding, 'Basis', Basis);

% Define StockSpec for the two assets
StockSpec1 = stockspec(Vol1, Price1, 'Continuous', Div1);
StockSpec2 = stockspec(Vol2, Price2, 'Continuous', Div2);
```

Price the Crack Spread Options

Use the function `spreadbybjs` in the Financial Instruments Toolbox™ to price the spread options using the Bjerksund and Stensland model.

```
Price = spreadbybjs(RateSpec, StockSpec1, StockSpec2, Settle, ...
                      Maturity, OptSpec, Strike, Corr)
```

```
Price = 2×1

          7.06
          6.43
```

The collar strategy allows the refiner to reduce the cost of the hedge to $0.63.

```
% CostOfHedge = Premium of Call - Premium of Put
CostOfHedge = Price(1) - Price(2)
```

```
CostOfHedge =
          0.63
```

The refiner is protected if the crack spread narrows to less than $4. If the crack spread widens to more than $4.50, the refiner will not benefit over this amount if he has hedged 100% of all its market exposure.

```
set(0, 'format', OldFormat);
```

See Also

More About

- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Mortgage-Backed Securities

What Are Mortgage-Backed Securities?

Mortgage-backed securities (MBSs) are a type of investment that represents ownership in a group of mortgages. Principal and interest from the individual mortgages are used to pay principal and interest on the MBS.

Ownership in a group of mortgages is typically represented by a *pass-through certificate* (PC). Most pass-through certificates are issued by the Government National Mortgage Agency, a branch of the United States government, or by one of two private corporations: Fannie Mae or Freddie Mac. With these certificates, homeowners' payments pass from the originating bank through the issuing agency to holders of the certificates. These agencies also frequently guarantee that the certificate holder receives timely payment of principal and interest from the PCs.

See Also

`mbscfamounts` | `mbsconvp` | `mbsconvy` | `mbsdurp` | `mbsdury` | `mbsnoprepay` | `mbspassthrough` | `mbsprice` | `mbswal` | `mbsyield` | `mbsprice2speed` | `mbsyield2speed` | `psaspeed2default` | `psaspeed2rate` | `mbsoas2price` | `mbsoas2yield` | `mbsprice2oas` | `mbsyield2oas`

Related Examples

- “Fixed-Rate Mortgage Pool” on page 5-3

Fixed-Rate Mortgage Pool

In this section...

Introduction

Financial Instruments Toolbox software supports calculations involved with generic fixed-rate mortgage pools and balloon mortgages. Generic fixed-rate mortgage pools and balloon mortgages have pass-through certificates (PC) that typically have embedded call options in the form of prepayment. Prepayment is an excess payment applied to the principal of a PC. These accelerated payments reduce the effective life of a PC.

The toolbox comes with a standard Bond Market Association (PSA) prepayment model and can generate multiples of standard prepayment speeds. The Public Securities Association provides a set of uniform practices for calculating the characteristics of mortgage-backed securities when there is an assumed prepayment function.

Alternatively, aside from the standard PSA implementation in this toolbox, you can supply your own projected prepayment vectors. Currently, however, custom prepayment functionality that incorporates pool-specific information and interest rate forecasts are not available in this toolbox. If you plan to use custom prepayment vectors in your calculations, you presumably already own such a suite in MATLAB.

Inputs to Functions

Because of the generic, all-purpose nature of the toolbox pass-through functions, you can fine-tune them to conform to a particular mortgage. Most functions require at least this set of inputs:

- Gross coupon rate
- Settlement date
- Issue (effective) date
- Maturity date

Typical optional inputs include standard prepayment speed (or customized vector), net coupon rate (if different from gross coupon rate), and payment delay in number of days.

All calculations are based on expected payment dates and actual cash flow to the investor. For example, when `GrossRate` and `CouponRate` differ as inputs to `mbsdurp`, the function returns a modified duration based on `CouponRate`. (A notable exception is `mbspassthrough`, which returns interest quantities based on the `GrossRate`.)

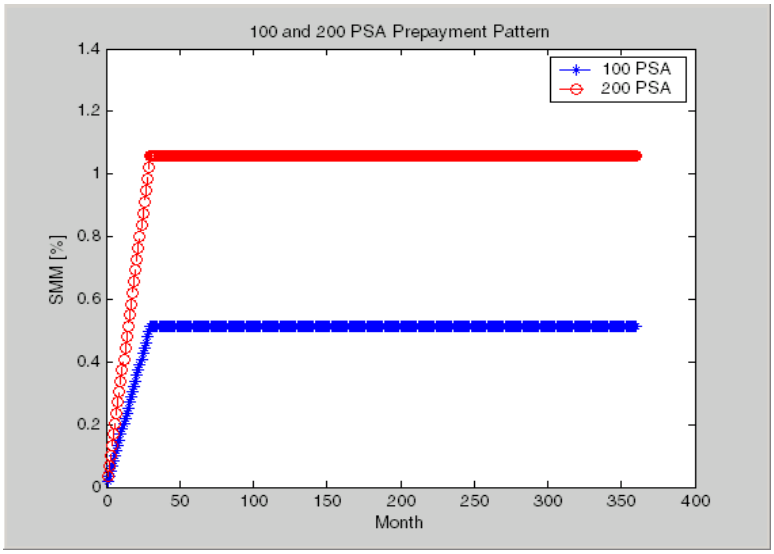
Generating Prepayment Vectors

You can generate PSA multiple prepayment vectors quickly. To generate prepayment vectors of 100 and 200 PSA, type

```
PSASpeed = [100, 200];
[CPR, SMM] = psaspeed2rate(PSASpeed)
```

This function computes two prepayment values: conditional prepayment rate (CPR) and single monthly mortality (SMM) rate. CPR is the percentage of outstanding principal prepaid in one year. SMM is the percentage of outstanding principal prepaid in one month. In other words, CPR is an annual version of SMM.

Since the entire 360-by-2 array is too long to show in this document, observe the SMM (100 and 200 PSA) plots, spaced one month apart, instead.



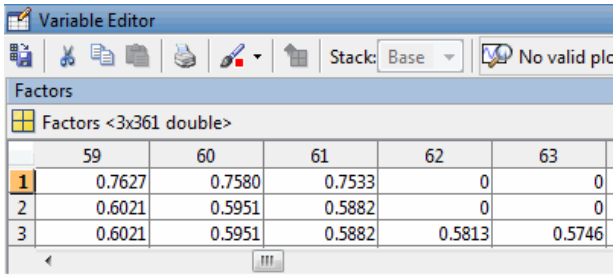
Prepayment assumptions form the basis upon which far more comprehensive MBS calculations are based. As an illustration, observe the following example, which shows the use of the function `mbscfamounts` to generate cash flows and timings based on a set of standard prepayments.

Consider three mortgage pools that were sold on the issue date (which starts unamortized). The first two pools "balloon out" in 60 months, and the third is regularly amortized to the end. The prepayment speeds are assumed to be 100, 200, and 200 PSA, respectively.

```
Settle     = [datetime(2000,2,1) ; datetime(2000,2,1) ; datetime(2000,2,1) ;datetime(2000,2,1)];

Maturity   = datetime(2030,2,1);

IssueDate  = datetime(2000,2,1);
GrossRate  = 0.08125;
CouponRate = 0.075;
Delay = 14;

PSASpeed = [100, 200];
[CPR, SMM] = psaspeed2rate(PSASpeed);

PrepayMatrix = ones(360,3);
```

```
PrepayMatrix(1:60,1:2) = SMM(1:60,1:2);
PrepayMatrix(:,3) = SMM(:,2);

[CFlowAmounts, CFlowDates, TFactors, Factors] = ...
mbscfamounts(Settle, Maturity, IssueDate, GrossRate, ...
CouponRate, Delay, [], PrepayMatrix);
```

The fourth output argument, `Factors`, indicates the fraction of the balance still outstanding at the beginning of each month. A snapshot of this argument in the MATLAB Variables editor illustrates the 60-month life of the first two of the mortgages with balloon payments and the continuation of the third mortgage until the end (360 months).



	59	60	61	62	63
1	0.7627	0.7580	0.7533	0	0
2	0.6021	0.5951	0.5882	0	0
3	0.6021	0.5951	0.5882	0.5813	0.5746

You can readily see that `mbscfamounts` is the building block of most fixed-rate and balloon pool cash flows.

Mortgage Prepayments

Prepayment is beneficial to the pass-through owner when a mortgage pool has been purchased at discount. The next example compares mortgage yields (compounded monthly) versus the purchase clean price with constant prepayment speed. The example illustrates that when you have purchased a pool at a discount, prepayment generates a higher yield with decreasing purchase price.

```
Price = [85; 90; 95];
Settle = datetime(2002,4,15);
Maturity = datetime(2030,1,1);
IssueDate = datetime(2000,1,1);
GrossRate = 0.08125;
CouponRate = 0.075;
Delay = 14;
Speed = 100;
```

Compute the mortgage and bond-equivalent yields.

```
[MYield, BEMBSYield] = mbsyield(Price, Settle, Maturity, ...
IssueDate, GrossRate, CouponRate, Delay, Speed)

MYield =

    0.1018
    0.0918
    0.0828

BEMBSYield =

    0.1040
```

```
    0.0936
    0.0842
```

If for this same pool of mortgages, there was no prepayment (`Speed = 0`), the yields would decline to

```
MYield =

    0.0926
    0.0861
    0.0802

BEMBSYield =

    0.0944
    0.0877
    0.0815
```

Likewise, if the rate of prepayment doubled (`Speed = 200`), the yields would increase to

```
MYield =

    0.1124
    0.0984
    0.0858

BEMBSYield =

    0.1151
    0.1004
    0.0873
```

For the same prepayment vector, deeper discount pools earn higher yields. For more information, see `mbsprice` and `mbsyield`.

Risk Measurement

Financial Instruments Toolbox provides the most basic risk measures of a pool portfolio:

- Modified duration
- Convexity
- Average life of pool

Consider the following example, which calculates the Macaulay and modified durations given the price of a mortgage pool.

```
Price = [95; 100; 105];
Settle = datetime(2002,4,15);
Maturity = datetime(2030,1,1);
IssueDate = datetime(2000,1,1);
GrossRate = 0.08125;
CouponRate = 0.075;
Delay = 14;
Speed = 100;

[YearDuration, ModDuration] = mbsdurp(Price, Settle, ...
Maturity, IssueDate, GrossRate, CouponRate, Delay, Speed)
```

```
YearDuration =

    6.1341
    6.3882
    6.6339

ModDuration =

    5.8863
    6.1552
    6.4159
```

Using Financial Instruments Toolbox functions, you can obtain modified duration and convexity from either price or yield, as long as you specify a prepayment vector or an assumed prepayment speed. The toolbox risk-measurement functions (`mbsdurp`, `mbsdury`, `mbsconvp`, `mbsconvy`, and `mbswal`) adhere to the guidelines listed in the *PSA Uniform Practices* manual.

Mortgage Pool Valuation

For accurate valuation of a mortgage pool, you must generate interest-rate paths and use them with mortgage pool characteristics to properly value the pool. A widely used methodology is the option-adjusted spread (OAS). OAS measures the yield spread that is not directly attributable to the characteristics of a fixed-income investment.

Calculating OAS

Prepayment alters the cash flows of an otherwise regularly amortizing mortgage pool. A comprehensive option-adjusted spread calculation typically begins with the generation of a set of paths of spot rates to predict prepayment. A path is collection of i spot-rate paths, with corresponding j cash flows on each of those paths.

The effect of the OAS on pool pricing is shown mathematically in the following equation, where K is the option-adjusted spread.

$$PoolPrice = \frac{1}{NumberofPaths} \times \sum_{i}^{NumberofPaths} \sum_{j}^{CF_{ij}} \frac{CF_{ij}}{(1 + zerorates_{ij} + K)^{T_{ij}}}$$

Calculating Effective Duration

Alternatively, if you are more interested in the sensitivity of a mortgage pool to interest rate changes, use effective duration, which is a more appropriate measure. Effective duration is defined mathematically with the following equation.

$$Effective\ Duration = \frac{P(y + \Delta y) - P(y - \Delta y)}{2P(y)\Delta y}$$

Calculating Market Price

The toolbox has all the components required to calculate OAS and effective duration if you supply prepayment vectors or assumptions. For OAS, given a prepayment vector, you can generate a set of cash flows with `mbscfamounts`. Discounting these cash flows with the reference curve and then adding OAS produces the market price. See “Computing Option-Adjusted Spread” on page 5-9 for a discussion on the computation of option-adjusted spread.

Effective duration is a more difficult issue. While modified duration changes the discounting process (by changing the yield used to discount cash flows), effective duration must account for the change in cash flow because of the change in yield. A possible solution is to recompute prices using `mbsprice` for a small change in yield, in both the upwards and downwards directions. In this case, you must recompute the prepayment input. Internally, this alters the cash flows of the mortgage pool. Assuming that the OAS stays constant in all yield environments, you can apply a set of discounting factors to the cash flows in up and down yield environments to find the effective duration.

See Also

`mbscfamounts` | `mbsconvp` | `mbsconvy` | `mbsdurp` | `mbsdury` | `mbsnoprepay` | `mbspassthrough` | `mbsprice` | `mbswal` | `mbsyield` | `mbsprice2speed` | `mbsyield2speed` | `psaspeed2default` | `psaspeed2rate` | `mbsoas2price` | `mbsoas2yield` | `mbsprice2oas` | `mbsyield2oas`

Related Examples

- “Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
- “Computing Option-Adjusted Spread” on page 5-9
- “Prepayments with Fewer Than 360 Months Remaining” on page 5-12
- “Pools with Different Numbers of Coupons Remaining” on page 5-14
- “Pricing Mortgage Backed Securities Using the Black-Derman-Toy Model” on page 5-34
- “Using Collateralized Mortgage Obligations (CMOs)” on page 5-40

More About

- “What Are Mortgage-Backed Securities?” on page 5-2

Computing Option-Adjusted Spread

The option-adjusted spread (OAS) is an amount of extra interest added above (or below if negative) the reference zero curve. To compute the OAS, you must provide the zero curve as an extra input. You can specify the zero curve in any intervals and with any compounding method. (To minimize any error due to interpolation, keep the intervals as regular and frequent as possible.) You must supply a prepayment vector or specify a speed corresponding to a standard PSA prepayment vector.

One way to compute the appropriate zero curve for an agency is to look at its bond yields and bootstrap them from the shortest maturity onwards. You can do this with Financial Toolbox™ functions `zbtprice` and `zbtyield`.

The following example shows how to calculate an appropriate zero curve followed by computation of the pool's OAS. This example calculates the OAS of a 30-year fixed rate mortgage with about a 28-year weighted average maturity left, given an assumption of 0, 50, and 100 PSA prepayment speeds.

Create curve for `zerorates`.

```
Bonds = [datenum('11/21/2002')   0        100    0    2    1;
         datenum('02/20/2003')   0        100    0    2    1;
         datenum('07/31/2004')   0.03     100    2    3    1;
         datenum('08/15/2007')   0.035    100    2    3    1;
         datenum('08/15/2012')   0.04875  100    2    3    1;
         datenum('02/15/2031')   0.05375  100    2    3    1];

Yields = [0.0162;
          0.0163;
          0.0211;
          0.0328;
          0.0420;
          0.0501];
```

Since the above is Treasury data and not selected agency data, a term structure of spread is assumed. In this example, the spread declines proportionally from a maximum of 250 basis points at the shortest maturity.

```
Yields = Yields + 0.025 * (1./[1:6]')
```

```
Yields =

    0.0412
    0.0288
    0.0294
    0.0391
    0.0470
    0.0543
```

Get parameters from `Bonds` matrix.

```
Settle = datetime(2002,8,20);
Maturity = Bonds(:,1);
CouponRate = Bonds(:,2);
Face = Bonds(:,3);
Period = Bonds(:,4);
Basis = Bonds(:,5);
EndMonthRule = Bonds(:,6);

[Prices, AccruedInterest] = bndprice(Yields, CouponRate, ...
Settle, Maturity, Period, Basis, EndMonthRule, [], [], [], [], ...
Face)
```

```
Prices =
```

```
   98.9747
   98.5804
  100.1040
   98.1802
  101.3808
   99.2535


AccruedInterest =

         0
         0
    0.1644
    0.0479
    0.0668
    0.0736
```

Use `zbtprice` to solve for zero rates.

```
[ZeroRatesP, CurveDatesP] = zbtprice(Bonds, Prices, Settle);
ZeroCompounding = 2*ones(size(ZeroRatesP));
ZeroMatrix = [CurveDatesP, ZeroRatesP, ZeroCompounding]

ZeroMatrix =

   1.0e+05 *

    7.3154    0.0000    0.0000
    7.3163    0.0000    0.0000
    7.3216    0.0000    0.0000
    7.3327    0.0000    0.0000
    7.3510    0.0000    0.0000
    7.4185    0.0000    0.0000
```

Use output from `zbtprice` to calculate the OAS.

```
Price = 95;
Settle = datenum('20-Aug-2002');
Maturity = datenum('2-Jan-2030');
IssueDate = datenum('2-Jan-2000');
GrossRate = 0.08125;
CouponRate = 0.075;
Delay = 14;
Interpolation = 1;
PrepaySpeed = [0; 50; 100];

OAS = mbsprice2oas(ZeroMatrix, Price, Settle, Maturity, ...
IssueDate, GrossRate, CouponRate, Delay, Interpolation, ...
PrepaySpeed)

OAS =

    26.0502
    28.6348
    31.2222
```

This example shows that one cash flow set is being discounted and solved for its OAS, as contrasted with the `NumberOfPaths` set of cash flows as shown in “Mortgage Pool Valuation” on page 5-7.

Averaging the sets of cash flows resulting from all simulations into one average cash flow vector and solving for the OAS, discounts the averaged cash flows to have a present value of today's (average) price.

While this example uses the mortgage pool price (`mbsprice2oas`) to determine the OAS, you can also use yield to resolve it (`mbsyield2oas`). Also, there are reverse OAS functions that return prices and yields given OAS (`mbsoas2price` and `mbsoas2yield`).

The example also restates earlier examples that show discount securities benefit from higher level of prepayment, keeping everything else unchanged. The relation is reversed for premium securities.

See Also

`mbscfamounts` | `mbsconvp` | `mbsconvy` | `mbsdurp` | `mbsdury` | `mbsnoprepay` | `mbspassthrough` | `mbsprice` | `mbswal` | `mbsyield` | `mbsprice2speed` | `mbsyield2speed` | `psaspeed2default` | `psaspeed2rate` | `mbsoas2price` | `mbsoas2yield` | `mbsprice2oas` | `mbsyield2oas`

Related Examples

- “Fixed-Rate Mortgage Pool” on page 5-3
- “Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
- “Pricing Mortgage Backed Securities Using the Black-Derman-Toy Model” on page 5-34
- “Prepayments with Fewer Than 360 Months Remaining” on page 5-12
- “Pools with Different Numbers of Coupons Remaining” on page 5-14
- “Using Collateralized Mortgage Obligations (CMOs)” on page 5-40

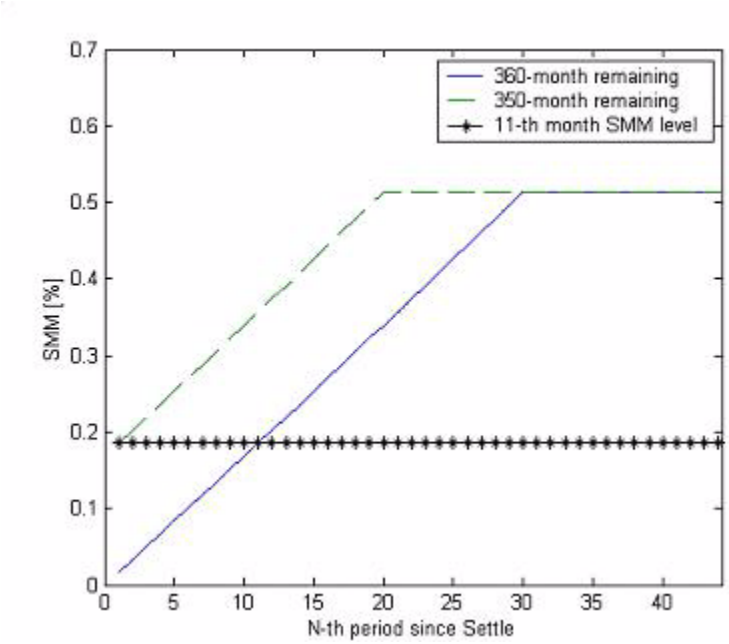
More About

- “What Are Mortgage-Backed Securities?” on page 5-2

Prepayments with Fewer Than 360 Months Remaining

When fewer than 360 months remain in the pool, the applicable PSA prepayment vector is "seasoned" by the pool's age. (Elements in the 360-element prepayment vector that represent past payments are skipped. For example, on a 30-year mortgage that is 10 months old, only the final 350 prepayments are applied.)

Assume, for example, that you have two 30-year loans, one new and another 10 months old. Both have the same PSA speed of 100 and prepay using the vectors plotted below.



Still within the scope of relative valuation, you could also solve for the percentage of the standard PSA prepayment vector given the pool's arbitrary, user-supplied prepayment vector, such that the PSA speed gives the same Macaulay duration as the user-supplied prepayment vector.

If you supply a custom prepayment vector, you must account for the number of months remaining.

```
Price = 101;
Settle = datetime(2001,1,1);
Maturity = datetime(2030,1,1);
IssueDate = datetime(2000,1,1);
GrossRate = 0.08125;
PrepayMatrix = 0.005*ones(348,1);
CouponRate   = 0.075;
Delay = 14;

ImpliedSpeed = mbsprice2speed(Price, Settle, Maturity, ...
IssueDate, GrossRate, PrepayMatrix, CouponRate, Delay)

ImpliedSpeed =

  104.2543
```

Examine the prepayment input. The remaining 29 years require 348 monthly elements in the prepayment vector. Suppose then, keeping everything the same, you change `Settle` to February 14, 2003.

```
Settle = datetime(2003,2,14);
```

You can use `cpncount` to count all incoming coupons received after `Settle` by invoking

```
NumCouponsRemaining = cpncount(Settle, Maturity, 12, 1, [], ...
IssueDate)
```

```
NumCouponsRemaining =
323
```

The input `12` defines the monthly payment frequency, `1` defines the 30/360 basis, and `IssueDate` defines aging and determination-of-holder date. Thus, you must supply a 323-element vector to account for a prepayment corresponding to each monthly payment.

See Also

`mbscfamounts` | `mbsconvp` | `mbsconvy` | `mbsdurp` | `mbsdury` | `mbsnoprepay` | `mbspassthrough` | `mbsprice` | `mbswal` | `mbsyield` | `mbsprice2speed` | `mbsyield2speed` | `psaspeed2default` | `psaspeed2rate` | `mbsoas2price` | `mbsoas2yield` | `mbsprice2oas` | `mbsyield2oas`

Related Examples

- “Fixed-Rate Mortgage Pool” on page 5-3
- “Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
- “Computing Option-Adjusted Spread” on page 5-9
- “Pricing Mortgage Backed Securities Using the Black-Derman-Toy Model” on page 5-34
- “Using Collateralized Mortgage Obligations (CMOs)” on page 5-40

More About

- “What Are Mortgage-Backed Securities?” on page 5-2

Pools with Different Numbers of Coupons Remaining

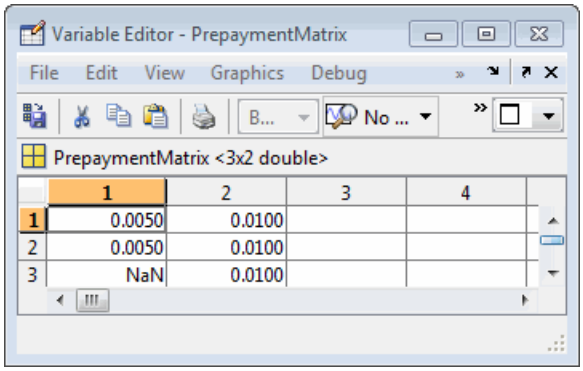
Suppose that one pool has two remaining coupons, and the other has three. MATLAB expects the prepayment matrix to be in the following format:

```
V11       V21
V12       V22
NaN       V23
```

V_{ij} denotes the single monthly mortality (SMM) rate for pool `i` during the `j`th coupon period since `Settle`.

The use of `NaN` to pad the prepayment matrix is necessary because MATLAB cannot concatenate vectors of different lengths into a matrix. Also, it can serve as an error check against any unintended operation (any MATLAB operation that would return `NaN`).

For example, assume that the 2-month pool has a constant SMM of 0.5% and the 3-month pool has a constant SMM of 1% in every period. The prepayment matrix you would create is depicted below.



Create this input in whatever manner is best for you.

Summary of Prepayment Data Vector Representation

- When you specify a PSA prepayment speed, MATLAB "seasons" the pool according to its age.
- When you specify your own prepayment matrix, identify the maximum number of coupons remaining using `cpncount`. Then supply the matrix elements up to the point when cash flow ceases to exist.
- When different length pools must exist in the same matrix, pad the shorter one(s) with `NaN`. Each column of the prepayment matrix corresponds to a specific pool.

See Also

`mbscfamounts` | `mbsconvp` | `mbsconvy` | `mbsdurp` | `mbsdury` | `mbsnoprepay` | `mbspassthrough` | `mbsprice` | `mbswal` | `mbsyield` | `mbsprice2speed` | `mbsyield2speed` | `psaspeed2default` | `psaspeed2rate` | `mbsoas2price` | `mbsoas2yield` | `mbsprice2oas` | `mbsyield2oas`

Related Examples

- “Fixed-Rate Mortgage Pool” on page 5-3

- “Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
- “Computing Option-Adjusted Spread” on page 5-9
- “Pricing Mortgage Backed Securities Using the Black-Derman-Toy Model” on page 5-34
- “Using Collateralized Mortgage Obligations (CMOs)” on page 5-40

More About

- “What Are Mortgage-Backed Securities?” on page 5-2

Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model

This example shows how to model prepayment in MATLAB® using functionality from the Financial Instruments Toolbox™. Specifically, a variation of the Richard and Roll prepayment model is implemented using a two factor Hull-White interest-rate model and a LIBOR Market Model to simulate future interest-rate paths. A mortgage-backed security is priced with both the custom and default prepayment models.

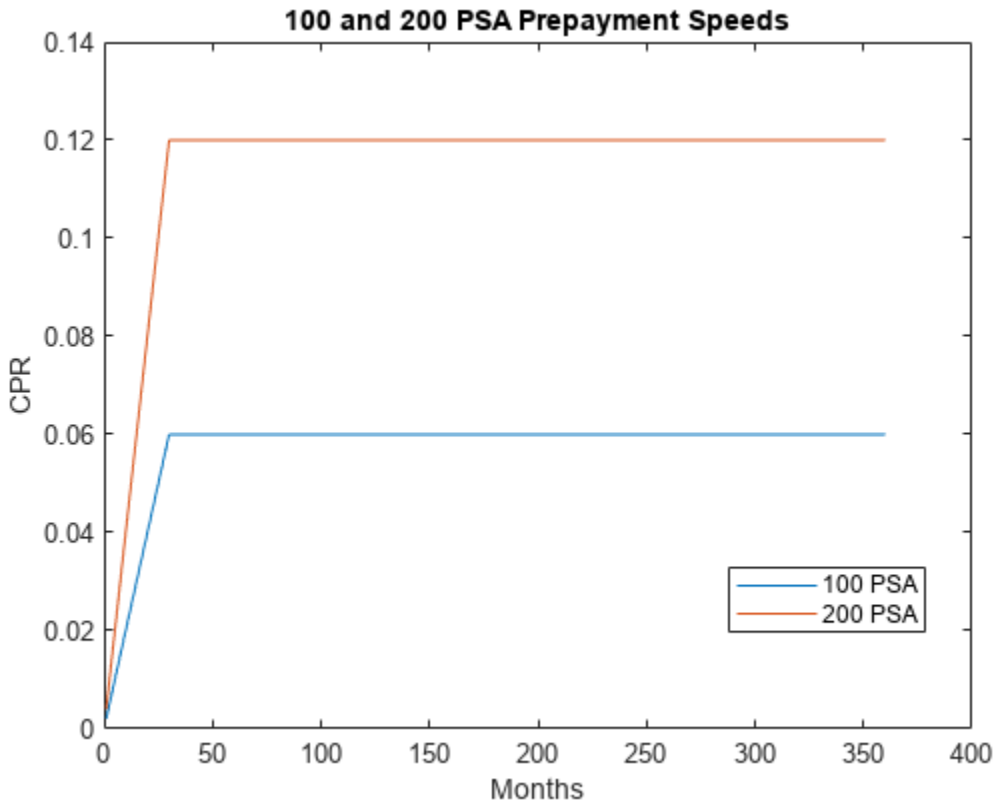
Introduction

Prepayment modeling is crucial to the analysis of mortgage-backed securities (MBS). Prepayments by individual mortgage holders affect both the amount and timing of cash flows and for collateralized mortgage obligations (for example, interest-only securities), prepayment can greatly affect the value of the securities.

PSA Model

The most basic prepayment model is the Public Securities Association (PSA) model, which assumes a ramp-up phase and then a constant conditional prepayment rate (CPR). The PSA model can be generated in MATLAB using the Financial Instruments Toolbox function `psaspeed2rate`.

```
G2PP_CPR = psaspeed2rate([100 200]);
figure
plot(G2PP_CPR)
title('100 and 200 PSA Prepayment Speeds')
xlabel('Months')
ylabel('CPR')
ylim([0 .14])
legend({'100 PSA','200 PSA'}, 'Location', 'Best')
```



Mortgage-Backed Security

The MBS analyzed in this example matures in 2020 and has the properties outlined in this section. Cash flows are generated for PSA prepayment speeds simply by entering the PSA speed as an input argument.

```
% Parameters for MBS passthrough to be priced
Settle = datetime(2007,12,15);
Maturity = datetime(2020,12,15);
IssueDate = datetime(2000,12,15);
GrossRate = .0475;
CouponRate = .045;
Delay = 14;
Period = 12;
Basis = 4;

% Generate cash flows and dates for baseline case using 100 PSA
[CFlowAmounts, CFlowDates] = mbscfamounts(Settle,Maturity, IssueDate,...
    GrossRate, CouponRate, Delay,100);
CFlowTimes = yearfrac(Settle,CFlowDates);
NumCouponsRemaining = cpncount(Settle, Maturity, Period,Basis, 1, IssueDate);
```

Richard and Roll Model

While prepayment modeling often involves complex and sophisticated modeling, often at the loan level, this example uses a slightly modified approach based on the model proposed by Richard and Roll [6 on page 5-32].

The Richard and Roll prepayment model involves the following factors:

- Refinancing incentive
- Seasonality (month of the year)
- Seasoning or age of the mortgage
- Burnout

Richard and Roll propose a multiplicative model of the following:

$$CPR = RefiIncentive * SeasoningMultiplier * SeasonalityMultiplier * BurnoutMultiplier$$

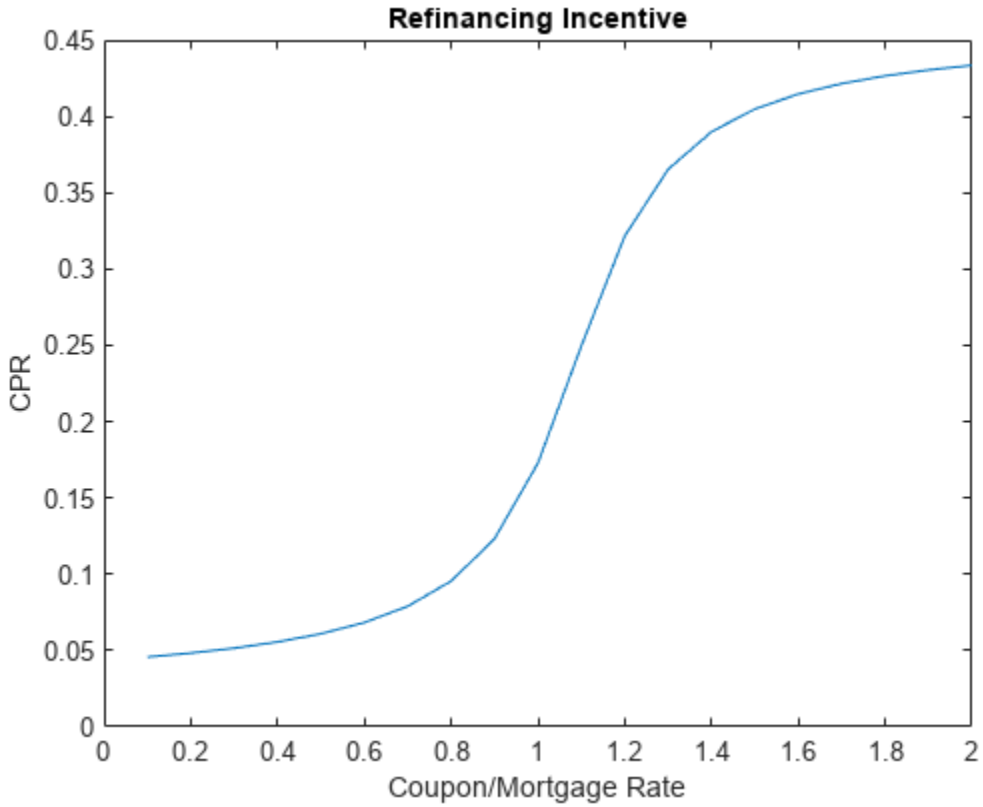
For the custom model in this example, the *Burnout Multiplier*, which describes the tendency of prepayment to slow when a significant number of homeowners have already refinanced, is ignored and the first three terms are used.

The refinancing incentive is a function of the ratio of the coupon-rate of the mortgage to the available mortgage rate at that particular point in time. For example, the Office of Thrift Supervision (OTS) proposes the following model:

$$Refi = .2406 - .1389 * arctan(5.952 * (1.089 - \frac{CouponRate}{MortgageRate}))$$

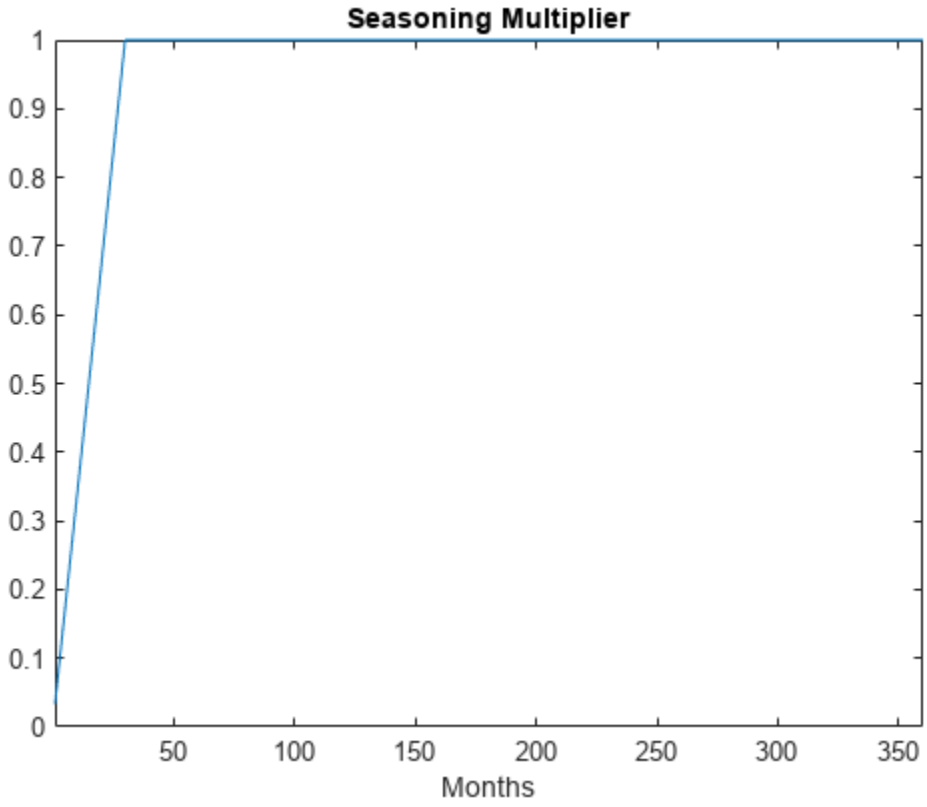
The refinancing incentive requires a simulation of future interest rates. This will be discussed later in this example.

```
C_M = .1:.1:2;
G2PP_Refi = .2406 - .1389 * atan(5.952*(1.089 - C_M));
figure
plot(C_M,G2PP_Refi)
xlabel('Coupon/Mortgage Rate')
ylabel('CPR')
title('Refinancing Incentive')
```



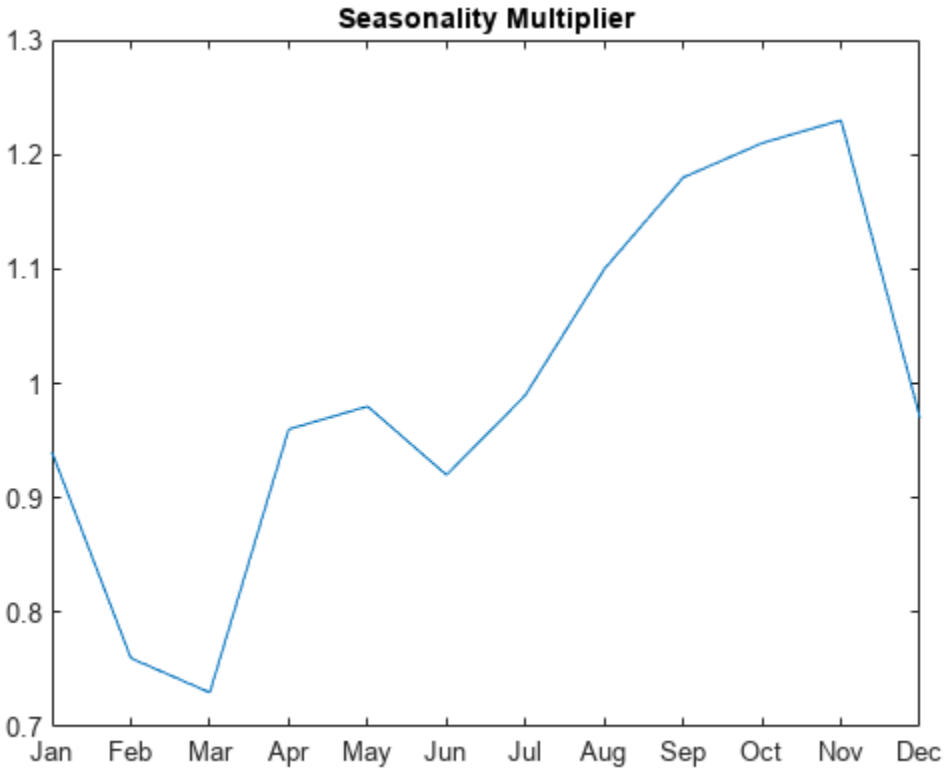
Seasoning captures the tendency of prepayment to ramp up at the beginning of a mortgage before leveling off. The OTS models the seasoning multiplier as follows:

```
Seasoning = ones(360,1);
Seasoning(1:29) = (1:29)/30;
figure
plot(Seasoning)
xlim([1 360])
title('Seasoning Multiplier')
xlabel('Months')
```



The seasonality multiplier simply models the seasonal behavior of prepayments -- this data is based on Figure 3 of [6 on page 5-32], which applies to the behavior of Ginnie Mae 30-year, single-family MBSs.

```
Seasonality = [.94 .76 .73 .96 .98 .92 .99 1.1 1.18 1.21 1.23 .97];
figure
plot(Seasonality)
xlim([1 12])
ax = gca;
ax.XTick = 1:12;
ax.XTickLabel = {'Jan','Feb','Mar','Apr','May','Jun','Jul','Aug', ...
    'Sep','Oct','Nov','Dec'};
title('Seasonality Multiplier')
```



G2++ Interest-Rate Model

Since the refinancing incentive requires a simulation of future interest rates, an interest-rate model must be used. One choice is a two-factor additive Gaussian model, referred to as G2++ by Brigo and Mercurio [2 on page 5-32].

The G2++ Interest Rate Model is:

$$r(t) = x(t) + y(t) + \varphi(t)$$

$$dx(t) = -ax(t)dt + \sigma dW_1(t)$$

$$dy(t) = -by(t)dt + \eta dW_2(t)$$

where $dW_1(t)dW_2(t)$ is a two-dimensional Brownian motion with correlation ρ

$$dW_1(t)dW_2(t) = \rho dt$$

$$\varphi(T) = f^M(0,T) + \frac{\sigma^2}{2a^2}(1-e^{-aT})^2 + \frac{\eta^2}{2b^2}(1-e^{-bT})^2 + \rho\frac{\sigma\eta}{ab}(1-e^{-aT})(1-e^{-bT})$$

and $r(t)$ is the short rate, a and b are mean reversion constants and σ and η are volatility constants, and $f^M(0,T)$ is the market forward rate, or the forward rate observed on the Settle date.

LIBOR Market Model

The LIBOR Market Model (LMM) differs from short-rate models in that it evolves a set of discrete forward rates. Specifically, the lognormal LMM specifies the following diffusion equation for each forward rate:

$$\frac{dF_i(t)}{F_i} = -\mu_i dt + \sigma_i(t) dW_i$$

where

dW is an N dimensional geometric Brownian motion with:

$$dW_i(t) dW_j(t) = \rho_{i,j} dt$$

The LMM relates the drifts of the forward rates based on no-arbitrage arguments. Specifically, under the Spot LIBOR measure, the drifts are expressed as the following:

$$\mu_i(t) = -\sigma_i(t) \sum_{j=q(t)}^{i} \frac{\tau_j \rho_{i,j} \sigma_j(t) F_j(t)}{1 + \tau_j F_j(t)}$$

where

τ_i is the time fraction associated with the ith forward rate

$q(t)$ is an index function defined by the relation $T_{q(t)-1} < t < T_{q(t)}$

and the Spot LIBOR numeraire is defined as the following:

$$B(t) = P(t, T_{q(t)}) \prod_{n=0}^{q(t)-1} (1 + \tau_n F_n(T_n))$$

Given the above, the choice with the LMM is how to model volatility and correlation.

The volatility of the rates can be modeled with a stochastic volatility, but for this example a deterministic volatility is used, and so a functional form needs to be specified. One of the most popular functional forms in the literature is the following:

$$\sigma_i(t) = \phi_i(a(T_i - t) + b)e^{c(T_i - t)} + d$$

where ϕ adjusts the curve to match the volatility for the i^{th} forward rate.

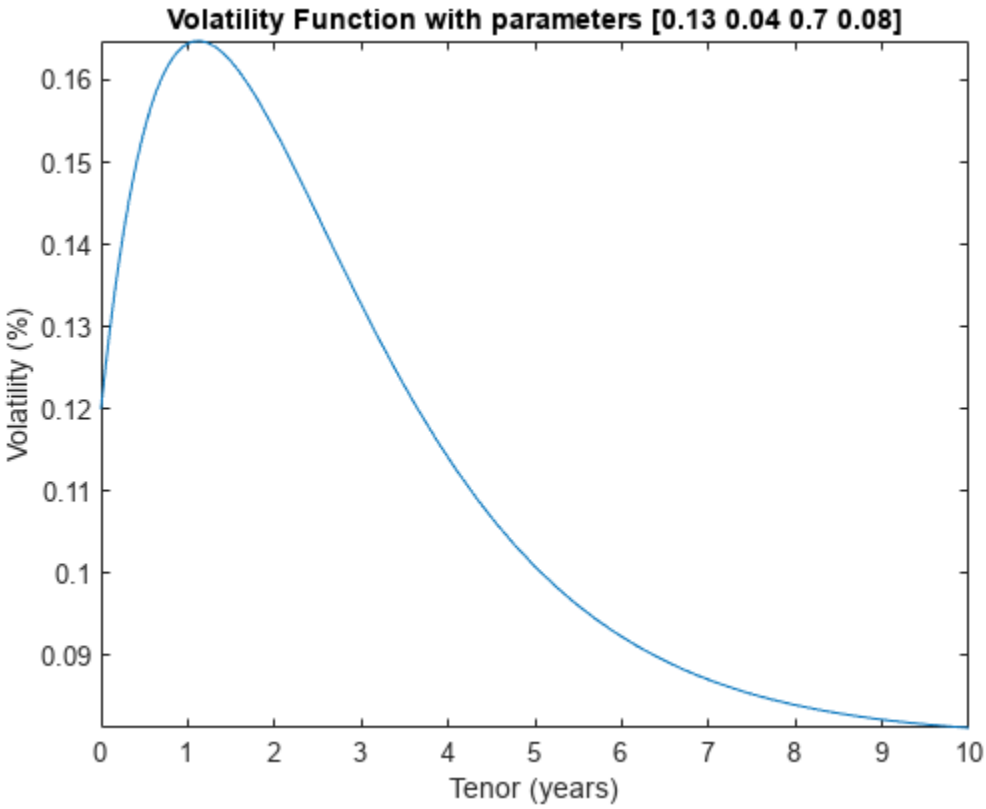
Similarly, the correlation between the forward rates needs to be specified. This can be estimated from historical data or fitted to option prices. For this example, the following functional form will be used:

$$\rho_{i,j} = e^{-\beta |i-j|}$$

Once the volatility and correlation are specified, the parameters need to be calibrated -- this can be done with historical or market data, typically swaptions or caps and floors. For this example, we simply use reasonable estimates for the correlation and volatility parameters.

```
% The volatility function to be used -- and one choice for the parameters
LMMVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
```

```
LMMVolParams = [.13 .04 .7 .08];

% Volatility specification
fplot(@(t) LMMVolFunc(LMMVolParams,t),[0 10])
title(['Volatility Function with parameters ' mat2str(LMMVolParams)])
ylabel('Volatility (%)')
xlabel('Tenor (years)')
```



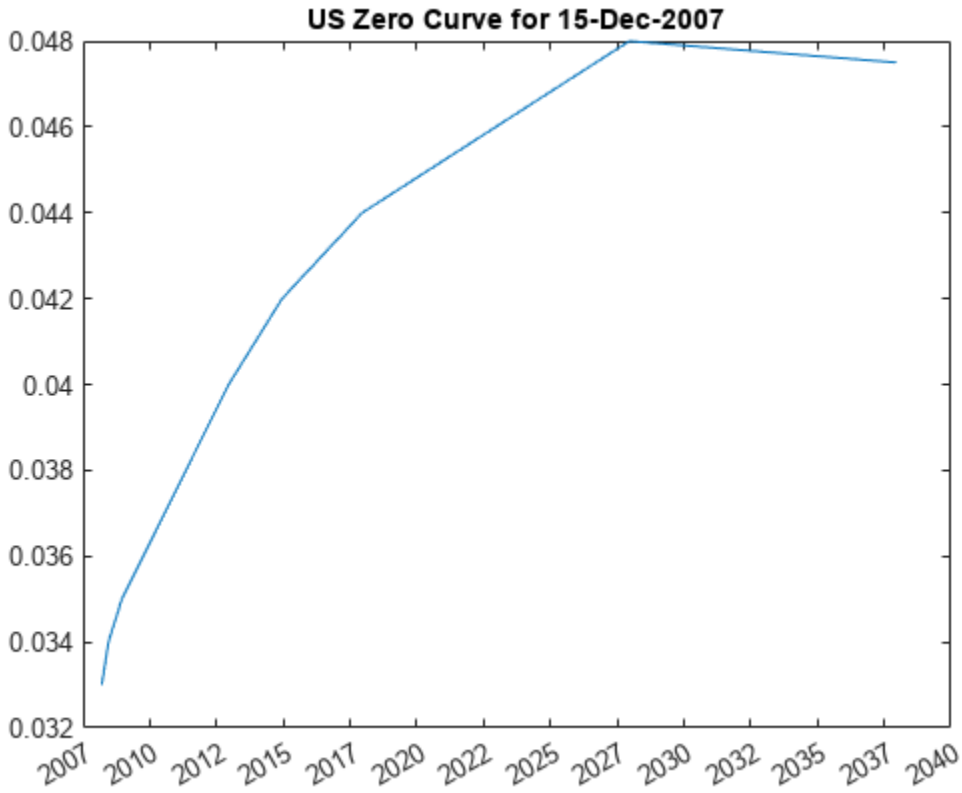
Calibration to Market Data

The parameters in the G2++ model can be calibrated to market data. Typically, the parameters are calibrated to observed interest-rate cap, floor and/or swaption data. For now, market cap data is used for calibration.

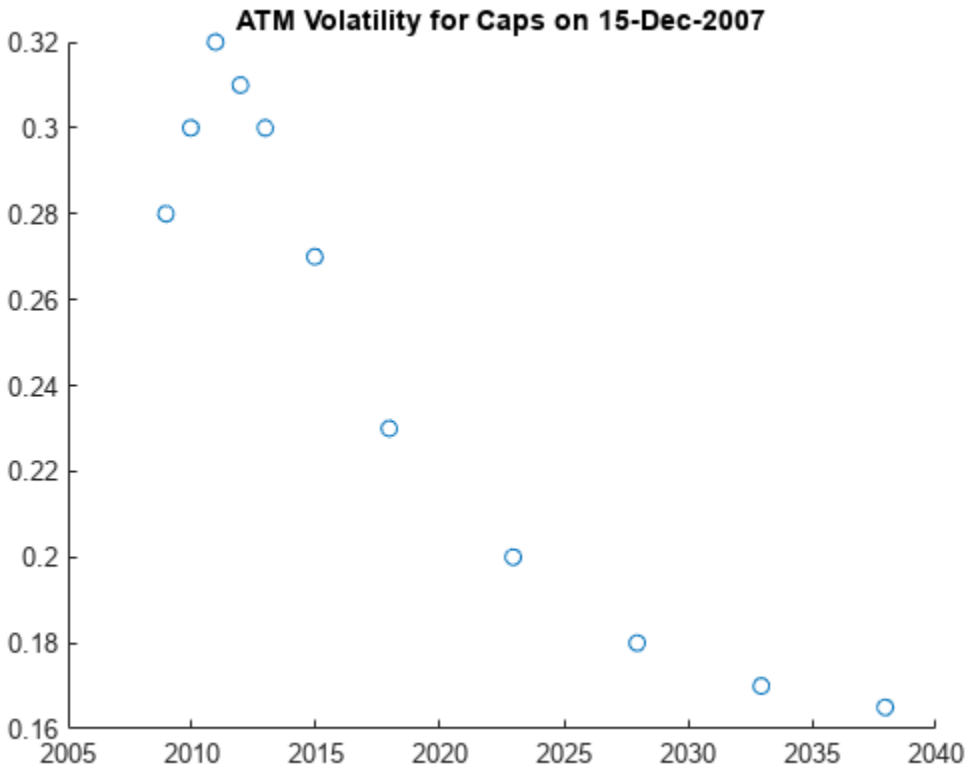
This data is hardcoded but could be imported into MATLAB with the Database Toolbox™ or Datafeed Toolbox™.

```
% Zero Curve -- this data is hardcoded for now, but could be bootstrapped
% using the bootstrap method of IRDataCurve.
ZeroTimes = [3/12 6/12 1 5 7 10 20 30]';
ZeroRates = [0.033 0.034 0.035 0.040 0.042 0.044 0.048 0.0475]';
ZeroDates = daysadd(Settle,360*ZeroTimes,1);
DiscountRates = zero2disc(ZeroRates,ZeroDates,Settle);
irdc = IRDataCurve('Zero',Settle,ZeroDates,ZeroRates);

figure
plot(ZeroDates,ZeroRates)
```

```
datetick
title(['US Zero Curve for ' datestr(Settle)])
```




```
% Cap Data
Reset = 2;
Notional = 100;
CapMaturity = daysadd(Settle,360*[1:5 7 10 15 20 25 30],1);
CapVolatility = [.28 .30 .32 .31 .30 .27 .23 .2 .18 .17 .165]';

% ATM strikes could be computed with swapbyzero
Strike = [0.0353 0.0366 0.0378 0.0390 0.0402 0.0421 0.0439 ...
    0.0456 0.0471 0.0471 0.0471]';

% This could be computed with capbyblk
BlackCapPrices = [0.1532 0.6416 1.3366 2.0290 2.7366 4.2960 6.5992 ...
    9.6787 12.2580 14.0969 15.7873]';

figure
scatter(CapMaturity,CapVolatility)
datetick
title(['ATM Volatility for Caps on ' datestr(Settle)])
```

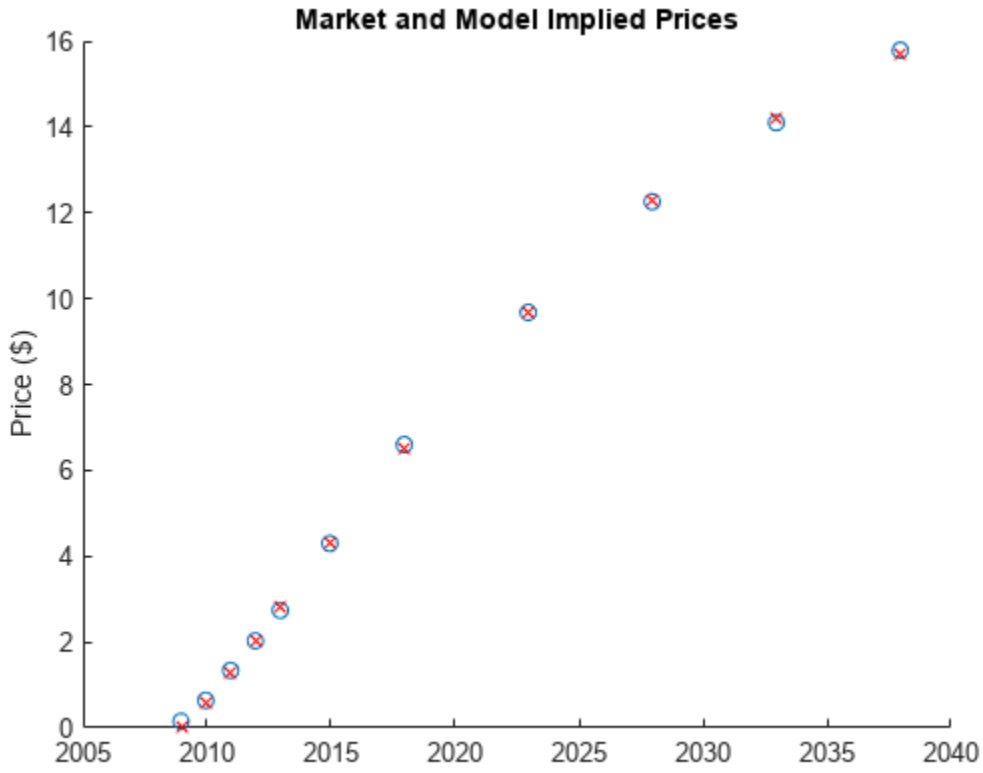


To calibrate the model parameters, a parameter set will be found that minimizes the sum of the squared differences between the G2++ predicted Cap values and the observed Black Cap values. The Optimization Toolbox™ function `lsqnonlin` is used in this example, although other approaches (for example, Global Optimization) may also be applicable. The function `capbylg2f` computes the analytic values for the caps given parameter values.

Upper and lower bounds for the model parameters are set to be relatively constrained. As Brigo and Mercurio discuss, the correlation parameter, *rho*, can often be close to `-1` when fitting a G2++ model to interest-rate cap prices. Therefore, *rho* is constrained to be between `-.7` and `.7` to ensure that the parameters represent a truly two-factor model. The remaining mean reversion and volatility parameters are constrained to be between `0` and `.5`. Calibration remains a complex task, and while the plot below indicates that the best fit parameters seem to do a reasonably good job of reproducing the Cap prices, it should be noted that the procedure outlined here simply represents one approach.

```
% Call to lsqnonlin to calibrate parameters
objfun = @(x) BlackCapPrices - capbylg2f(irdc,x(1),x(2),x(3),x(4),x(5),Strike,CapMaturity);
x0 = [.5 .05 .1 .01 -.1];
lb = [0 0 0 0 -.7];
ub = [.5 .5 .5 .5 .7];

G2PP_Params = lsqnonlin(objfun,x0,lb,ub);

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.
```

```
a = G2PP_Params(1);
b = G2PP_Params(2);
sigma = G2PP_Params(3);
eta = G2PP_Params(4);
rho = G2PP_Params(5);

% Compare the results
figure
scatter(CapMaturity,BlackCapPrices)
hold on
scatter(CapMaturity,capbylg2f(irdc,a,b,sigma,eta,rho,Strike,CapMaturity),'rx')
datetick
title('Market and Model Implied Prices')
ylabel('Price ($)')
```



G2++ Model Implementation

The `LinearGaussian2F` model can be used to specify the G2++ model and simulate future paths interest rates.

```
% G2++ model from Brigo and Mercurio with time homogeneous volatility
% parameters
G2PP = LinearGaussian2F(irdc,a,b,sigma,eta,rho);
```

LIBOR Market Model Implementation

After the volatility and correlation have been calibrated, Monte Carlo simulation is used to evolve the rates forward in time. The `LiborMarketModel` object is used to simulate the forward rates.

While factor reduction is often used with the LMM to reduce computational complexity, there is no factor reduction in this example.

6M LIBOR rates are chosen to be evolved in this simulation. Since a monthly prepayment vector must be computed, interpolation is used to generate the intermediate rates. Simple linear interpolation is used.

```
numForwardRates = 46;

% Instead of being fit, VolPhi is simply hard-coded  --
% representative of a declining volatility over time.
VolPhi = linspace(1.2,.8,numForwardRates-1)';

Beta = .08;
CorrFunc = @(i,j,Beta) exp(-Beta*abs(i-j));
CorrMat = CorrFunc(meshgrid(1:numForwardRates-1)',meshgrid(1:numForwardRates-1),Beta);

VolFunc = cell(length(VolPhi),1);
for jdx = 1:length(VolPhi)
    VolFunc(jdx) = {@(t) VolPhi(jdx)*ones(size(t)).*(LMMVolParams(1)*t + ...
        LMMVolParams(2)).*exp(-LMMVolParams(3)*t) + LMMVolParams(4)};
end

LMM = LiborMarketModel(irdc,VolFunc,CorrMat);
```

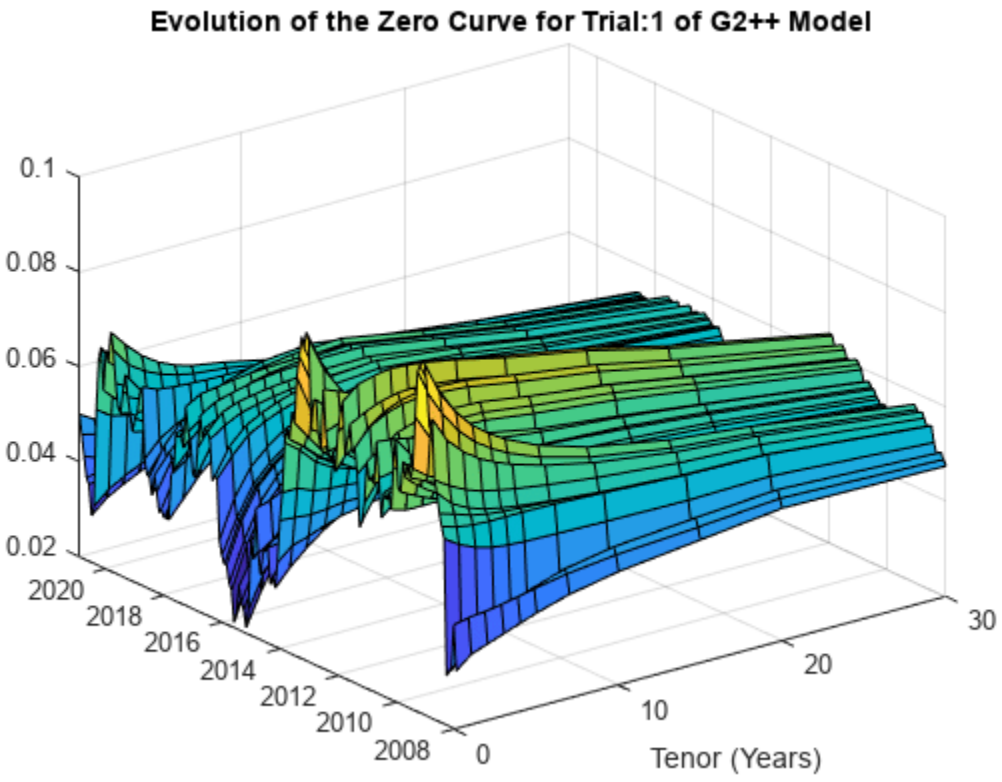
G2++ Monte Carlo Simulation

The various interest-rate paths can be simulated by calling the `simTermStructs` method.

One limitation to two-factor Gaussian models like this one is that it does permit negative interest rates. This is a concern, particularly in low interest-rate environments. To handle this possibility, any interest-rate paths with negative rates are simply rejected.

```
nPeriods = NumCouponsRemaining;
nTrials = 100;
DeltaTime = 1/12;

% Generate factors and short rates
Tenor = [1/12 1 2 3 4 5 7 10 15 20 30];
G2PP_SimZeroRates = G2PP.simTermStructs(nPeriods,'NTRIALS',nTrials,...
    'Tenor',Tenor,'DeltaTime',DeltaTime);

SimDates = daysadd(Settle,360*DeltaTime*(0:nPeriods),1);

% Tenors that will be recovered for each simulation date. The stepsize is
% included here to facilitate computing a discount factor for each
% simulation path.

% Remove any paths that go negative
NegIdx = squeeze(any(any(G2PP_SimZeroRates < 0,1),2));
G2PP_SimZeroRates(:,:,NegIdx) = [];
nTrials = size(G2PP_SimZeroRates,3);

% Plot evolution of one sample path
trialIdx = 1;
figure
surf(Tenor,SimDates,G2PP_SimZeroRates(:,:,trialIdx))
datetick y keepticks keeplimits
```

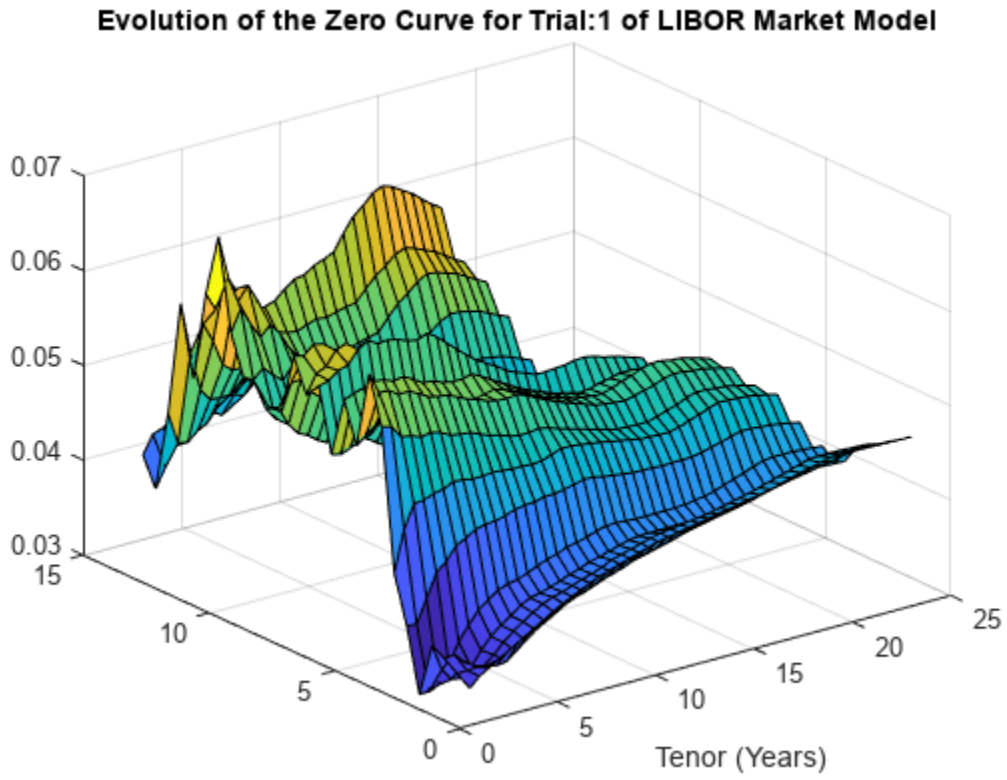
```
title(['Evolution of the Zero Curve for Trial:' num2str(trialIdx) ' of G2++ Model'])
xlabel('Tenor (Years)')
```



LIBOR Market Model Simulation

The various interest-rate paths can be simulated by calling the `simTermStructs` method of the `LiborMarketModel` object.

```
LMMPeriod = 2; % Semiannual rates
LMMNumPeriods = NumCouponsRemaining/12*LMMPeriod; % Number of semiannual periods
LMMDeltaTime = 1/LMMPeriod;
LMMNTRIALS = 100;

% Simulate
[LMMZeroRates, LMMForwardRates] = LMM.simTermStructs(LMMNumPeriods,'nTrials',LMMNTRIALS,'DeltaTi
ForwardTimes = 1/2:1/2:numForwardRates/2;
LMMSimTimes = 0:1/LMMPeriod:LMMNumPeriods/LMMPeriod;

% Plot evolution of one sample path
trialIdx = 1;
figure
tmpPlotData = LMMZeroRates(:,:,trialIdx);
tmpPlotData(tmpPlotData == 0) = NaN;
surf(ForwardTimes,LMMSimTimes,tmpPlotData)
title(['Evolution of the Zero Curve for Trial:' num2str(trialIdx) ' of LIBOR Market Model'])
xlabel('Tenor (Years)')
```



Compute Mortgage Rates from Simulation

Once the interest-rate paths have been simulated, the mortgage rate needs to be computed -- one approach, discussed by [7 on page 5-32], is to compute the mortgage rate from a combination of the 2-year and 10-year rates.

For this example, the following is used:

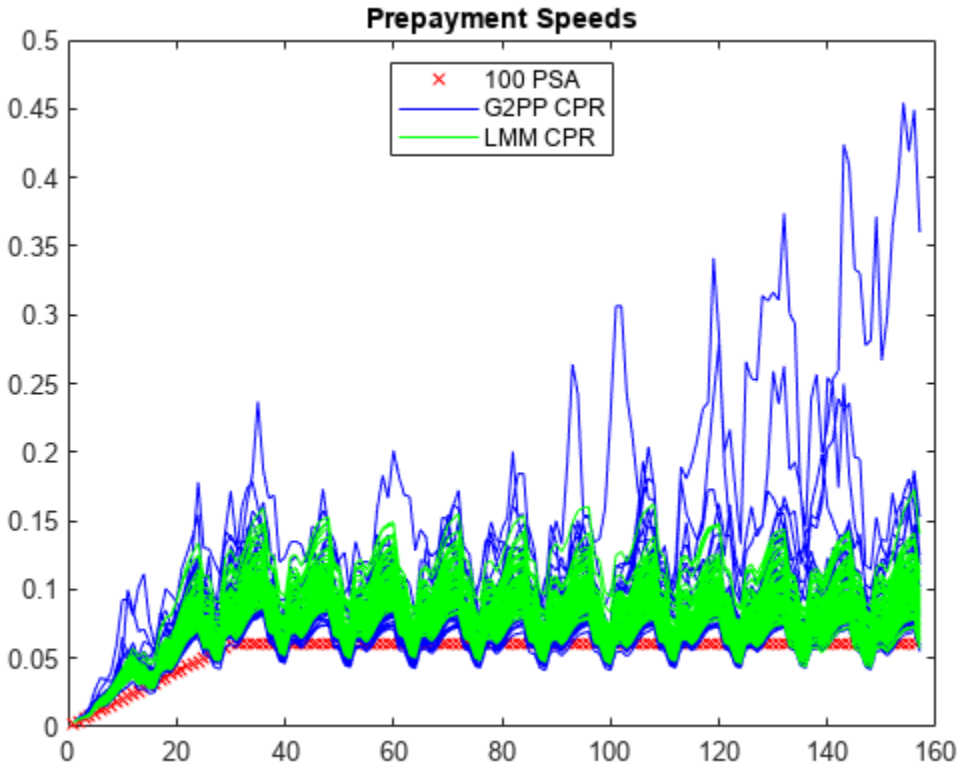
$$MortgageRate = .024 + .2*TwoYearRate + .6*TenYearRate$$

```
% Compute mortgage rates from interest rate paths
TwoYearRates = squeeze(G2PP_SimZeroRates(:,Tenor == 2,:));
TenYearRates = squeeze(G2PP_SimZeroRates(:,Tenor == 7,:));
G2PP_MortgageRates = .024 + .2*TwoYearRates + .6*TenYearRates;

LMMMortgageRates = squeeze(.024 + .2*LMMZeroRates(:,4,:) + .6*LMMZeroRates(:,20,:));
LMMDiscountFactors = squeeze(cumprod(1./(1 + LMMZeroRates(:,1,:)*.5)));

% Interpolate to get monthly mortgage rates
MonthlySimTimes = 0:1/12:LMMNumPeriods/LMMPeriod;
LMMMonthlyMortgageRates = zeros(nPeriods+1,LMMNTRIALS);
LMMMonthlyDF = zeros(nPeriods+1,LMMNTRIALS);
for trialidx = 1:LMMNTRIALS
   LMMMonthlyMortgageRates(:,trialidx) = interp1(LMMSimTimes,LMMMortgageRates(:,trialidx),Monthly
   LMMMonthlyDF(:,trialidx) = interp1(LMMSimTimes,LMMDiscountFactors(:,trialidx),MonthlySimTimes
end
```

Computing CPR and Generating and Valuing Cash Flows

Once the Mortgage Rates have been simulated, the CPR can be computed from the multiplicative model for each interest-rate path.

```
% Compute Seasoning and Refinancing Multipliers
Seasoning = ones(nPeriods+1,1);
Seasoning(1:30) = 1/30*(1:30);
G2PP_Refi = .2406 - .1389 * atan(5.952*(1.089 - CouponRate./G2PP_MortgageRates));
LMM_Refi = .2406 - .1389 * atan(5.952*(1.089 - CouponRate./LMMMonthlyMortgageRates));

% CPR is simply computed by evaluating the multiplicative model
G2PP_CPR = bsxfun(@times,G2PP_Refi,Seasoning.*(Seasonality(month(CFlowDates))'));
LMM_CPR = bsxfun(@times,LMM_Refi,Seasoning.*(Seasonality(month(CFlowDates))'));

% Compute single monthly mortality (SMM) from CPR
G2PP_SMM = 1 - (1 - G2PP_CPR).^(1/12);
LMM_SMM = 1 - (1 - LMM_CPR).^(1/12);

% Plot CPR's against 100 PSA
CPR_PSA100 = psaspeed2rate(100);
figure
PSA_handle = plot(CPR_PSA100(1:nPeriods),'rx');
hold on
G2PP_handle = plot(G2PP_CPR,'b');
LMM_handle = plot(LMM_CPR,'g');
title('Prepayment Speeds')
legend([PSA_handle(1) G2PP_handle(1) LMM_handle(1)],{'100 PSA','G2PP CPR','LMM CPR'},'Location',
```



Generate Cash Flows and Compute Present Value

With a vector of single monthly mortalities (SMM) computed for each interest-rate path, cash flows for the MBS can be computed and discounted.

```
% Compute the baseline zero rate at each cash flow time
CFlowZero = interp1(ZeroTimes,ZeroRates,CFlowTimes,'linear','extrap');

% Compute DF for each cash flow time
CFlowDF_Zero = zero2disc(CFlowZero,CFlowDates,Settle);

% Compute the price of the MBS using the zero curve
Price_Zero = CFlowAmounts*CFlowDF_Zero';

% Generate the cash flows for each IR Path
G2PP_CFlowAmounts = mbscfamounts(Settle, ...
    repmat(Maturity,1,nTrials), IssueDate, GrossRate, CouponRate, Delay, [], G2PP_SMM(2:end,:));

% Compute the DF for each IR path
G2PP_CFlowDFSim = cumprod(exp(squeeze(-G2PP_SimZeroRates(:,1,:).*DeltaTime)));

% Present value the cash flows for each MBS
G2PP_Price_Ind = sum(G2PP_CFlowAmounts.*G2PP_CFlowDFSim',2);
G2PP_Price = mean(G2PP_Price_Ind);

% Repeat for LMM
LMM_CFlowAmounts = mbscfamounts(Settle, ...
```

```
    repmat(Maturity,1,LMMNTRIALS), IssueDate, GrossRate, CouponRate, Delay, [], LMM_SMM(2:end,:))

% Present value the cash flows for each MBS
LMM_Price_Ind = sum(LMM_CFlowAmounts.*LMMMonthlyDF',2);
LMM_Price = mean(LMM_Price_Ind);
```

The results from the different approaches can be compared. The number of trials for the G2++ model will typically be less than 100 due to the filtering out of any paths that produce negative interest rates.

Additionally, while the number of trials for the G2++ model in this example is set to be 100, it is often the case that a larger number of simulations need to be run to produce an accurate valuation.

```
fprintf('                     # of Monte Carlo Trials: %8d\n'    , nTrials)
```

```
                    # of Monte Carlo Trials:       72
```

```
fprintf('                     # of Time Periods/Trial: %8d\n\n'  , nPeriods)
```

```
                    # of Time Periods/Trial:      156
```

```
fprintf('                      MBS Price with PSA 100: %8.4f\n'  , Price_Zero)
```

```
                     MBS Price with PSA 100:   1.0187
```

```
fprintf(' MBS Price with Custom G2PP Prepayment Model: %8.4f\n\n', G2PP_Price)
```

```
 MBS Price with Custom G2PP Prepayment Model:   0.9884
```

```
fprintf(' MBS Price with Custom LMM Prepayment Model: %8.4f\n\n', LMM_Price)
```

```
 MBS Price with Custom LMM Prepayment Model:   0.9993
```

Conclusion

This example shows how to calibrate and simulate a G2++ interest-rate model and how to use the generated interest-rate paths in a prepayment model loosely based on the Richard and Roll model. This example also provides a useful starting point to using the G2++ and LMM interest-rate models in other financial applications.

Bibliography

This example is based on the following books, papers, and journal articles:

1. Andersen, L. and V. Piterbarg. *Interest Rate Modeling*. Atlantic Financial Press, 2010.
2. Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice with Smile, Inflation and Credit* (2nd ed. 2006 ed.). Springer Verlag. ISBN 978-3-540-22149-4, 2001.
3. Hayre, L., ed., *Salomon Smith Barney Guide to Mortgage-Backed and Asset-Backed Securities*. New York: John Wiley & Sons, 2001.
4. Karpishpan, Y., O. Turel, and A. Hasha. "Introducing the Citi LMM Term Structure Model for Mortgages". *The Journal of Fixed Income.* Volume 20. 44-58, 2010.
5. Rebonato, R., K. McKay, and R. White (2010). *The Sabr/Libor Market Model: Pricing, Calibration and Hedging for Complex Interest-Rate Derivatives*. John Wiley & Sons, 2010.
6. Richard, S. F., and R. Roll. "Prepayments on Fixed Rate Mortgage-Backed Securities". *Journal of Portfolio Management.* 1989.

7 Office of Thrift Supervision, "Net Portfolio Value Model Manual". March 2000.

8 Stein, H. J., Belikoff, A. L., Levin, K. and Tian, X. "Analysis of Mortgage Backed Securities: Before and after the Credit Crisis" (January 5, 2007). "Credit Risk Frontiers: Subprime Crisis, Pricing and Hedging, CVA, MBS, Ratings, and Liquidity". Bielecki, Tomasz,; Damiano Brigo and Frederic Patras., eds., February 2011. Available at SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=955358

See Also

`mbscfamounts` | `mbsconvp` | `mbsconvy` | `mbsdurp` | `mbsdury` | `mbsnoprepay` | `mbspassthrough` | `mbsprice` | `mbswal` | `mbsyield` | `mbsprice2speed` | `mbsyield2speed` | `psaspeed2default` | `psaspeed2rate` | `mbsoas2price` | `mbsoas2yield` | `mbsprice2oas` | `mbsyield2oas`

Related Examples

- “Pricing Mortgage Backed Securities Using the Black-Derman-Toy Model” on page 5-34
- “Using Collateralized Mortgage Obligations (CMOs)” on page 5-40

More About

- “What Are Mortgage-Backed Securities?” on page 5-2

Pricing Mortgage Backed Securities Using the Black-Derman-Toy Model

This example illustrates how the Financial Toolbox™ and Financial Instruments Toolbox™ are used to price a level mortgage backed security using the BDT model.

Load the BDT Tree Stored in the Data File

```
load mbsexample.mat
```

Observe the Interest-Rate Tree

Visualize the interest rate evolution along the tree by looking at the output structure `BDTTree`. `BDTTree` returns an inverse discount tree, which you can convert into an interest-rate tree with the `cvtree` function.

```
BDTTreeR = cvtree(BDTTree);
```

Look at the upper branch and lower branch paths of the tree:

```
OldFormat = get(0, 'format');
format short

%Rate at root node:
RateRoot      = treepath(BDTTreeR.RateTree, 0)

RateRoot = 0.0399

%Rates along upper branch:
RatePathUp    = treepath(BDTTreeR.RateTree, [1 1 1 1 1])

RatePathUp = 6×1

    0.0399
    0.0397
    0.0391
    0.0383
    0.0373
    0.0360


%Rates along lower branch:
RatePathDown = treepath(BDTTreeR.RateTree, [2 2 2 2 2])

RatePathDown = 6×1

    0.0399
    0.0470
    0.0550
    0.0638
    0.0734
    0.0841
```

Compute the Price Tree for the Non-Prepayable Mortgage

Let's say that we have a three year $10000 level prepayable loan, with a mortgage interest rate of 4.64% semi-annually compounded.

```
MortgageAmount = 10000;
CouponRate = 0.0464;
Period = 2;
Settle='01-Jan-2007';
Maturity='01-Jan-2010';
Compounding = BDTTree.TimeSpec.Compounding;

format bank
```

Use the function `amortize` in the Financial Toolbox™ to calculate the mortgage payment of the loan (MP), the interest and principal components, and the outstanding principal balance.

```
NumPeriods = date2time(Settle,Maturity, Compounding)';

[Principal, InterestPayment, OutstandingBalance, MP] = amortize(CouponRate/Period, NumPeriods, M

% Display Principal, Interest and Outstanding balances
PrincipalAmount = Principal'
```

```
PrincipalAmount = 6×1

       1572.59
       1609.07
       1646.40
       1684.60
       1723.68
       1763.67
```

```
InterestPaymentAmount = InterestPayment'
```

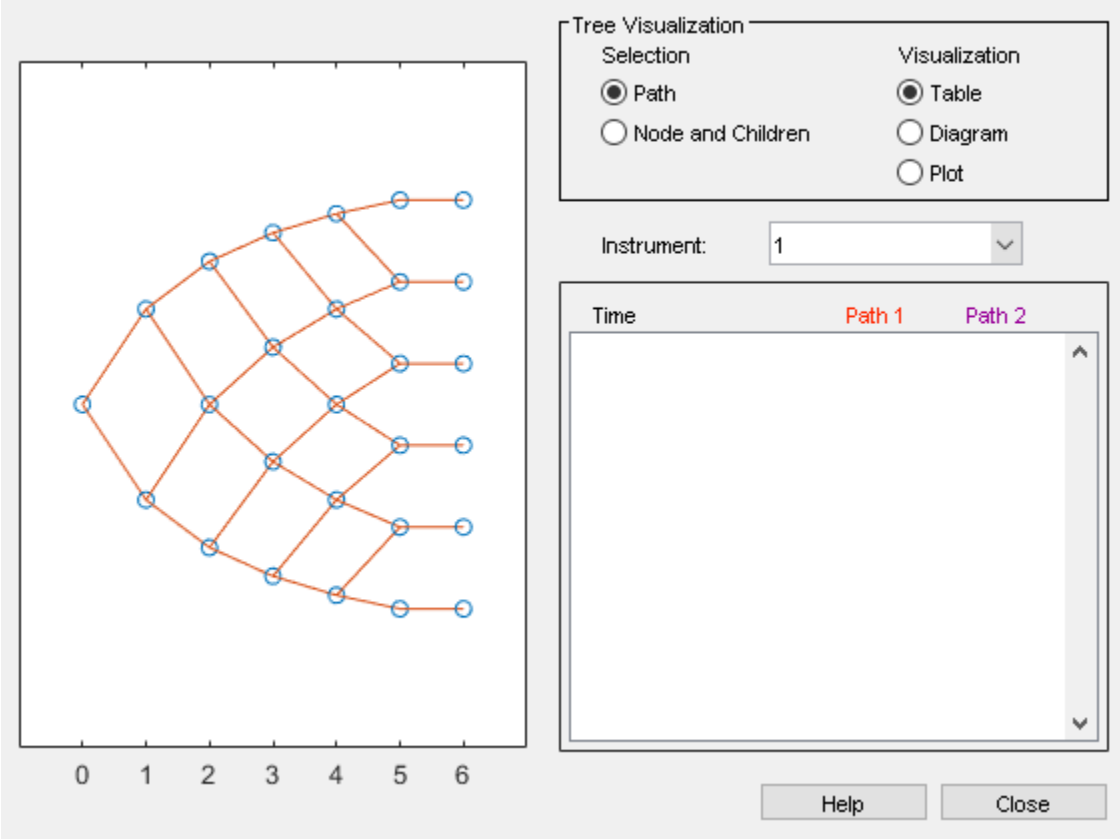
```
InterestPaymentAmount = 6×1

        232.00
        195.52
        158.19
        119.99
         80.91
         40.92
```

```
OutstandingBalanceAmount = OutstandingBalance'
```

```
OutstandingBalanceAmount = 6×1

       8427.41
       6818.34
       5171.94
       3487.35
       1763.67
          0.00
```

```
CFlowAmounts = MP*ones(1,NumPeriods);
% The CFlowDates are the same as the tree level dates
```

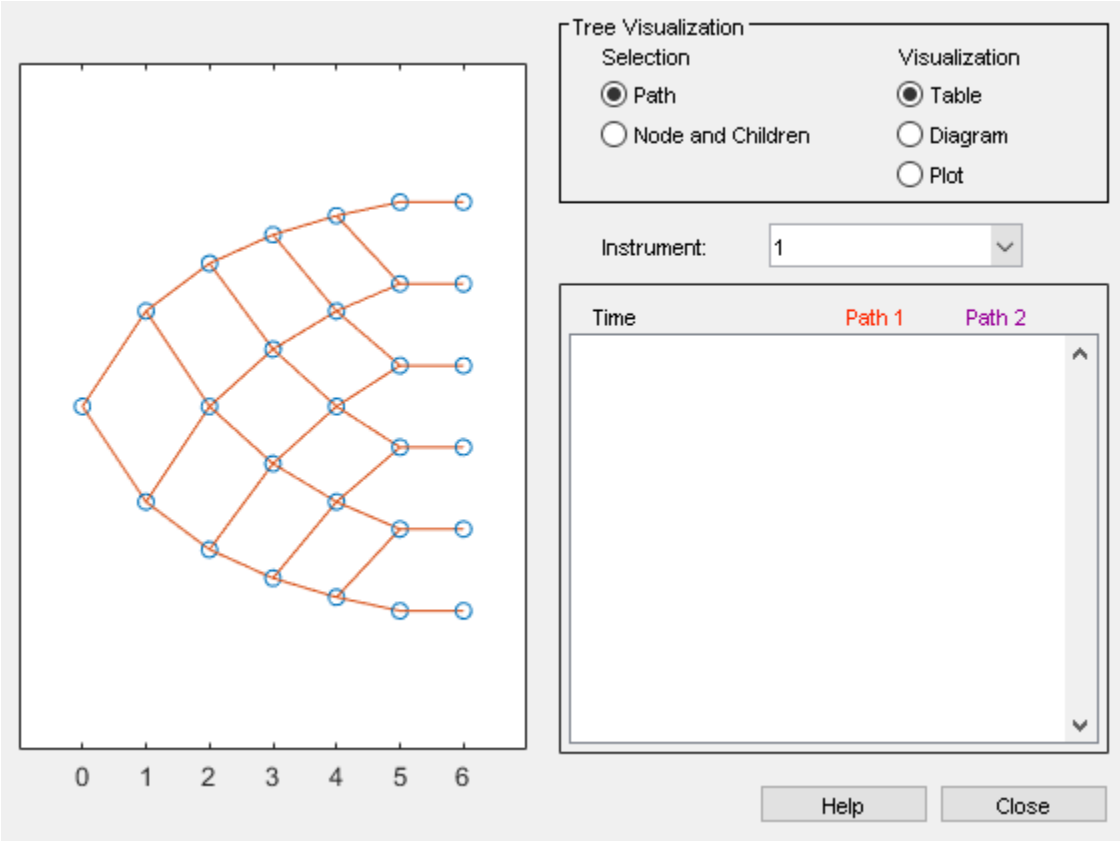
```
CFlowDates= {'01-Jul-2007' ,'01-Jan-2008' ,'01-Jul-2008' , '01-Jan-2009' , '01-Jul-2009' , '01-Ja

% Calculate the price of the non-prepayable mortgage
[PriceNonPrepayableMortgage, PriceTreeNonPrepayableMortgage] = cfbybdt(BDTTree, CFlowAmounts, CF
for iLevel = 2:length(PriceTreeNonPrepayableMortgage.PTree)
    PriceTreeNonPrepayableMortgage.PTree{iLevel}(:,:)= PriceTreeNonPrepayableMortgage.PTree{iLeve
end

% Look at the price of the mortgage today tObs = 0
PriceNonPrepayableMortgage

PriceNonPrepayableMortgage =
      10017.47


% The value of the non-prepayable mortgage is $10017.47. This value exceeds
% the $10000 amount borrowed since the homeowner received not only $10000, but
% also a prepayment option.

% Look at the value of the mortgage on the last date, right after the last
% mortgage payment, is zero:
PriceTreeNonPrepayableMortgage.PTree{end}

ans = 1×6

             0             0             0             0             0             0


% Visualize the price tree for the non-prepayable mortgage.
treeviewer(PriceTreeNonPrepayableMortgage)
```



Compute the Price Tree of the Prepayment Option

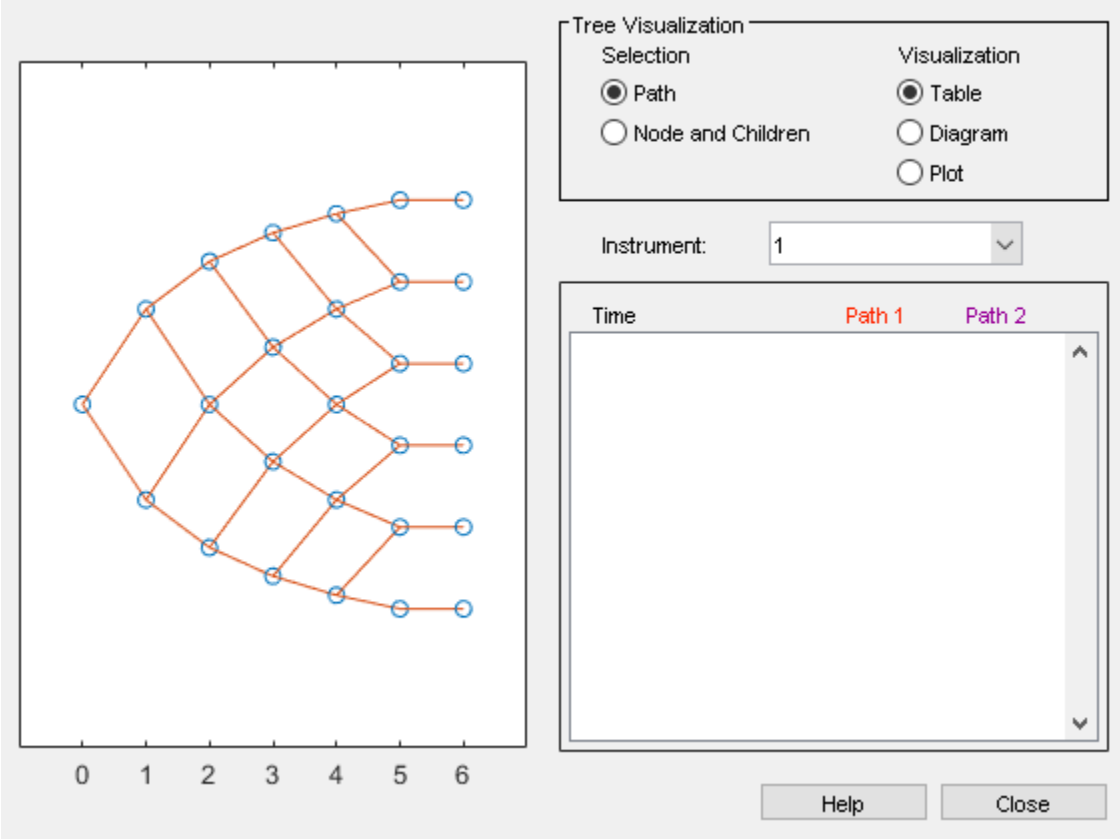
```
% The Prepayment option is like a call option on a bond.
%
% The exercise price or strike will be equal to the outstanding principal amount
% which has been calculated using the function amortize.

OptSpec = 'call';
Strike = [MortgageAmount OutstandingBalance];
ExerciseDates =[Settle CFlowDates];
AmericanOpt = 0;
Maturity = CFlowDates(end);

% Compute the price of the prepayment option:
[PricePrepaymentOption, PriceTreePrepaymentOption] = prepaymentbybdt(BDTTree, OptSpec, Strike, Ex
                            0, Settle, Maturity,[], [], [], ...
                           [], [], [],  [], 0, [], CFlowAmounts);


% Look at the price of the prepayment option today (tObs = 0)
PricePrepaymentOption

PricePrepaymentOption =
         17.47


% The value of the prepayment option is $17.47 as expected.
```

```
% Visualize the price tree for the prepayment option
treeviewer(PriceTreePrepaymentOption)
```



Calculate the Price Tree of the Prepayable Mortgage.

```
% Compute the price of the prepayable mortgage.

PricePrepayableMortgage = PriceNonPrepayableMortgage - PricePrepaymentOption;

PriceTreePrepayableMortgage = PriceTreeNonPrepayableMortgage;

for iLevel = 1:length(PriceTreeNonPrepayableMortgage.PTree)
    PriceTreePrepayableMortgage.PTree{iLevel}(:,:)= PriceTreeNonPrepayableMortgage.PTree{iLevel}
        PriceTreePrepaymentOption.PTree{iLevel}(:,:);
end

% Look at the price of the prepayable mortgage today (tObs = 0)
PricePrepayableMortgage

PricePrepayableMortgage =
      10000.00


% The value of the prepayable mortgage is $10000 as expected.

% Visualize the price and price tree for the prepayable mortgage
treeviewer(PriceTreePrepayableMortgage)
```




```
set(0, 'format', OldFormat);
```

See Also

`mbscfamounts` | `mbsconvp` | `mbsconvy` | `mbsdurp` | `mbsdury` | `mbsnoprepay` | `mbspassthrough` | `mbsprice` | `mbswal` | `mbsyield` | `mbsprice2speed` | `mbsyield2speed` | `psaspeed2default` | `psaspeed2rate` | `mbsoas2price` | `mbsoas2yield` | `mbsprice2oas` | `mbsyield2oas`

Related Examples

- “Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
- “Using Collateralized Mortgage Obligations (CMOs)” on page 5-40

More About

- “What Are Mortgage-Backed Securities?” on page 5-2

Using Collateralized Mortgage Obligations (CMOs)

What Are CMOs?

Financial Instruments Toolbox supports collateralized mortgage obligations (CMOs) to provide investors with a greater range of risk and return characteristics than mortgage-backed securities (MBS). In contrast to an MBS, which simply redirects principal and interest cash flows to investors on a pro rata basis, a CMO structures cash flows to different tranches, or slices, to create securities that are better tailored to specific investors.

For example, banks might be primarily concerned with extension risk, or the risk that their investment lengthens in time due to increasing interest rates, given that they typically have short-term deposits as liabilities. Insurance companies and pension funds might be concerned primarily with contraction risk, or the risk that their investment will pay off too soon, with liabilities that have much longer lives. A CMO structure addresses the interest-rate risk of extension or contraction with a blend of short-term and long-term CMO securities, called tranches.

See Also

`cmoseqcf` | `cmosched` | `cmoschedcf` | `mbscfamounts` | `mbspassthrough`

Related Examples

- “CMO Workflow” on page 5-47
- “Prepayment Risk” on page 5-41
- “Create PAC and Sequential CMO” on page 5-49
- “Fixed-Rate Mortgage Pool” on page 5-3

More About

- “What Are Mortgage-Backed Securities?” on page 5-2

Prepayment Risk

Prepayment risk is the risk that the term of the security varies according to differing rates of repayment of principal by borrowers (repayments from refinancings, sales, curtailments, or foreclosures). In a CMO, you can structure the principal (and associated coupon) stream from the underlying mortgage pool collateral to allocate prepayment risk. If principal is prepaid faster than expected (for example, if mortgage rates fall and borrowers refinance), then the overall term of the mortgage pool collateral shortens.

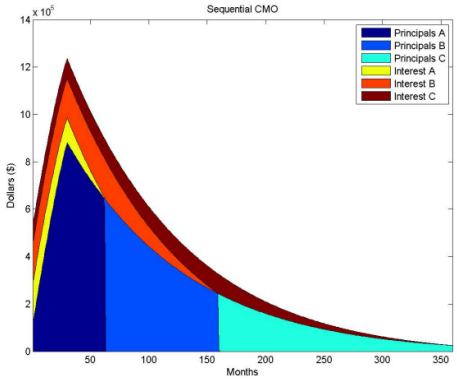
You cannot remove prepayment risk, but you can reallocate it among CMO tranches so that some tranches have some protection against this risk, and other tranches will absorb more of this risk. To facilitate this allocation of prepayment risk, CMOs are structured such that prepayments are allocated among tranches using a fixed set of rules. The most common schemes for prepayment tranching are:

- Sequential tranching, with or without, Z-bond tranching
- Schedule bond tranching
  - Planned amortization class (PAC) bonds
  - Target amortization class (TAC) bonds

Financial Instruments Toolbox supports these schemes for prepayment tranching for CMOs and tools for pricing and scheduling cash flows between the tranches, as well as analyzing the price and yield for CMOs. Financial Instruments Toolbox functionality for CMOs does not model credit risk. Therefore, this functionality is most appropriate for CMOs where credit risk is not an issue (for example, agency CMOs where the underlying mortgage pool collateral is insured for default by the agency Government-Sponsored Enterprises (GSEs), such as Fannie Mae and Freddie Mac).

Sequential Tranches Without a Z-Bond

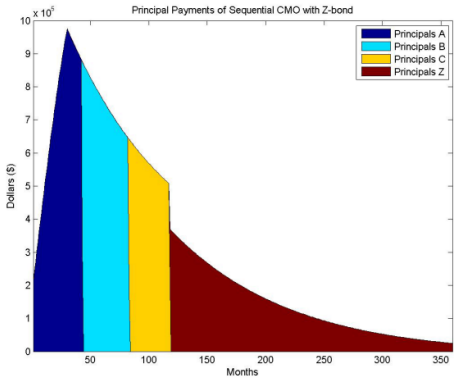
All available principal and interest payments go to the first sequential tranche, until its balance decrements to zero, then to the second, and so on. For example, consider the following example where all principal and interest from the underlying mortgage pool is repaid on tranche A first, then tranche B, then tranche C. Interest is paid on each tranche as long as the principal for the tranche has not been retired.



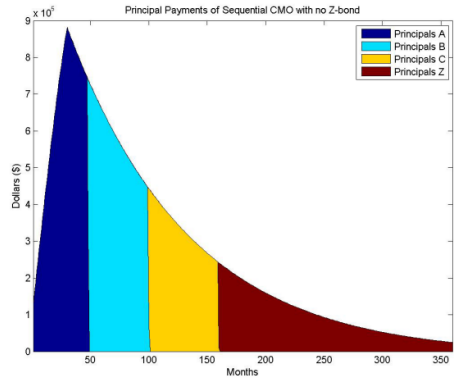
Sequential Tranches with a Z-Bond

The Z-bond, also called an accrual bond, is a type of interest and principal pay rule. The Z-bond tranche supports other sequential pay tranches by not receiving an interest payment. The interest payment that would have accrued to the Z-bond tranche pays off the principal of other bonds, and the principal of the Z-bond tranche increases. The Z-bond tranche starts receiving interest and principal payments only after the other tranches in the CMO have been fully paid. The Z-bond tranche is used in a sequential-pay structure to accelerate the principal repayments of the sequential-pay bonds.

A Z-bond differs from other CMO instruments because it is not tranching principal but interest. The Z-bond receives no cash flows until all other securities have been paid off. In the interim, the interest that is owed to the Z-bond is accrued to its principal. The following chart demonstrates the difference between a Z-bond and a normal sequential pay tranche. The C tranche pays off sooner with the Z-bond, because the interest cash flows to the Z-bond are being used to pay down the principal of the C tranche.



For comparison, the following graphic is the same sequential CMO with no Z-bond.



PAC Tranches

Planned amortization class (PAC) bonds help reduce the effects of prepayment risk. They are designed to produce more stable cash flows by redirecting prepayments from the underlying mortgage collateral to other classes (tranches) called companion or support classes. PAC bonds have a principal payment rate over a predetermined period of time. The PAC bond payment schedule is

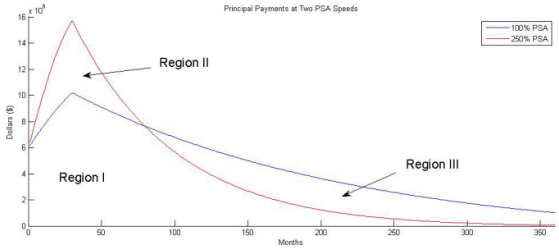
determined by two different prepayment rates, which together form a band (also called a collar). Early in the life of the CMO, the prepayment at the lower PSA yields a lower prepayment. Later in its life, the principal in the higher PSA declines enough that it yields a lower prepayment. The PAC tranche receives whichever rate is lower, so it will change prepayment at one PSA for the first part of its life, then switch to the other rate. The ability to stay on this schedule is maintained by a support bond, which absorbs excess prepayments, and receives fewer prepayments to prevent extension of average life.

However, the PAC is only protected from extension to the amount that prepayments are made on the underlying MBSs. If there is a sustained period of fast prepayments, then that might completely eliminate a PAC bond's outstanding support class. When the principal of the associated PAC bond is exhausted, the CMO is called a "busted PAC", or "busted collar". Alternatively, in times of slow prepayments, amortization of the support bonds is delayed if there is not enough principal for the currently paying PAC bond. This extends the average life of the class.

A PAC bond protects against both extension and contraction risk by:

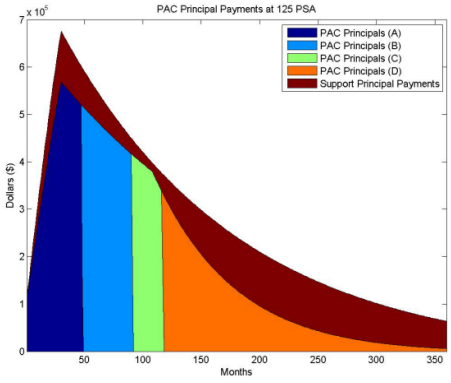
- Specifying a schedule of principal payments for the PAC bond
- Including support tranches that are allocated prepayments inside a specified prepayment band

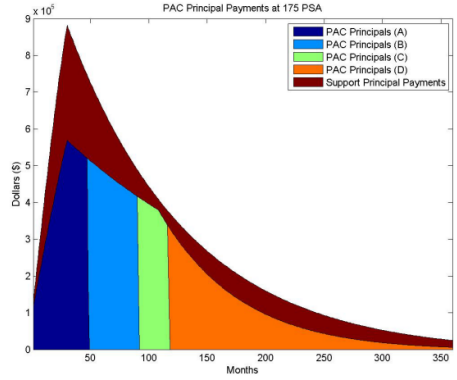
PAC bonds typically specify a band expressed using the PSA model. A PAC bond with a range of 100–250% has this principal schedule.

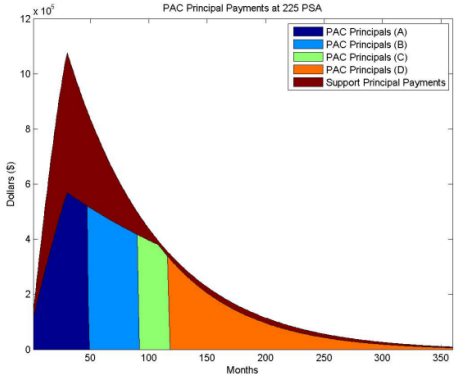


The principal repayment schedule is the minimum principal payment as Region 1 shows. This is the principal payment schedule as long as the actual prepayment stays within the prepayment band of 100–250% PSA.

For example, for different prepayment speeds of 125%, 175%, and 225% PSA, the actual principal payments are shown in the following graphs. At higher prepayment speeds, the support tranche is allocated principal earlier while the principal timing for the other tranches remains constant.



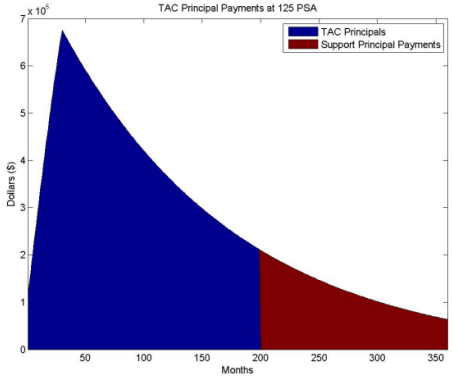


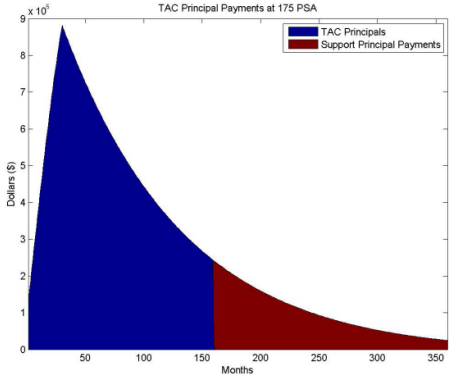


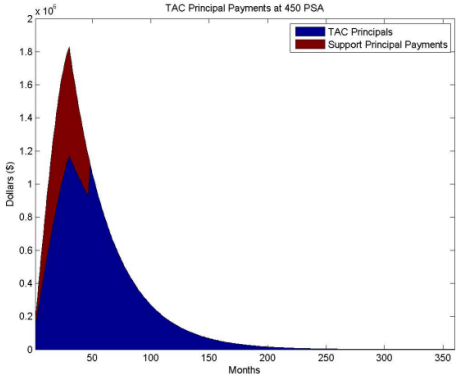
TAC Tranches

Target amortization class (TAC) bonds are similar to PAC bonds, but they do not provide protection against extension of average life. Create the schedule of principal payments by using just a single PSA. TAC bonds pay a “targeted” principal payment schedule at a single, constant prepayment speed. As long as the underlying mortgage collateral does not prepay at a rate slower than this speed, the TAC bond payment schedule is met. TAC bonds can protect against increasing prepayments and early retirement of the TAC bond investment. If the principal cash flow from the mortgage collateral exceeds the TAC schedule, the excess is allocated to TAC companion (support) classes. Alternatively, if prepayments fall below the speed necessary to maintain the TAC schedule, the weighted average life of the TAC is extended. The TAC bond does not protect against low prepayment rates.

For example, here is a TAC structure rated for 125%, 175%, and 450% PSA.







For prepayments below 175% PSA, the TAC bond extends like a normal sequential pay CMO. TAC bonds are appealing because they offer higher yields than comparable PAC bonds. The unaddressed risk from low prepayment rates generally does not concern investors as much as risk from high prepayment rates.

See Also

`cmoseqcf` | `cmosched` | `cmoschedcf` | `mbscfamounts` | `mbspassthrough`

Related Examples

- “CMO Workflow” on page 5-47
- “Prepayment Risk” on page 5-41
- “Create PAC and Sequential CMO” on page 5-49
- “Fixed-Rate Mortgage Pool” on page 5-3

More About

- “What Are CMOs?” on page 5-40
- “What Are Mortgage-Backed Securities?” on page 5-2

CMO Workflow

In general, the CMO workflow is:

1. Calculate underlying mortgage cash flows.
2. Define CMO tranches
3. If using a PAC or TAC CMO, calculate the principal schedule.
4. Calculate cash flows for each tranche.
5. Analyze the CMO by computing price, yield, spread of CMO cash flows.

Calculate Underlying Mortgage Cash Flows

Underlying mortgage pool pass-through cash flows are calculated by the existing function `mbspassthrough`. The CMO cash flow functions require the principal payments (including prepayments) calculated from existing functions `mbspassthrough` or `mbscfamounts`.

```
principal = 10000000;
coupon = 0.06;
terms = 360;
psa = 150;

[principal_balance, monthly_payments, sched_principal_payments,...
interest_payments, prepayments] = mbspassthrough(principal,...
coupon, terms, terms, psa, []);

principal_payments = sched_principal_payments.' + prepayments.';
```

After determining principal payments for the underlying mortgage collateral, you can generate cash flows for a sequential CMO, with or without a Z-bond, by using `cmoseqcf`. For a PAC or TAC CMO, the cash flows are generated using `cmoschedcf`

Define CMO Tranches

Define CMO tranche; for example, define a CMO with two tranches:

```
TranchePrincipals = [500000; 500000];
TrancheCoupons = [0.06; 0.06];
```

If Using a PAC or TAC CMO, Calculate Principal Schedule

Calculate the PAC/TAC principal balance schedule based on a band of PSA speeds. For scheduled CMOs (PAC/TAC), the CMO cash flow functions additionally take in the principal balance schedule calculated by the CMO schedule function `cmosched`.

```
terms = 360;
coupon = 0.06;
principal = 10000000;
speed = [100 300];
[balanceSchedule, initialBalance] = cmosched(principal, coupon,...
terms, terms, speed, TranchePrincipals(1));
```

Calculate Cash Flows for Each Tranche

You can reuse the output from the cash flow generation functions to further divide the cash flows into tranches. For example, the output from `cmoschedcf` for a PAC tranche can be divided into sequential

tranches by passing the principal cash flows of the PAC tranche into the `cmoschedcf` function. The outputs of the CMO cash flow functions are the principal and interest cash flows, and the principal balance.

```
[principal_balances, principal_cashflows, interest_cashflows] = cmoschedcf(principal_payments,...
TranchePrincipals, TrancheCoupons, balanceSchedule);
```

Analyze CMO by Computing Price, Yield, and Spread of CMO Cash Flows

The outputs from the CMO functions (`cmoseqcf` and `cmoschedcf`) are cash flows. The functions used to analyze a CMO are based on these cash flows. To that end, you can use `cfbyzero`, `cfspread`, `cfyield`, and `cfprice` to compute prices, yield, and spreads for the CMO cash flows. In addition, using the following, you can calculate a weighted average life (WAL) for each tranche in the CMO:

$$WAL = \sum_{i=1}^{n} \frac{P_i}{P} t_i$$

where:

P is the total principal.

P_i is the principal repayment of the coupon i.

$\frac{P_i}{P}$ is the fraction of the principal repaid in coupon i.

t_i is the time in years from the start to coupon i.

See Also

`cmoseqcf` | `cmosched` | `cmoschedcf` | `mbscfamounts` | `mbspassthrough`

Related Examples

- “Using Collateralized Mortgage Obligations (CMOs)” on page 5-40
- “Create PAC and Sequential CMO” on page 5-49
- “Fixed-Rate Mortgage Pool” on page 5-3

More About

- “What Are Mortgage-Backed Securities?” on page 5-2

Create PAC and Sequential CMO

This example shows how to use an underlying mortgage-backed security (MBS) pool for a 30-year fixed-rate mortgage of 6% to define a PAC bond, and then define a sequential CMO from the PAC bond. Analyze the CMO by comparing the CMO spread to a zero-rate curve for a 30-year Treasury bond and then calculate the weighted-average life (WAL) for the PAC bond.

Step 1. Define the underlying mortgage pool.

```
principal = 100000000;
grossrate = 0.06;
coupon = 0.05;
originalTerm = 360;
termRemaining = 360;
speed = 100;
delay = 14;

Settle      = datenum('1-Jan-2011');
IssueDate   = datenum('1-Jan-2011');
Maturity    = addtodate(IssueDate, 360, 'month');
```

Step 2. Calculate underlying pool cash flow.

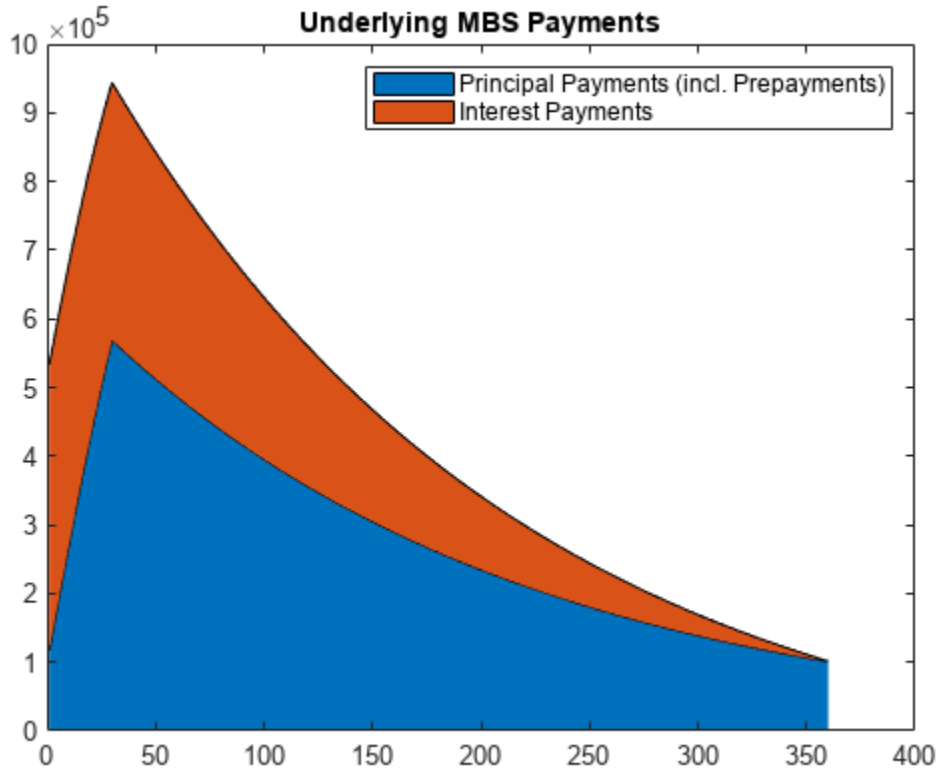
```
[CFlowAmounts, CFlowDates, ~, ~, ~, UnitPrincipal, UnitInterest, ...
UnitPrepayment] = mbscfamounts(Settle, Maturity, IssueDate, grossrate, ...
coupon, delay, speed, []);
```

Step 3. Calculate prepayments.

```
principalPayments = UnitPrincipal * principal;
netInterest = UnitInterest * principal;
prepayments = UnitPrepayment * principal;
dates = CFlowDates' + delay;
```

Step 4. Generate a plot for underlying MBS payments.

```
area([principalPayments'+prepayments', netInterest'])
title('Underlying MBS Payments');
legend('Principal Payments (incl. Prepayments)', 'Interest Payments')
```

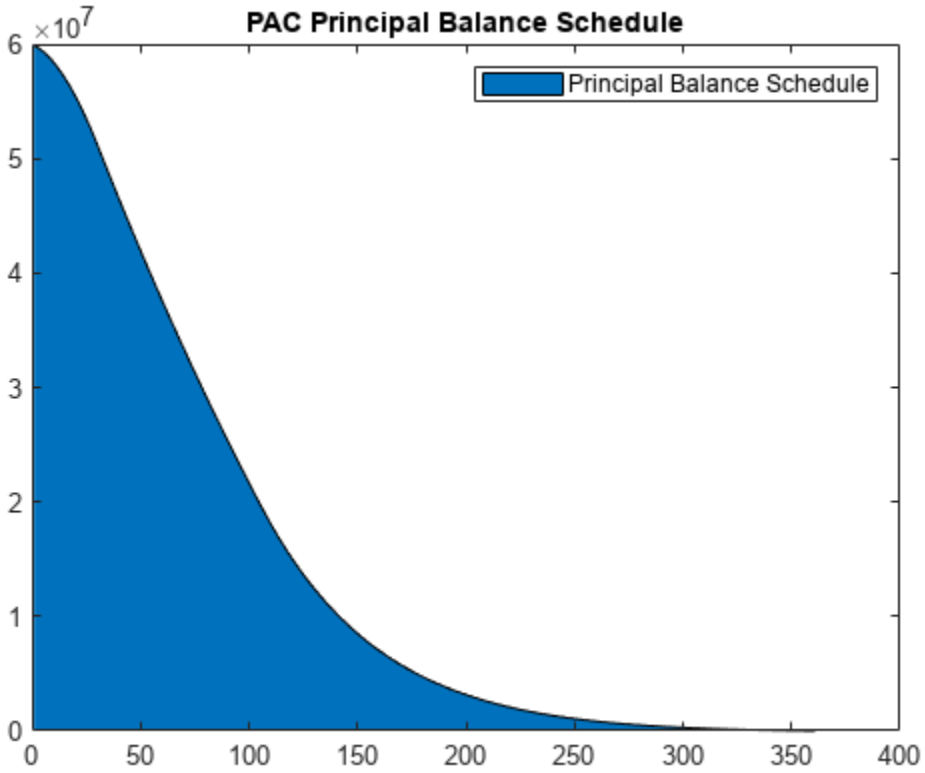


Step 5. Calculate the PAC schedule.

```
pacSpeed = [80 300];
[balanceSchedule, pacInitBalance] = ...
cmosched(principal, grossrate, originalTerm, termRemaining, ...
pacSpeed, []);
```

Step 6. Generate a plot for the PAC principal balance schedule.

```
figure;
area([pacInitBalance'; balanceSchedule'])
title('PAC Principal Balance Schedule');
legend('Principal Balance Schedule');
```



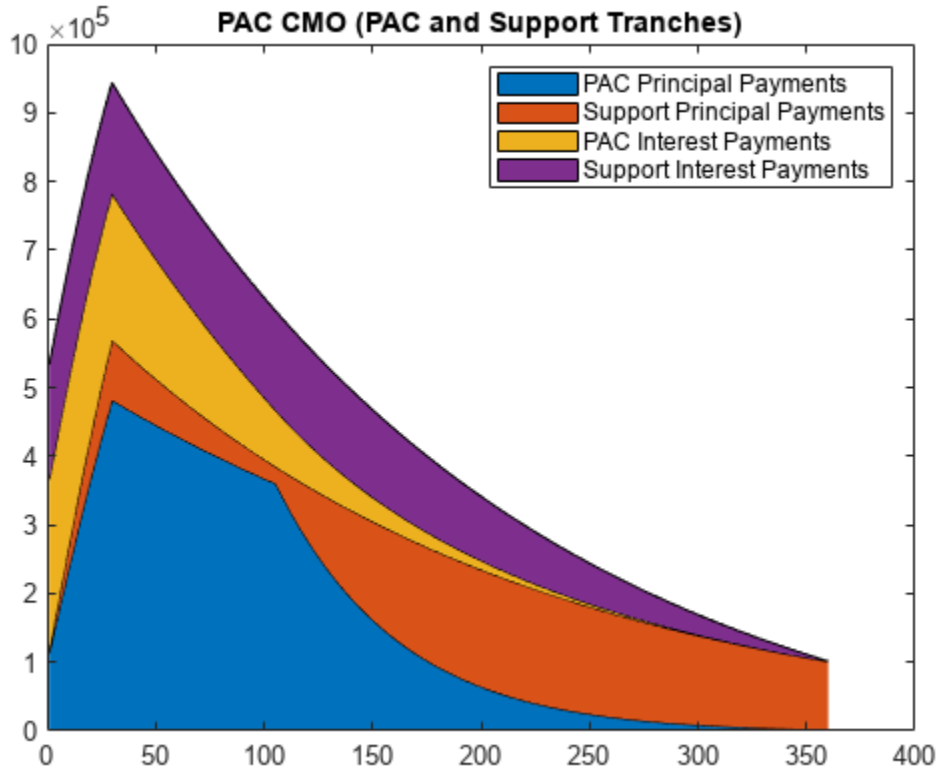
Step 7. Calculate PAC cash flow.

```
pacTranchePrincipals = [pacInitBalance; principal-pacInitBalance];
pacTrancheCoupons = [0.05; 0.05];
[pacBalances, pacPrincipals, pacInterests] = ...
cmoschedcf(principalPayments+prepayments, ...
pacTranchePrincipals, pacTrancheCoupons, balanceSchedule);
```

Step 8. Generate a plot for the PAC CMO tranches.

Generate a plot for the PAC CMO tranches:

```
figure;
area([pacPrincipals' pacInterests']);
title('PAC CMO (PAC and Support Tranches)');
legend('PAC Principal Payments', 'Support Principal Payments', ...
'PAC Interest Payments', 'Support Interest Payments');
```



Step 9. Create sequential CMO from the PAC bond.

CMO tranches, A, B, C, and D

```
seqTranchePrincipals = ...
[20000000; 20000000; 10000000; pacInitBalance-50000000];
seqTrancheCoupons = [0.05; 0.05; 0.05; 0.05];
```

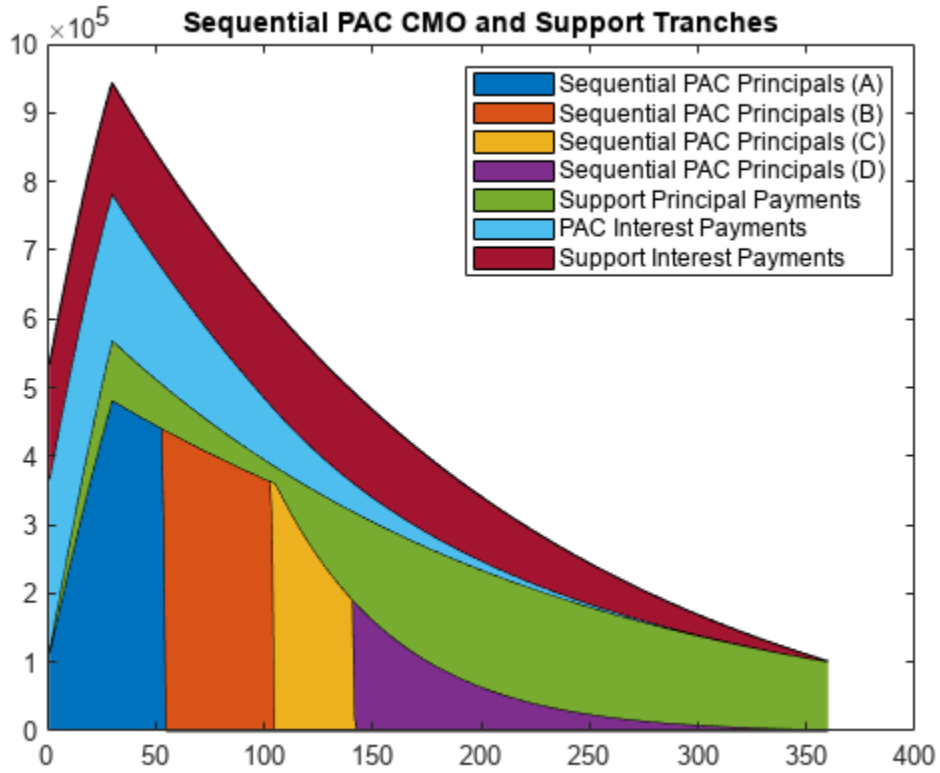
Step 10. Calculate cash flows for each tranche.

```
[seqBalances, seqPrincipals, seqInterests] = ...
cmoseqcf(pacPrincipals(1, :), seqTranchePrincipals, ...
seqTrancheCoupons, false);
```

Step 11. Generate a plot for the sequential PAC CMO.

Generate a plot for the sequential PAC CMO:

```
figure
area([seqPrincipals' pacPrincipals(2, :)' pacInterests']);
title('Sequential PAC CMO and Support Tranches');
legend('Sequential PAC Principals (A)', 'Sequential PAC Principals (B)', ...
'Sequential PAC Principals (C)', 'Sequential PAC Principals (D)', ...
'Support Principal Payments', 'PAC Interest Payments', ...
'Support Interest Payments');
```



Step 12. Create the discount curve.

```
CurveSettle = datenum('1-Jan-2011');
ZeroRates = [0.01 0.03 0.10 0.19 0.45 0.81 1.76 2.50 3.18 4.09 4.38]'/100;
CurveTimes = [1/12 3/12 6/12 1 2 3 5 7 10 20 30]';
CurveDates = daysadd(CurveSettle, 360 * CurveTimes, 1);
zeroCurve = intenvset('Rates', ZeroRates, 'StartDates', CurveSettle, ...
'EndDates', CurveDates);
```

Step 13. Price the CMO cash flows.

The cash flow for the sequential PAC principal A tranche is calculated using the cash flow functions `cfbyzero`, `cfyield`, `cfprice`, and `cfspread`.

```
cflows = seqPrincipals(1, :)+seqInterests(1, :);
cfdates = dates(2:end)';
price1 = cfbyzero(zeroCurve, cflows, cfdates, Settle, 4)
```

```
price1 = 2.2109e+07
```

```
yield = cfyield(cflows, cfdates, price1, Settle, 'Basis', 4)
```

```
yield = 0.0090
```

```
price2 = cfprice(cflows, cfdates, yield, Settle, 'Basis', 4)
```

```
price2 = 2.2109e+07
```

```
spread = cfspread(zeroCurve, price2, cflows, cfdates, Settle, 'Basis', 4)
```

```
spread = 5.5084e-12
```

```
WAL = sum(cflows .* yearfrac(Settle, cfdates, 4)) / sum(cflows)
```

```
WAL = 2.5408
```

The weighted average life (WAL) for the sequential PAC principal A tranche is `2.54` years.

See Also

`cmoseqcf` | `cmosched` | `cmoschedcf` | `mbscfamounts` | `cfbyzero` | `cfyield` | `cfprice` | `cfspread` | `cfbyzero`

Related Examples

- “Fixed-Rate Mortgage Pool” on page 5-3

More About

- “Using Collateralized Mortgage Obligations (CMOs)” on page 5-40

Debt Instruments

Agency Option-Adjusted Spreads

Often bonds are issued with embedded options, which then makes standard price/yield or spread measures irrelevant. For example, a municipality concerned about the chance that interest rates may fall in the future might issue bonds with a provision that allows the bond to be repaid before the bond's maturity. This is a call option on the bond and must be incorporated into the valuation of the bond. Option-adjusted spread (OAS), which adjusts a bond spread for the value of the option, is the standard measure for valuing bonds with embedded options. Financial Instruments Toolbox software supports computing option-adjusted spreads for bonds with single embedded options using the agency model.

The Securities Industry and Financial Markets Association (SIFMA) has a simplified approach to compute OAS for agency issues (Government Sponsored Entities like Fannie Mae and Freddie Mac) termed "Agency OAS". In this approach, the bond has only one call date (European call) and uses Black's model (a variation on Black Scholes, `http://en.wikipedia.org/wiki/Black_model`) to value the bond option. The price of the bond is computed as follows:

$$\texttt{Price}_{\text{Callable}} = \texttt{Price}_{\text{NonCallable}} - \texttt{Price}_{\text{Option}}$$

where

$\texttt{Price}_{\text{Callable}}$ is the price of the callable bond.

$\texttt{Price}_{\text{NonCallable}}$ is the price of the noncallable bond, that is, price of the bond using `bndspread`.

$\texttt{Price}_{\text{Option}}$ is the price of the option, that is, price of the option using Black's model.

The Agency OAS is the spread, when used in the previous formula, yields the market price. Financial Instruments Toolbox software supports these functions:

Agency OAS

Agency OAS Functions	Purpose
`agencyoas`	Compute the OAS of the callable bond using the Agency OAS model.
`agencyprice`	Price the callable bond OAS using Agency using the OAS model.

Computing the Agency OAS for Bonds

To compute the Agency OAS using `agencyoas`, you must provide the zero curve as the input `ZeroData`. You can specify the zero curve in any intervals and with any compounding method. You can do this using Financial Toolbox™ functions `zbtprice` and `zbtyield`. Or, you can use `IRDataCurve` to construct an `IRDataCurve` object, and then use the `getZeroRates` to convert to dates and data for use in the `ZeroData` input.

After creating the `ZeroData` input for `agencyoas`, you can then:

1. Assign parameters for `CouponRate`, `Settle`, `Maturity`, `Vol`, `CallDate`, and `Price`.
2. Compute the option-adjusted spread using `agencyoas` to derive the `OAS` output.

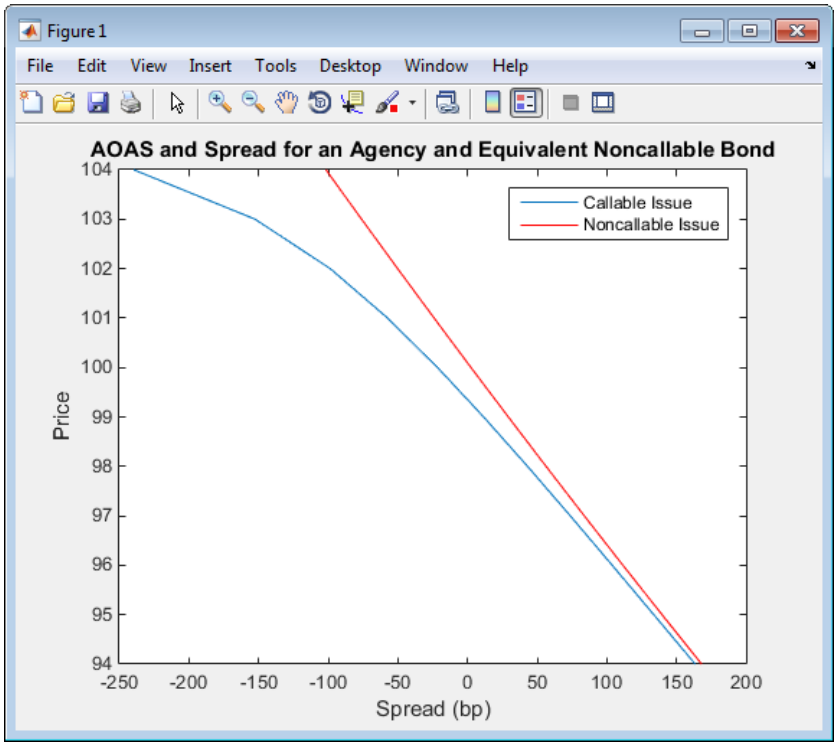
   If you have the Agency OAS for the callable bond, you can use the `OAS` value as an input to `agencyprice` to determine the price for a callable bond.

In the following example, the Agency OAS is computed using `agencyoas` for a range of bond prices and the spread of an identically priced noncallable bond is calculated using `bndspread`.

```
%% Data
% Bond data -- note that there is only 1 call date
Settle = datetime(2010,1,20);
Maturity = datetime(2013,12,30);
Coupon = .022;
Vol = .5117;
CallDate = datetime(2010,12,30);
Period = 2;
Basis = 1;
Face = 100;

% Zero Curve data
ZeroTime = [.25 .5 1 2 3 4 5 7 10 20 30]';
ZeroDates = daysadd(Settle,360*ZeroTime,1);
ZeroRates = [.0008 .0017 .0045 .0102 .0169 .0224 .0274 .0347 .0414 .0530 .0740]';
ZeroData = [ZeroDates ZeroRates];
CurveCompounding = 2;
CurveBasis = 1;

Price = 94:104;
OAS = agencyoas(ZeroData, Price', Coupon, Settle,Maturity, Vol, CallDate,'Basis',Basis)
Spread = bndspread(ZeroData, Price', Coupon, Settle, Maturity)
plot(OAS,Price)
hold on
plot(Spread,Price,'r')
xlabel('Spread (bp)')
ylabel('Price')
title('AOAS and Spread for an Agency and Equivalent Noncallable Bond')
legend({'Callable Issue','Noncallable Issue'})
```

```
OAS =

  163.4942
  133.7306
  103.8735
   73.7505
   43.1094
   11.5608
  -21.5412
  -57.3869
  -98.5675
 -152.5226
 -239.6462


Spread =

  168.1412
  139.7047
  111.6123
   83.8561
   56.4286
   29.3227
    2.5314
  -23.9523
  -50.1348
  -76.0226
 -101.6218
```

The following plot demonstrates as the price increases, the value of the embedded option in the Agency issue increases, and the value of the issue itself does not increase as much as it would for a noncallable bond, illustrating the negative convexity of this issue:



See Also

`agencyoas` | `agencyprice`

Related Examples

- “Using Zero-Coupon Bonds” on page 6-5
- “Stepped-Coupon Bonds” on page 6-8
- “Term Structure Calculations” on page 6-11

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3

Using Zero-Coupon Bonds

In this section...

Introduction

A zero-coupon bond is a corporate, Treasury, or municipal debt instrument that pays no periodic interest. Typically, the bond is redeemed at maturity for its full face value. It is a security issued at a discount from its face value, or it may be a coupon bond stripped of its coupons and repackaged as a zero-coupon bond.

Financial Instruments Toolbox software provides functions for valuing zero-coupon debt instruments. These functions supplement existing coupon bond functions such as `bndprice` and `bndyield` that are available in Financial Toolbox software.

Measuring Zero-Coupon Bond Function Quality

Zero-coupon function quality is measured by how consistent the results are with coupon-bearing bonds. Because the zero coupon's yield is bond-equivalent, comparisons with coupon-bearing bonds are possible.

In the textbook case, where time (t) is measured continuously and the rate (r) is continuously compounded, the value of a zero bond is the principal multiplied by e^{-rt}. In reality, the rate quoted is continuous and the basis can be variable, requiring a more consistent approach to meet the stricter demands of accurate pricing.

The following two examples

- “Pricing Treasury Notes” on page 6-5
- “Pricing Corporate Bonds” on page 6-7

show how the zero functions are consistent with supported coupon bond functions.

Pricing Treasury Notes

A Treasury note can be considered to be a package of zeros. The toolbox functions that price zeros require a coupon bond equivalent yield. That yield can originate from any type of coupon paying bond, with any periodic payment, or any accrual basis. The next example shows the use of the toolbox to price a Treasury note and compares the calculated price with the actual price quotation for that day.

```
Settle = datetime(2003,2,3);
MaturityCpn = datetime(2009,5,15);
Period = 2;
Basis = 0;
```

```
% Quoted yield.
QYield = 0.03342;

% Quoted price.
QPriceACT = 112.127;

CouponRate = 0.055;
```

Extract the cash flow and compute price from the sum of zeros discounted.

```
[CFlows, CDates] = cfamounts(CouponRate, Settle, MaturityCpn, ...
Period, Basis);
MaturityofZeros = CDates;
```

Compute the price of the coupon bond identically as a collection of zeros by multiplying the discount factors to the corresponding cash flows.

```
PriceofZeros = CFlows * zeroprice(QYield, Settle, ...
MaturityofZeros, Period, Basis)/100;
```

The following table shows the intermediate calculations.

Cash Flows	Discount Factors		Discounted Cash Flows
-1.2155	1.0000		-1.2155
2.7500	0.9908		2.7246
2.7500	0.9745		2.6799
2.7500	0.9585		2.6359
2.7500	0.9427		2.5925
2.7500	0.9272		2.5499
2.7500	0.9120		2.5080
2.7500	0.8970		2.4668
2.7500	0.8823		2.4263
2.7500	0.8678		2.3864
2.7500	0.8535		2.3472
2.7500	0.8395		2.3086
2.7500	0.8257		2.2706
102.7500	0.8121		83.4451
		Total	112.1263

Compare the quoted price and the calculated price based on zeros.

```
[QPriceACT PriceofZeros]

ans =

112.1270   112.1263
```

This example shows that `zeroprice` can satisfactorily price a Treasury note, a semiannual actual/actual basis bond, as if it were a composed of a series of zero-coupon bonds.

Pricing Corporate Bonds

You can similarly price a corporate bond, for which there is no corresponding zero-coupon bond, as opposed to a Treasury note, for which corresponding zeros exist. You can create a synthetic zero-coupon bond and arrive at the quoted coupon-bond price when you later sum the zeros.

```
Settle = datetime(2003,2,5);
MaturityCpn = datetime(2009,1,14);
Period = 2;
Basis = 1;
% Quoted yield.
QYield = 0.05974;
% Quoted price.
QPrice30 = 99.382;
CouponRate = 0.05850;
```

Extract cash flow and compute price from the sum of zeros.

```
[CFlows, CDates] = cfamounts(CouponRate, Settle, MaturityCpn, ...
Period, Basis);

Maturity = CDates;
```

Compute the price of the coupon bond identically as a collection of zeros by multiplying the discount factors to the corresponding cash flows.

```
Price30 = CFlows * zeroprice(QYield, Settle, Maturity, Period, ...
Basis)/100;
```

Compare quoted price and calculated price based on zeros.

```
[QPrice30 Price30]

ans =

99.3820   99.3828
```

As a test of fidelity, intentionally giving the wrong basis, say actual/actual (`Basis = 0`) instead of 30/360, gives a price of 99.3972. Such a systematic error, if recurring in a more complex pricing routine, quickly adds up to large inaccuracies.

In summary, the zero functions in MATLAB software facilitate extraction of present value from virtually any fixed-coupon instrument, up to any period in time.

See Also

`bndprice` | `bndyield`

Related Examples

- “Agency Option-Adjusted Spreads” on page 6-2
- “Stepped-Coupon Bonds” on page 6-8
- “Term Structure Calculations” on page 6-11

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3

Stepped-Coupon Bonds

In this section...

Introduction

A stepped-coupon bond has a fixed schedule of changing coupon amounts. Like fixed coupon bonds, stepped-coupon bonds could have different periodic payments and accrual bases.

The functions `stepcpnprice` and `stepcpnyield` compute prices and yields of such bonds. An accompanying function `stepcpncfamounts` produces the cash flow schedules pertaining to these bonds.

Cash Flows from Stepped-Coupon Bonds

Consider a bond that has a schedule of two coupons. Suppose that the bond starts out with a 2% coupon that steps up to 4% in 2 years and onward to maturity. Assume that the issue and settlement dates are both March 15, 2003. The bond has a 5-year maturity. Use `stepcpncfamounts` to generate the cash flow schedule and times.

```
Settle      = datenum('15-Mar-2003');
Maturity    = datenum('15-Mar-2008');
ConvDates   = [datenum('15-Mar-2005')];
CouponRates = [0.02, 0.04];

[CFlows, CDates, CTimes] = stepcpncfamounts(Settle, Maturity, ...
ConvDates, CouponRates)

CFlows =

     0     1     1     1     1     2     2     2     2     2   102


CDates =

      731655      731839      732021      732205      732386      732570      732751      732935      733116      733300


CTimes =

     0     1     2     3     4     5     6     7     8     9    10
```

Notably, `ConvDates` has one less element than `CouponRates` because MATLAB software assumes that the first element of `CouponRates` indicates the coupon schedule between `Settle` (March 15, 2003) and the first element of `ConvDates` (March 15, 2005), shown diagrammatically below.

	Pay 2% from March 15, 2003		Pay 4% from March 15, 2003
Effective 2% on March 15, 2003		Effective 4% on March 15, 2005	

Coupon Dates	Semiannual Coupon Payment
15-Mar-03	0
15-Sep-03	1
15-Mar-04	1
15-Sep-04	1
15-Mar-05	1
15-Sep-05	2
15-Mar-06	2
15-Sep-06	2
15-Mar-07	2
15-Sep-07	2
15-Mar-08	102

The payment on March 15, 2005 is still a 2% coupon. Payment of the 4% coupon starts with the next payment, September 15, 2005. March 15, 2005 is the end of first coupon schedule, not to be confused with the beginning of the second.

In summary, MATLAB takes user input as the end dates of coupon schedules and computes the next coupon dates automatically.

The payment due on settlement (zero in this case) represents the accrued interest due on that day. It is negative if such amount is nonzero. Comparison with `cfamounts` in Financial Toolbox shows that the two functions operate identically.

Price and Yield of Stepped-Coupon Bonds

The toolbox provides two basic analytical functions to compute price and yield for stepped-coupon bonds. Using the above bond as an example, you can compute the price when the yield is known.

You can estimate the yield to maturity as a number-of-year weighted average of coupon rates. For this bond, the estimated yield is:

$$\frac{(2 \times 2) + (4 \times 3)}{5}$$

.

or 3.33%. While definitely not exact (due to nonlinear relation of price and yield), this estimate suggests close to par valuation and serves as a quick first check on the function.

```
Yield = 0.0333;

[Price, AccruedInterest] = stepcpnprice(Yield, Settle, ...
Maturity, ConvDates, CouponRates)
```

```
Price =

   99.2237


AccruedInterest =
```

```
     0
```

The price returned is 99.2237 (per $100 notional), and the accrued interest is zero, consistent with our earlier assertions.

To validate that there is consistency among the stepped-coupon functions, you can use the above price and see if indeed it implies a 3.33% yield by using `stepcpnyield`.

```
YTM = stepcpnyield(Price, Settle, Maturity, ConvDates, ...
CouponRates)
```

```
YTM =

    0.0333
```

See Also

`stepcpnprice` | `stepcpnyield` | `stepcpncfamounts`

Related Examples

- “Agency Option-Adjusted Spreads” on page 6-2
- “Using Zero-Coupon Bonds” on page 6-5
- “Term Structure Calculations” on page 6-11

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3

Term Structure Calculations

In this section...

Introduction

So far, a more formal definition of "yield" and its application has not been developed. In many situations when cash flow is available, discounting factors to the cash flows may not be immediately apparent. In other cases, what is relevant is often a *spread*, the difference between curves (also known as the term structure of spread).

All these calculations require one main ingredient, the Treasury spot, par-yield, or forward curve. Typically, the generation of these curves starts with a series of on-the-run and selected off-the-run issues as inputs.

MATLAB software uses these bonds to find spot rates one at a time, from the shortest maturity onwards, using bootstrap techniques. All cash flows are used to construct the spot curve, and rates between maturities (for these coupons) are interpolated linearly.

Computing Spot and Forward Curves

For an illustration of how this works, observe the use of `zbtyield` (or equivalently `zbtprice`) on a portfolio of six Treasury bills and bonds.

Bills	Maturity Date	Current Yield
3 month	4/17/03	1.15
6 month	7/17/03	1.18

Notes/Bonds	Coupon	Maturity Date	Current Yield
2 year	1.750	12/31/04	1.68
5 year	3.000	11/15/07	2.97
10 year	4.000	11/15/12	4.01
30 year	5.375	2/15/31	4.92

You can specify prices or yields to the bonds above to infer the spot curve. The function `zbtyield` accepts yields (bond-equivalent yield, to be exact).

To proceed, first assemble the above table into a variable called `Bonds`. The first column contains maturities, the second contains coupons, and the third contains notionals or face values of the bonds. (Note that bills have zero coupons.)

```
Bonds = [datenum('04/17/2003')    0        100;
         datenum('07/17/2003')    0        100;
         datenum('12/31/2004')    0.0175   100;
         datenum('11/15/2007')    0.03     100;
```

```
         datenum('11/15/2012')    0.04     100;
         datenum('02/15/2031')    0.05375  100];
```

Then specify the corresponding yields.

```
Yields  = [0.0115;
           0.0118;
           0.0168;
           0.0297;
           0.0401;
           0.0492];
```

You are now ready to compute the spot curve for each of these six maturities. The spot curve is based on a settlement date of January 17, 2003.

```
Settle = datenum('17-Jan-2003');
[ZeroRates, CurveDates] = zbtyield(Bonds, Yields, Settle)
```

```
ZeroRates =

    0.0115
    0.0118
    0.0168
    0.0302
    0.0418
    0.0550


CurveDates =

      731688
      731779
      732312
      733361
      735188
      741854
```

This gets you the Treasury spot curve for the day.

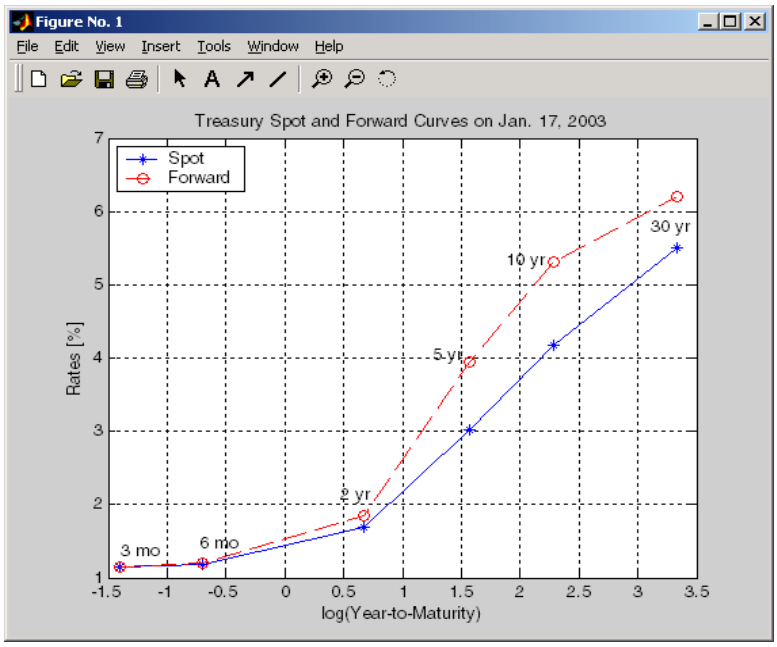
You can compute the forward curve from this spot curve with `zero2fwd`.

```
[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates, ...
Settle)
```

```
ForwardRates =

    0.0115
    0.0121
    0.0185
    0.0394
    0.0530
    0.0621


CurveDates =

      731688
      731779
      732312
```

```
733361
735188
741854
```

Here the notion of forward rates refers to rates between the maturity dates shown above, not to a certain period (forward 3-month rates, for example).



Computing Spreads

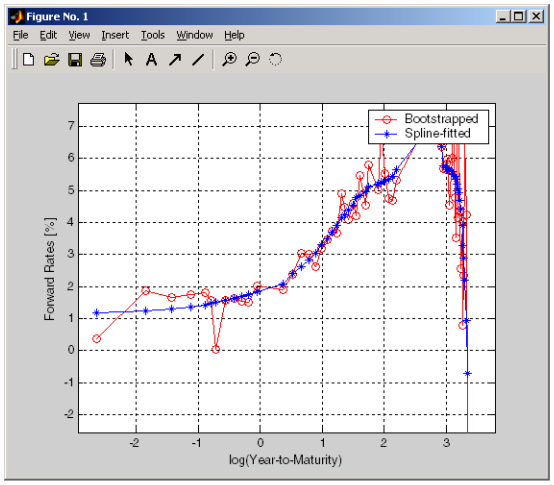
Calculating the spread between specific, fixed forward periods (such as the Treasury-Eurodollar spread) requires an extra step. Interpolate the zero rates (or zero prices, instead) for the corresponding maturities on the interval dates. Then use the interpolated zero rates to deduce the forward rates, and thus the spread of Eurodollar forward curve segments versus the relevant forward segments from Treasury bills.

Additionally, the variety of curve functions (including `zero2fwd`) helps to standardize such calculations. For instance, by making both rates quoted with quarterly compounding and on an actual/360 basis, the resulting spread structure is fully comparable. This avoids the small inconsistency that occurs when directly comparing the bond-equivalent yield of a Treasury bill to the quarterly forward rates implied by Eurodollar futures.

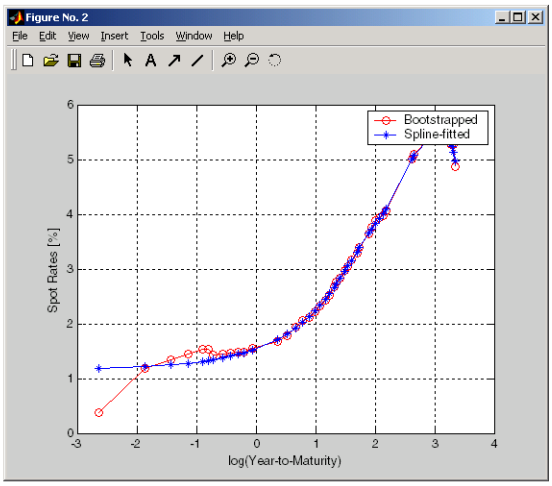
Noise in Curve Computations

When introducing more bonds in constructing curves, noise may become a factor and may need some “smoothing” (with splines, for example); this helps obtain a smoother forward curve.

The following spot and forward curves are constructed from 67 Treasury bonds. The fitted and bootstrapped spot curve (bottom right figure) displays comparable stability. The forward curve (upper-left figure) contains significant noise and shows an improbable forward rate structure. The noise is not necessarily bad; it could uncover trading opportunities for a relative-value approach. Yet, a more balanced approach is desired when the bootstrapped forward curve oscillates this much and contains a negative rate as large as -10% (not shown in the plot because it is outside the limits).



Implied Forward Curves. The jagged curve comes from direct bootstrapping. The smooth curve shows the effect of smoothing with splines.



Implied Spot Rate Curves. These curves correspond to the forward curve above.

This example uses `termfit`, an example function from Financial Toolbox software that also requires the use of Curve Fitting Toolbox™ software.

See Also

`zbtyield` | `zbtprice`

Related Examples

- “Agency Option-Adjusted Spreads” on page 6-2
- “Using Zero-Coupon Bonds” on page 6-5
- “Stepped-Coupon Bonds” on page 6-8

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3

Derivative Securities

Interest Rate Swaps

In this section...

Swap Pricing Assumptions

Financial Instruments Toolbox contains the function `liborfloat2fixed`, which computes a fixed-rate par yield that equates the floating-rate side of a swap to the fixed-rate side. The solver sets the present value of the fixed side to the present value of the floating side without having to line up and compare fixed and floating periods.

Assumptions on Floating-Rate Input

- Rates are quarterly, for example, that of Eurodollar futures.
- Effective date is the first third Wednesday after the settlement date.
- All delivery dates are spaced 3 months apart.
- All periods start on the third Wednesday of delivery months.
- All periods end on the same dates of delivery months, 3 months after the start dates.
- Accrual basis of floating rates is actual/360.
- Applicable forward rates are estimated by interpolation in months when forward-rate data is not available.

Assumptions on Fixed-Rate Output

- Design allows you to create a bond of any coupon, basis, or frequency, based on the floating-rate input.
- The start date is a valuation date, that is, a date when an agreement to enter into a contract by the settlement date is made.
- Settlement can be on or after the start date. If it is after, a forward fixed-rate contract results.
- Effective date is assumed to be the first third Wednesday after settlement, the same date as that of the floating rate.
- The end date of the bond is a designated number of years away, on the same day and month as the effective date.
- Coupon payments occur on anniversary dates. The frequency is determined by the period of the bond.
- Fixed rates are not interpolated. A fixed-rate bond of the same present value as that of the floating-rate payments is created.

Swap Pricing Example

This example shows the use of the functions in computing the fixed rate applicable to a series of 2-, 5-, and 10-year swaps based on Eurodollar market data. According to the Chicago Mercantile Exchange (`https://www.cmegroup.com`), Eurodollar data on Friday, October 11, 2002, was as shown in the following table.

Note This example illustrates swap calculations in MATLAB software. Timing of the data set used was not rigorously examined and was assumed to be the proxy for the swap rate reported on October 11, 2002.

Eurodollar Data on Friday, October 11, 2002

Month	Year	Settle
10	2002	98.21
11	2002	98.26
12	2002	98.3
1	2003	98.3
2	2003	98.31
3	2003	98.275
6	2003	98.12
9	2003	97.87
12	2003	97.575
3	2004	97.26
6	2004	96.98
9	2004	96.745
12	2004	96.515
3	2005	96.33
6	2005	96.135
9	2005	95.955
12	2005	95.78
3	2006	95.63
6	2006	95.465
9	2006	95.315
12	2006	95.16
3	2007	95.025
6	2007	94.88
9	2007	94.74
12	2007	94.595
3	2008	94.48
6	2008	94.375
9	2008	94.28
12	2008	94.185
3	2009	94.1
6	2009	94.005
9	2009	93.925
12	2009	93.865
3	2010	93.82
6	2010	93.755
9	2010	93.7
12	2010	93.645

Month	Year	Settle
3	2011	93.61
6	2011	93.56
9	2011	93.515
12	2011	93.47
3	2012	93.445
6	2012	93.41
9	2012	93.39

Using this data, you can compute 1-, 2-, 3-, 4-, 5-, 7-, and 10-year swap rates with the toolbox function `liborfloat2fixed`. The function requires you to input only Eurodollar data, the settlement date, and tenor of the swap. MATLAB software then performs the required computations.

To illustrate how this function works, first load the data contained in the supplied Excel® worksheet `EDdata.xls`.

```
[EDRawData, textdata] = xlsread('EDdata.xls');
```

Extract the month from the first column and the year from the second column. The rate used as proxy is the arithmetic average of rates on opening and closing.

```
Month = EDRawData(:,1);
Year  = EDRawData(:,2);
IMMData = (EDRawData(:,4)+EDRawData(:,6))/2;
EDFutData = [Month, Year, IMMData]
```

```
EDFutData =

   1.0e+03 *

    0.0100    2.0020    0.0982
    0.0110    2.0020    0.0983
    0.0120    2.0020    0.0983
    0.0010    2.0030    0.0983
    0.0020    2.0030    0.0983
    0.0030    2.0030    0.0983
    0.0060    2.0030    0.0982
    0.0090    2.0030    0.0979
    0.0120    2.0030    0.0976
    0.0030    2.0040    0.0973
    0.0060    2.0040    0.0970
    0.0090    2.0040    0.0968
    0.0120    2.0040    0.0966
    0.0030    2.0050    0.0964
    0.0060    2.0050    0.0962
    0.0090    2.0050    0.0960
    0.0120    2.0050    0.0958
    0.0030    2.0060    0.0957
    0.0060    2.0060    0.0955
    0.0090    2.0060    0.0954
    0.0120    2.0060    0.0952
    0.0030    2.0070    0.0951
    0.0060    2.0070    0.0949
    0.0090    2.0070    0.0948
```

```
    0.0120    2.0070    0.0946
    0.0030    2.0080    0.0945
    0.0060    2.0080    0.0944
    0.0090    2.0080    0.0943
    0.0120    2.0080    0.0942
    0.0030    2.0090    0.0941
    0.0060    2.0090    0.0940
    0.0090    2.0090    0.0939
    0.0120    2.0090    0.0939
    0.0030    2.0100    0.0938
    0.0060    2.0100    0.0937
    0.0090    2.0100    0.0937
    0.0120    2.0100    0.0936
    0.0030    2.0110    0.0936
    0.0060    2.0110    0.0935
    0.0090    2.0110    0.0935
    0.0120    2.0110    0.0935
    0.0030    2.0120    0.0934
    0.0060    2.0120    0.0934
    0.0090    2.0120    0.0934
```

Next, input the current date.

```
Settle = datetime(2002,10,11);
```

To compute for the 2-year swap rate, set the tenor to 2.

```
Tenor = 2;
```

Finally, compute the swap rate with `liborfloat2fixed`.

```
[FixedSpec, ForwardDates, ForwardRates] = ...
liborfloat2fixed(EDFutData, Settle, Tenor)
```

MATLAB returns a par-swap rate of 2.23% using the default setting (quarterly compounding and 30/360 accrual), and forward dates and rates data (quarterly compounded).

```
FixedSpec =

       Coupon: 0.0223
       Settle: '16-Oct-2002'
     Maturity: '16-Oct-2004'
       Period: 4
        Basis: 1

ForwardDates =

      731505
      731596
      731687
      731778
      731869
      731967
      732058
      732149

ForwardRates =
```

```
0.0178
0.0168
0.0171
0.0189
0.0216
0.0250
0.0280
0.0306
```

In the `FixedSpec` output, note that the swap rate actually goes forward from the third Wednesday of October 2002 (October 16, 2002), 5 days after the original `Settle` input (October 11, 2002). This, however, is still the best proxy for the swap rate on `Settle`, as the assumption merely starts the swap's effective period and does not affect its valuation method or its length.

The correction suggested by Hull and White improves the result by turning on convexity adjustment as part of the input to `liborfloat2fixed`. (See Hull, J., *Options, Futures, and Other Derivatives*, 4th Edition, Prentice-Hall, 2000.) For a long swap, for example, five years or more, this correction could prove to be large.

The adjustment requires additional parameters:

- `StartDate`, which you make the same as `Settle` (the default) by providing an empty matrix `[]` as input.
- `ConvexAdj` to tell `liborfloat2fixed` to perform the adjustment.
- `RateParam`, which provides the parameters `a` and `S` as input to the Hull-White short rate process.
- Optional parameters `InArrears` and `Sigma`, for which you can use empty matrices `[]` to accept the MATLAB defaults.
- `FixedCompound`, with which you can facilitate comparison with values cited in Table H15 of *Federal Reserve Statistical Release* by turning the default quarterly compounding into semiannual compounding, with the (default) basis of 30/360.

```
StartDate = [];
Interpolation = [];
ConvexAdj = 1;
RateParam = [0.03; 0.017];
FixedCompound = 2;
[FixedSpec, ForwardDaates, ForwardRates] = ...
liborfloat2fixed(EDFutData, Settle, Tenor, StartDate, ...
Interpolation, ConvexAdj, RateParam, [], [], FixedCompound)
```

This returns 2.21% as the 2-year swap rate, quite close to the reported swap rate for that date.

Analogously, the following table summarizes the solutions for 1-, 3-, 5-, 7-, and 10-year swap rates (convexity-adjusted and unadjusted).

Calculated and Market Average Data of Swap Rates on Friday, October 11, 2002

Swap Length (Years)	Unadjusted	Adjusted	Table H15	Adjusted Error (Basis Points)
1	1.80%	1.79%	1.80%	-1
2	2.24%	2.21%	2.22%	-1
3	2.70%	2.66%	2.66%	0
4	3.12%	3.03%	3.04%	-1
5	3.50%	3.37%	3.36%	+1
7	4.16%	3.92%	3.89%	+3
10	4.87%	4.42%	4.39%	+3

Portfolio Hedging

You can use these results further, such as for hedging a portfolio. The `liborduration` function provides a duration-hedging capability. You can isolate assets (or liabilities) from interest-rate risk exposure with a swap arrangement.

Suppose that you own a bond with these characteristics:

- $100 million face value
- 7% coupon paid semiannually
- 5% yield to maturity
- Settlement on October 11, 2002
- Maturity on January 15, 2010
- Interest accruing on an actual/365 basis

Use of the `bnddury` function from Financial Toolbox software shows a modified duration of 5.6806 years.

To immunize this asset, you can enter into a pay-fixed swap, specifically a swap in the amount of notional principal (*Ns*) such that *Ns***SwapDuration* + $100M*5.6806 = 0 (or *Ns* = -100*5.6806/ *SwapDuration*).

Suppose again, you choose to use a 5-, 7-, or 10-year swap (3.37%, 3.92%, and 4.42% from the previous table) as your hedging tool.

```
SwapFixRate = [0.0337; 0.0392; 0.0442];
Tenor = [5; 7; 10];
Settle = '11-Oct-2002';
PayFixDuration = liborduration(SwapFixRate, Tenor, Settle)
```

```
PayFixDuration =

   -3.6835
   -4.7307
   -6.0661
```

This gives a duration of -3.6835, -4.7307, and -6.0661 years for 5-, 7-, and 10-year swaps. The corresponding notional amount is computed by

```
Ns = -100*5.6806./PayFixDuration

Ns =

  154.2163
  120.0786
   93.6443
```

The notional amount entered in pay-fixed side of the swap instantaneously immunizes the portfolio.

See Also

`liborfloat2fixed` | `liborduration` | `liborprice`

Related Examples

- “Analysis of Bond Futures” on page 7-12
- “Fitting the Diebold Li Model” on page 7-15
- “Managing Interest-Rate Risk with Bond Futures” on page 2-125

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Bond Futures

Bond futures are futures contracts where the commodity for delivery is a government bond. There are established global markets for government bond futures. Bond futures provide a liquid alternative for managing interest-rate risk.

In the U.S. market, the Chicago Mercantile Exchange (CME) offers futures on Treasury bonds and notes with maturities of 2, 5, 10, and 30 years. Typically, the following bond future contracts from the CME have maturities of 3, 6, 9, and 12 months:

- 30-year U.S. Treasury bond
- 10-year U.S. Treasury bond
- 5-year U.S. Treasury bond
- 2-year U.S. Treasury bond

The short position in a Treasury bond or note future contract must deliver to the long position in one of many possible existing Treasury bonds. For example, in a 30-year Treasury bond future, the short position must deliver a Treasury bond with at least 15 years to maturity. Because these bonds have different values, the bond future contract is standardized by computing a conversion factor. The conversion factor normalizes the price of a bond to a theoretical bond with a coupon of 6%. The price of a bond future contract is represented as:

$$InvoicePrice = FutPrice \times CF + AI$$

where:

FutPrice is the price of the bond future.

CF is the conversion factor for a bond to deliver in a futures contract.

AI is the accrued interest.

The short position in a futures contract has the option of which bond to deliver and, in the U.S. bond market, when in the delivery month to deliver the bond. The short position typically chooses to deliver the bond known as the Cheapest to Deliver (CTD). The CTD bond most often delivers on the last delivery day of the month.

Financial Instruments Toolbox software supports the following bond futures:

- U.S. Treasury bonds and notes
- German Bobl, Bund, Buxl, and Schatz
- UK gilts
- Japanese government bonds (JGBs)

The functions supporting all bond futures are:

Function	Purpose
`convfactor`	Calculates bond conversion factors for U.S. Treasury bonds, German Bobl, Bund, Buxl, and Schatz, U.K. gilts, and JGBs.
`bndfutprice`	Prices bond future given repo rates.

Function	Purpose
`bndfutimprepo`	Calculates implied repo rates for a bond future given price.

The functions supporting U.S. Treasury bond futures are:

Function	Purpose
`tfutbyprice`	Calculates future prices of Treasury bonds given the spot price.
`tfutbyyield`	Calculates future prices of Treasury bonds given current yield.
`tfutimprepo`	Calculates implied repo rates for the Treasury bond future given price.
`tfutpricebyrepo`	Calculates Treasury bond futures price given the implied repo rates.
`tfutyieldbyrepo`	Calculates Treasury bond futures yield given the implied repo rates.

See Also

`convfactor` | `bndfutprice` | `bndfutimprepo` | `tfutbyprice` | `tfutbyyield` | `tfutimprepo` | `tfutpricebyrepo` | `tfutyieldbyrepo` | `bnddurp` | `bnddury`

Related Examples

- “Analysis of Bond Futures” on page 7-12
- “Fitting the Diebold Li Model” on page 7-15
- “Managing Interest-Rate Risk with Bond Futures” on page 2-125

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Analysis of Bond Futures

The following example demonstrates analyzing German Euro-Bund futures traded on Eurex. However, `convfactor`, `bndfutprice`, and `bndfutimprepo` apply to bond futures in the U.S., U.K., Germany, and Japan. The workflow for this analysis is:

1. Calculate bond conversion factors.
2. Calculate implied repo rates to find the CTD bond.
3. Price the bond future using the term implied repo rate.

Calculating Bond Conversion Factors

Use conversion factors to normalize the price of a particular bond for delivery in a futures contract. When using conversion factors, the assumption is that a bond for delivery has a 6% coupon. Use `convfactor` to calculate conversion factors for all bond futures from the U.S., Germany, Japan, and U.K.

For example, conversion factors for Euro-Bund futures on Eurex are listed at `www.eurexchange.com`. The delivery date for Euro-Bund futures is the 10th day of the month, as opposed to bond futures in the U.S., where the short position has the option of choosing when to deliver the bond.

For the 4% bond, compute the conversion factor with:

```
CF1 = convfactor('10-Sep-2009','04-Jul-2018', .04,.06,3)
```

```
CF1 =

     0.8659
```

This syntax for `convfactor` works fine for bonds with standard coupon periods. However, some deliverable bonds have long or short first coupon periods. Compute the conversion factors for such bonds using the optional input parameters `StartDate` and `FirstCouponDate`. Specify all optional input arguments for `convfactor` as parameter/value pairs:

```
CF2 = convfactor(datetime(2009,9,10),datetime(2019,1,4), .0375,'Convention',3,'startdate',...
datetime(2008,11,14))
```

```
CF2 =

    0.8426
```

Calculating Implied Repo Rates to Find the CTD Bond

To determine the availability of the cheapest bond for deliverable bonds against a futures contract, compute the implied repo rate for each bond. The bond with the highest repo rate is the cheapest because it has the lowest initial value, thus yielding a higher return, provided you deliver it with the stated futures price. Use `bndfutimprepo` to calculate repo rates:

```
% Bond Properties
CouponRate = [.0425;.0375;.035];
Maturity = [datenum('04-Jul-2018');datenum('04-Jan-2019');datenum('04-Jul-2019')];
CF = [0.882668;0.842556;0.818193];
Price = [105.00;100.89;98.69];

% Futures Properties
FutSettle = '09-Jun-2009';
```

```
FutPrice = 118.54;
Delivery = '10-Sep-2009';

% Note that the default for BNDFUTIMPREPO is for the bonds to be
% semi-annual with a day count basis of 0.  Since these are German
% bonds, we need to have a Basis of 8 and a Period of 1
ImpRepo = bndfutimprepo(Price, FutPrice, FutSettle, Delivery, CF, ...
CouponRate, Maturity,'Basis',8,'Period',1)
```

```
ImpRepo =

    0.0261
   -0.0022
   -0.0315
```

Pricing Bond Futures Using the Term Implied Repo Rate

Use `bndfutprice` to perform price calculations for all bond futures from the U.S., Germany, Japan, and U.K. To price the bond, given a term repo rate:

```
% Assume a term repo rate of .0091;
RepoRate = .0091;
[FutPrice,AccrInt] = bndfutprice(RepoRate, Price(1), FutSettle,...
Delivery, CF(1), CouponRate(1), Maturity(1),...
'Basis',8,'Period',1)
```

```
FutPrice =

  118.0126


AccrInt =

    0.7918
```

See Also

`convfactor` | `bndfutprice` | `bndfutimprepo` | `tfutbyprice` | `tfutbyyield` | `tfutimprepo` | `tfutpricebyrepo` | `tfutyieldbyrepo` | `bnddurp` | `bnddury`

Related Examples

- “Managing Present Value with Bond Futures” on page 7-14
- “Fitting the Diebold Li Model” on page 7-15
- “Managing Interest-Rate Risk with Bond Futures” on page 2-125

More About

- “Bond Futures” on page 7-10
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Managing Present Value with Bond Futures

The Present Value of a Basis Point (PVBP) is used to manage interest-rate risk. PVBP is a measure that quantifies the change in price of a bond given a one-basis point shift in interest rates. The PVBP of a bond is computed with the following:

$$PVBPBond = \frac{Duration \times MarketValue}{100}$$

The PVBP of a bond futures contract can be computed with the following:

$$PVBPFutures = \frac{PVBPCTDBond}{CTDConversionFactor}$$

Use `bnddurp` and `bnddury` from Financial Toolbox software to compute the modified durations of CTD bonds. For more information, see “Managing Interest-Rate Risk with Bond Futures” on page 2-125 and “Fitting the Diebold Li Model” on page 7-15.

See Also

`convfactor` | `bndfutprice` | `bndfutimprepo` | `tfutbyprice` | `tfutbyyield` | `tfutimprepo` | `tfutpricebyrepo` | `tfutyieldbyrepo` | `bnddurp` | `bnddury`

Related Examples

- “Analysis of Bond Futures” on page 7-12
- “Fitting the Diebold Li Model” on page 7-15
- “Managing Interest-Rate Risk with Bond Futures” on page 2-125

More About

- “Bond Futures” on page 7-10
- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Fitting the Diebold Li Model

This example shows how to construct a Diebold Li model of the US yield curve for each month from 1990 to 2010. This example also demonstrates how to forecast future yield curves by fitting an autoregressive model to the time series of each parameter.

The paper can be found here:

https://www.nber.org/papers/w10048

Load the Data

The data used are monthly Treasury yields from 1990 through 2010 for tenors of 1 Mo, 3 Mo, 6 Mo, 1 Yr, 2 Yr, 3 Yr, 5 Yr, 7 Yr, 10 Yr, 20 Yr, 30 Yr.

Daily data can be found here:

https://www.treasury.gov/resource-center/data-chart-center/interest-rates/Pages/TextView.aspx?data=yieldAll

Data is stored in a MATLAB® data file as a MATLAB `dataset` object.

```
load Data_USYieldCurve

% Extract data for the last day of each month
MonthYearMat = repmat((1990:2010)',1,12)';
EOMDates = lbusdate(MonthYearMat(:),repmat((1:12)',21,1));
MonthlyIndex = find(ismember(Dataset.Properties.ObsNames,datestr(EOMDates)));
Estimationdataset = Dataset(MonthlyIndex,:);
EstimationData = double(Estimationdataset);
```

Diebold Li Model

Diebold and Li start with the Nelson Siegel model

$$y = \beta_0 + (\beta_1 + \beta_2)\frac{\tau}{m}(1 - e^{\frac{-m}{\tau}}) - \beta_2 e^{\frac{-m}{\tau}}$$

and rewrite it to be the following:

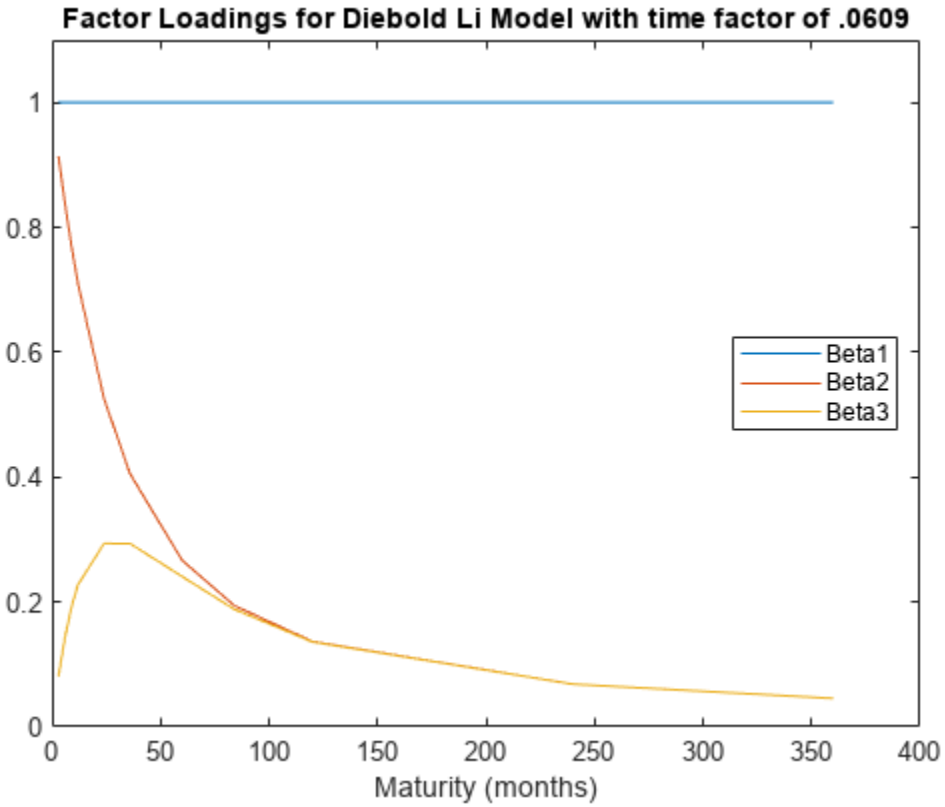
$$y_t(\tau) = \beta_{1t} + \beta_{2t}\left(\frac{1 - e^{-\lambda_t\tau}}{\lambda_t\tau}\right) + \beta_{3t}\left(\frac{1 - e^{-\lambda_t\tau}}{\lambda_t\tau} - e^{-\lambda_t\tau}\right)$$

The above model allows the factors to be interpreted in the following way: Beta1 corresponds to the long term/level of the yield curve, Beta2 corresponds to the short term/slope, and Beta3 corresponds to the medium term/curvature. λ determines the maturity at which the loading on the curvature is maximized, and governs the exponential decay rate of the model.

Diebold and Li advocate setting λ to maximize the loading on the medium term factor, Beta3, at 30 months. This also transforms the problem from a nonlinear fitting to a simple linear regression.

```
% Explicitly set the time factor lambda
lambda_t = .0609;
```

```
% Construct a matrix of the factor loadings
% Tenors associated with data
TimeToMat = [3 6 9 12 24 36 60 84 120 240 360]';
X = [ones(size(TimeToMat)) (1 - exp(-lambda_t*TimeToMat))./(lambda_t*TimeToMat) ...
    ((1 - exp(-lambda_t*TimeToMat))./(lambda_t*TimeToMat) - exp(-lambda_t*TimeToMat))];

% Plot the factor loadings
plot(TimeToMat,X)
title('Factor Loadings for Diebold Li Model with time factor of .0609')
xlabel('Maturity (months)')
ylim([0 1.1])
legend({'Beta1','Beta2','Beta3'},'location','east')
```



Fit the Model

A `DieboldLi` object is developed to facilitate fitting the model from yield data. The `DieboldLi` object inherits from the `IRCurve` object, so the `getZeroRates`, `getDiscountFactors`, `getParYields`, `getForwardRates`, and `toRateSpec` methods are all implemented. Additionally, the method `fitYieldsFromBetas` is implemented to estimate the Beta parameters given a lambda parameter for observed market yields.

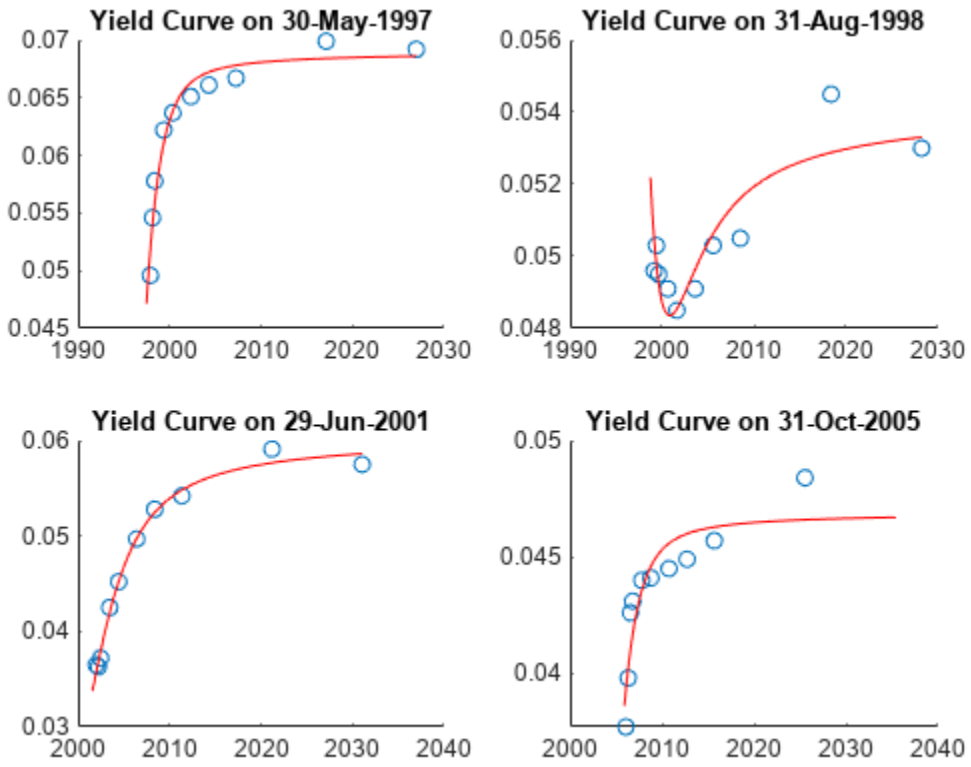
The `DieboldLi` object is used to fit a Diebold Li model for each month from 1990 through 2010.

```
% Preallocate the Betas
Beta = zeros(size(EstimationData,1),3);

% Loop through and fit each end of month yield curve
```

```
for jdx = 1:size(EstimationData,1)
    tmpCurveModel = DieboldLi.fitBetasFromYields(EOMDates(jdx),lambda_t*12,daysadd(EOMDates(jdx)
    Beta(jdx,:) = [tmpCurveModel.Beta1 tmpCurveModel.Beta2 tmpCurveModel.Beta3];
end
```

The Diebold Li fits on selected dates are included here

```
PlotSettles = [datetime(1997,5,30) , datetime(1998,8,31) , datetime(2001,6,29) , datetime(2005,10
figure
for jdx = 1:length(PlotSettles)
    subplot(2,2,jdx)
    tmpIdx = find(strcmpi(Estimationdataset.Properties.ObsNames,datestr(PlotSettles(jdx))));
    tmpCurveModel = DieboldLi.fitBetasFromYields(PlotSettles(jdx),lambda_t*12,...
        daysadd(PlotSettles(jdx),30*TimeToMat),EstimationData(tmpIdx,:)');
    scatter(daysadd(PlotSettles(jdx),30*TimeToMat),EstimationData(tmpIdx,:))
    hold on
    PlottingDates = (PlotSettles(jdx)+30:30:PlotSettles(jdx)+30*360)';
    plot(PlottingDates,tmpCurveModel.getParYields(PlottingDates),'r-')
    title(['Yield Curve on ' datestr(PlotSettles(jdx))])
    datetick
end
```



Forecasting

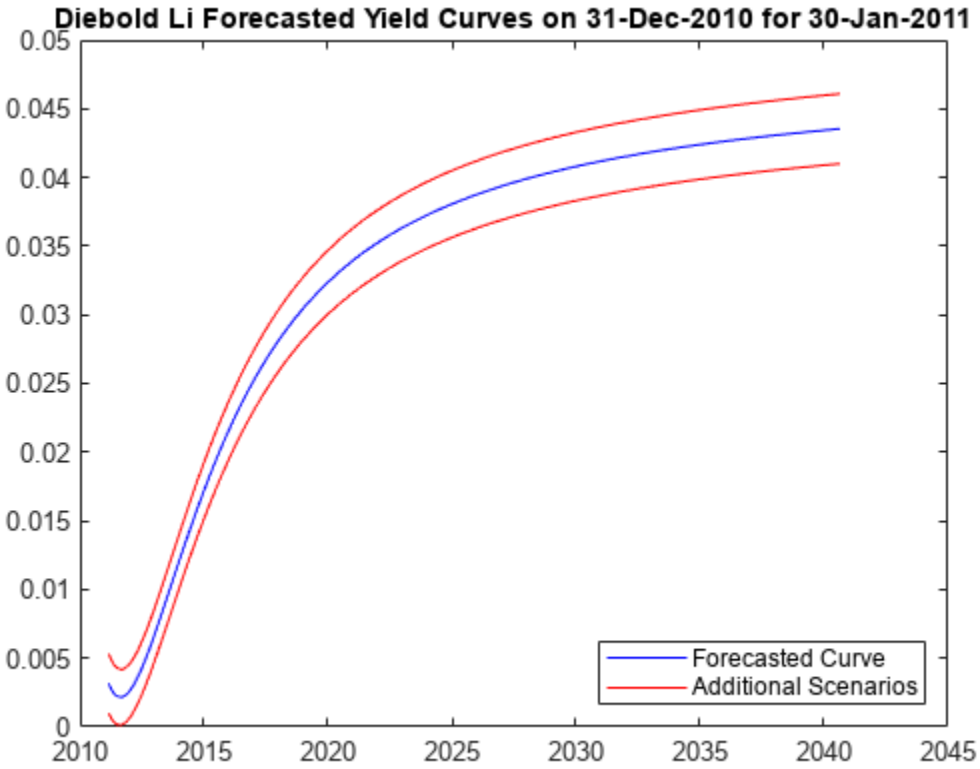
The Diebold Li model can be used to forecast future yield curves. Diebold and Li propose fitting an AR(1) model to the time series of each Beta parameter. This fitted model can then be used to forecast future values of each parameter, and by extension, future yield curves.

For this example the MATLAB function `regress` is used to estimate the parameters for an AR(1) model for each Beta.

The confidence intervals for the regression fit are also used to generate two additional yield curve forecasts that serve as additional possible scenarios for the yield curve.

The `MonthsLag` variable can be adjusted to make different period ahead forecasts. For example, changing the value from `1` to `6` would change the forecast from a 1 month ahead to 6 month ahead forecast.

```
MonthsLag = 1;

[tmpBeta,bint] = regress(Beta(MonthsLag+1:end,1),[ones(size(Beta(MonthsLag+1:end,1))) Beta(1:end
ForecastBeta(1,1) = [1 Beta(end,1)]*tmpBeta;
ForecastBeta_Down(1,1) = [1 Beta(end,1)]*bint(:,1);
ForecastBeta_Up(1,1) = [1 Beta(end,1)]*bint(:,2);
[tmpBeta,bint]  = regress(Beta(MonthsLag+1:end,2),[ones(size(Beta(MonthsLag+1:end,2))) Beta(1:en
ForecastBeta(1,2) = [1 Beta(end,2)]*tmpBeta;
ForecastBeta_Down(1,2) = [1 Beta(end,2)]*bint(:,1);
ForecastBeta_Up(1,2) = [1 Beta(end,2)]*bint(:,2);
[tmpBeta,bint]  = regress(Beta(MonthsLag+1:end,3),[ones(size(Beta(MonthsLag+1:end,3))) Beta(1:en
ForecastBeta(1,3) = [1 Beta(end,3)]*tmpBeta;
ForecastBeta_Down(1,3) = [1 Beta(end,3)]*bint(:,1);
ForecastBeta_Up(1,3) = [1 Beta(end,3)]*bint(:,2);

% Forecasted yield curve
figure
Settle = daysadd(EOMDates(end),30*MonthsLag);
DieboldLi_Forecast = DieboldLi('ParYield',Settle,[ForecastBeta lambda_t*12]);
DieboldLi_Forecast_Up = DieboldLi('ParYield',Settle,[ForecastBeta_Up lambda_t*12]);
DieboldLi_Forecast_Down = DieboldLi('ParYield',Settle,[ForecastBeta_Down lambda_t*12]);
PlottingDates = (Settle+30:30:Settle+30*360)';
plot(PlottingDates,DieboldLi_Forecast.getParYields(PlottingDates),'b-')
hold on
plot(PlottingDates,DieboldLi_Forecast_Up.getParYields(PlottingDates),'r-')
plot(PlottingDates,DieboldLi_Forecast_Down.getParYields(PlottingDates),'r-')
title(['Diebold Li Forecasted Yield Curves on ' datestr(EOMDates(end)) ' for '  datestr(Settle)]
legend({'Forecasted Curve','Additional Scenarios'},'location','southeast')
datetick
```



Bibliography

This example is based on the following paper:

[1] Francis X. Diebold, Canlin Li. "Forecasting the Term Structure of Government Bond Yields." *Journal of Econometrics*, Volume 130, Issue 2, February 2006, pp. 337-364.

See Also

`convfactor` | `bndfutprice` | `bndfutimprepo` | `tfutbyprice` | `tfutbyyield` | `tfutimprepo` | `tfutpricebyrepo` | `tfutyieldbyrepo` | `bnddurp` | `bnddury`

Related Examples

- “Analysis of Bond Futures” on page 7-12
- “Managing Interest-Rate Risk with Bond Futures” on page 2-125

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

Credit Derivatives

Counterparty Credit Risk and CVA

This example shows how to compute the unilateral credit value (valuation) adjustment (CVA) for a bank holding a portfolio of vanilla interest-rate swaps with several counterparties. CVA is the expected loss on an over-the-counter contract or portfolio of contracts due to counterparty default. The CVA for a particular counterparty is defined as the sum over all points in time of the discounted expected exposure at each moment multiplied by the probability that the counterparty defaults at that moment, all multiplied by 1 minus the recovery rate. The CVA formula is:

$$CVA = (1 - R)\int_0^T discEE(t)dPD(t)$$

Where `R` is the recovery, `discEE` the discounted expected exposure at time t, and `PD` the default probability distribution.

The expected exposure is computed by first simulating many future scenarios of risk factors for the given contract or portfolio. Risk factors can be interest rates, as in this example, but will differ based on the portfolio and can include FX rates, equity or commodity prices, or anything that will affect the market value of the contracts. Once a sufficient set of scenarios has been simulated, the contract or portfolio can be priced on a series of future dates for each scenario. The result is a matrix, or "cube", of contract values.

These prices are converted into exposures after taking into account collateral agreements that the bank might have in place as well as netting agreements, as in this example, where the values of several contracts may offset each other, lowering their total exposure.

The contract values for each scenario are discounted to compute the discounted exposures. The discounted expected exposures can then be computed by a simple average of the discounted exposures at each simulation date.

Finally, counterparty default probabilities are typically derived from credit default swap (CDS) market quotes and the CVA for the counterparty can be computed according to the above formula. Assume that a counterparty default is independent of its exposure (no wrong-way risk).

This example demonstrates a portfolio of vanilla interest-rate swaps with the goal of computing the CVA for a particular counterparty.

Read Swap Portfolio

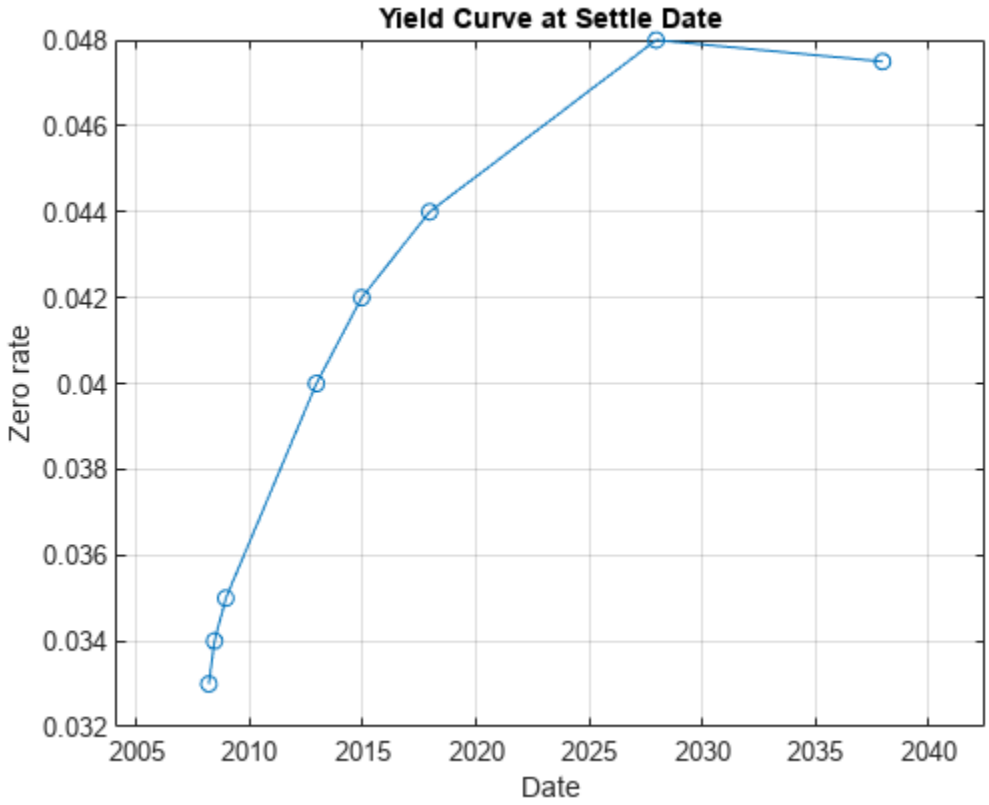
The portfolio of swaps is close to zero value at time t = `0`. Each swap is associated with a counterparty and may or may not be included in a netting agreement.

```
% Read swaps from spreadsheet
swapFile = 'cva-swap-portfolio.xls';
swaps = readtable(swapFile,'Sheet','Swap Portfolio');
swaps.LegType = [swaps.LegType ~swaps.LegType];
swaps.LegRate = [swaps.LegRateReceiving swaps.LegRatePaying];
swaps.LegReset = ones(size(swaps,1),1);

numSwaps = size(swaps,1);
```

For more information on the swap parameters for `CounterpartyID` and `NettingID`, see `creditexposures`. For more information on the swap parameters for `Principal`, `Maturity`, `LegType`, `LegRate`, `LatestFloatingRate`, `Period`, and `LegReset`, see `swapbyzero`.

Create RateSpec from the Interest-Rate Curve

```
Settle = datenum('14-Dec-2007');

Tenor = [3 6 12 5*12 7*12 10*12 20*12 30*12]';
ZeroRates = [0.033 0.034 0.035 0.040 0.042 0.044 0.048 0.0475]';

ZeroDates = datemnth(Settle,Tenor);
Compounding = 2;
Basis = 0;
RateSpec = intenvset('StartDates', Settle,'EndDates', ZeroDates, ...
    'Rates', ZeroRates,'Compounding',Compounding,'Basis',Basis);

figure;
plot(ZeroDates, ZeroRates, 'o-');
xlabel('Date');
datetick('keeplimits');
ylabel('Zero rate');
grid on;
title('Yield Curve at Settle Date');
```



Set Changeable Simulation Parameters

You can vary the number of simulated interest-rate scenarios that you generate. This example sets the simulation dates to be more frequent at first, then turning less frequent further in the future.

```
% Number of Monte Carlo simulations
numScenarios = 1000;
```

```
% Compute monthly simulation dates, then quarterly dates later.
simulationDates = datemnth(Settle,0:12);
simulationDates = [simulationDates datemnth(simulationDates(end),3:3:74)]';
numDates = numel(simulationDates);
```

Compute Floating Reset Dates

For each simulation date, compute previous floating reset date for each swap.

```
floatDates = cfdates(Settle-360,swaps.Maturity,swaps.Period);
swaps.FloatingResetDates = zeros(numSwaps,numDates);
for i = numDates:-1:1
    thisDate = simulationDates(i);
    floatDates(floatDates > thisDate) = 0;
    swaps.FloatingResetDates(:,i) = max(floatDates,[],2);
end
```

Setup Hull-White Single Factor Model

The risk factor that is simulated to value the contracts is the zero curve. For this example, you model the interest-rate term structure using the one-factor Hull-White model. This is a model of the short rate and is defined as:

$$dr = [\theta(t) - ar]dt + \sigma dz$$

where

- dr: Change in the short rate after a small change in time, dt
- a: Mean reversion rate
- σ: Volatility of the short rate
- dz: A Weiner process (a standard normal process)
- $\theta(t)$: Drift function defined as:

$$\theta(t) = F_t(0,t) + aF(0,t) + \frac{\sigma^2}{2a}(1 - e^{-2at})$$

$F(0,t)$: Instantaneous forward rate at time t

$F_t(0,t)$: Partial derivative of F with respect to time

Once you have simulated a path of the short rate, generate a full yield curve at each simulation date using the formula:

$$R(t,T) = -\frac{1}{(T-t)}\ln A(t,T) + \frac{1}{(T-t)}B(t,T)r(t)$$

$$\ln A(t,T) = \ln\frac{P(0,T)}{P(0,t)} + B(t,T)F(0,t) - \frac{1}{4a^3}\sigma^2(e^{-aT} - e^{-at})^2(e^{2at} - 1)$$

$$B(t,T) = \frac{1 - e^{-a(T-t)}}{a}$$

$R(t,T)$: Zero rate at time t for a period of $T - t$

$P(t,T)$: Price of a zero coupon bond at time t that pays one dollar at time T

Each scenario contains the full term structure moving forward through time, modeled at each of our selected simulation dates.

Refer to the “Calibrating Hull-White Model Using Market Data” on page 2-92 example in the Financial Instruments Toolbox™ Users' Guide for more details on Hull-White one-factor model calibration.

```
Alpha = 0.2;
Sigma = 0.015;

hw1 = HullWhite1F(RateSpec,Alpha,Sigma);
```

Simulate Scenarios

For each scenario, simulate the future interest-rate curve at each valuation date using the Hull-White one-factor interest-rate model.

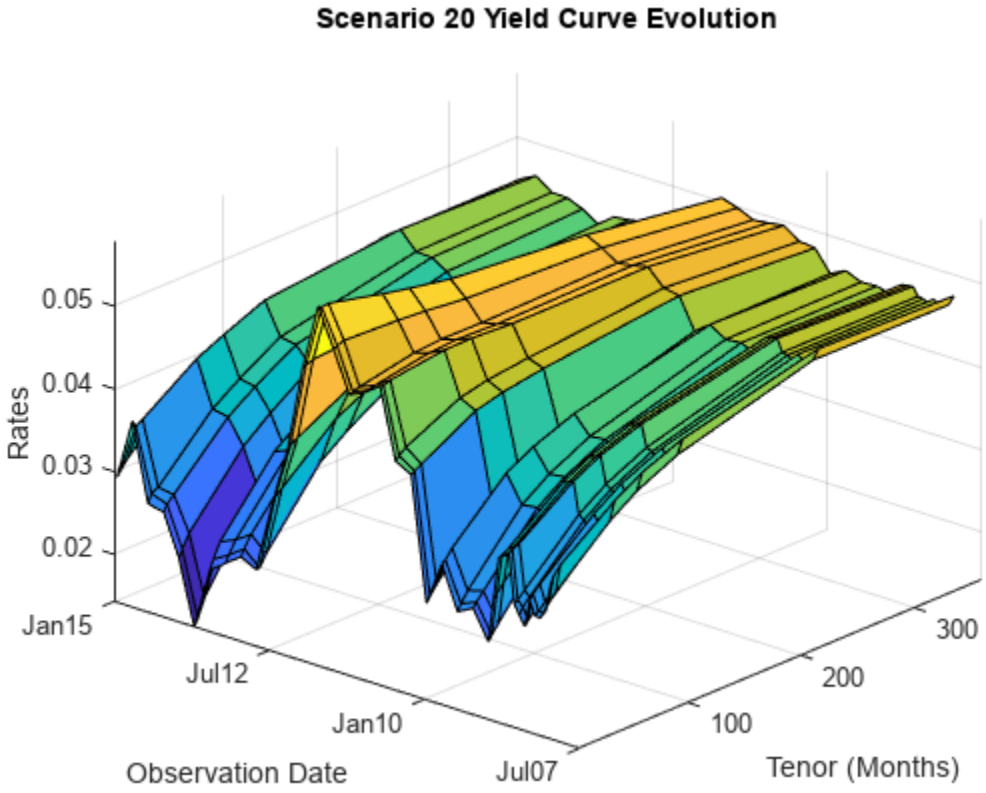
```
% Use reproducible random number generator (vary the seed to produce
% different random scenarios).
prevRNG = rng(0, 'twister');

dt = diff(yearfrac(Settle,simulationDates,1));
nPeriods = numel(dt);
scenarios = hw1.simTermStructs(nPeriods, ...
    'nTrials',numScenarios, ...
    'deltaTime',dt);

% Restore random number generator state
rng(prevRNG);

% Compute the discount factors through each realized interest rate
% scenario.
dfactors = ones(numDates,numScenarios);
for i = 2:numDates
    tenorDates = datemnth(simulationDates(i-1),Tenor);
    rateAtNextSimDate = interp1(tenorDates,squeeze(scenarios(i-1,:,:)), ...
        simulationDates(i),'linear','extrap');
    % Compute D(t1,t2)
    dfactors(i,:) = zero2disc(rateAtNextSimDate, ...
        repmat(simulationDates(i),1,numScenarios),simulationDates(i-1),-1,3);
end
dfactors = cumprod(dfactors,1);
```

Inspect a Scenario

Create a surface plot of the yield curve evolution for a particular scenario.

```
i = 20;
figure;
surf(Tenor, simulationDates, scenarios(:,:,i))
axis tight
datetick('y','mmmyy');
xlabel('Tenor (Months)');
ylabel('Observation Date');
zlabel('Rates');
ax = gca;
```

```
ax.View = [-49 32];
title(sprintf('Scenario %d Yield Curve Evolution\n',i));
```



Compute Mark to Market Swap Prices

For each scenario the swap portfolio is priced at each future simulation date. Prices are computed using a price approximation function, `hswapapprox`. It is common in CVA applications to use simplified approximation functions when pricing contracts due to the performance requirements of these Monte Carlo simulations.

Since the simulation dates do not correspond to the swaps cash flow dates (where the floating rates are reset) estimate the latest floating rate with the 1-year rate (all swaps have period 1 year) interpolated between the nearest simulated rate curves.

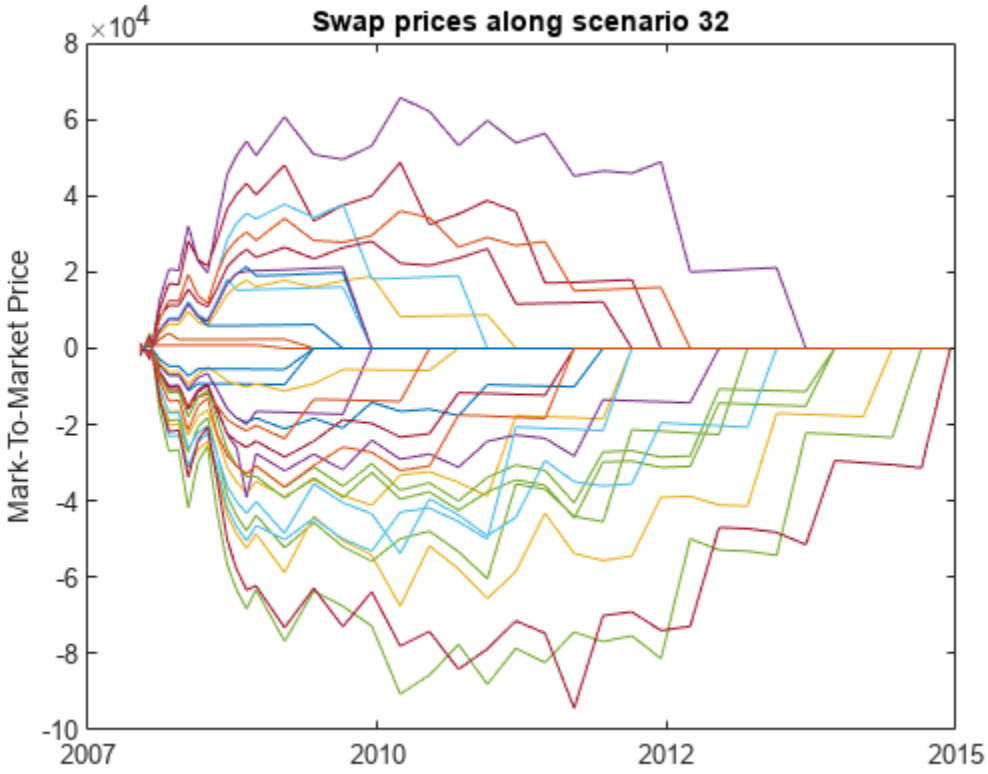
The swap prices are then aggregated into a "cube" of values which contains all future contract values at each simulation date for each scenario. The resulting cube of contract prices is a 3-dimensional matrix where each row represents a simulation date, each column a contract, and each "page" a different simulated scenario.

```
% Compute all mark-to-market values for this scenario. Use an
% approximation function here to improve performance.
values = hcomputeMTMValues(swaps,simulationDates,scenarios,Tenor);
```

Inspect Scenario Prices

Create a plot of the evolution of all swap prices for a particular scenario.

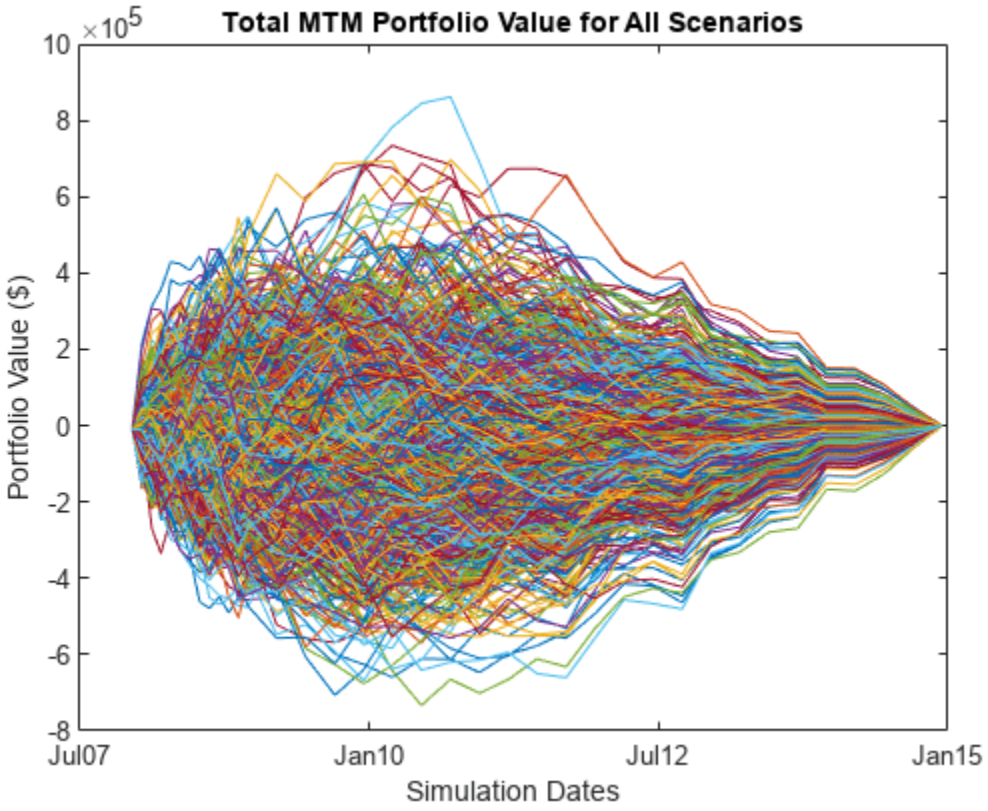
```
i = 32;
figure;
plot(simulationDates, values(:,:,i));
datetick;
ylabel('Mark-To-Market Price');
title(sprintf('Swap prices along scenario %d', i));
```



Visualize Simulated Portfolio Values

Plot the total portfolio value for each scenario of the simulation. As each scenario moves forward in time, the values of the contracts move up or down depending on how the modeled interest-rate term structure changes. As the swaps get closer to maturity, their values will begin to approach zero since the aggregate value of all remaining cash flows will decrease after each cash flow date.

```
% View portfolio value over time
figure;
totalPortValues = squeeze(sum(values, 2));
plot(simulationDates,totalPortValues);
title('Total MTM Portfolio Value for All Scenarios');
datetick('x','mmmyy')
ylabel('Portfolio Value ($)')
xlabel('Simulation Dates')
```



Compute Exposure by Counterparty

The exposure of a particular contract (i) at time t is the maximum of the contract value (Vi) and `0`:

$$E_i(t) = \max\{V_i(t), 0\}$$

And the exposure for a particular counterparty is simply a sum of the individual contract exposures:

$$E_{cp}(t) = \sum E_i(t) = \sum \max\{V_i(t), 0\}$$

In the presence of netting agreements, however, contracts are aggregated together and can offset each other. Therefore the total exposure of all contracts in a netting agreement is:

$$E_{na}(t) = \max\{\sum V_i(t), 0\}$$

Compute these exposures for the entire portfolio as well as each counterparty at each simulation date using the `creditexposures` function.

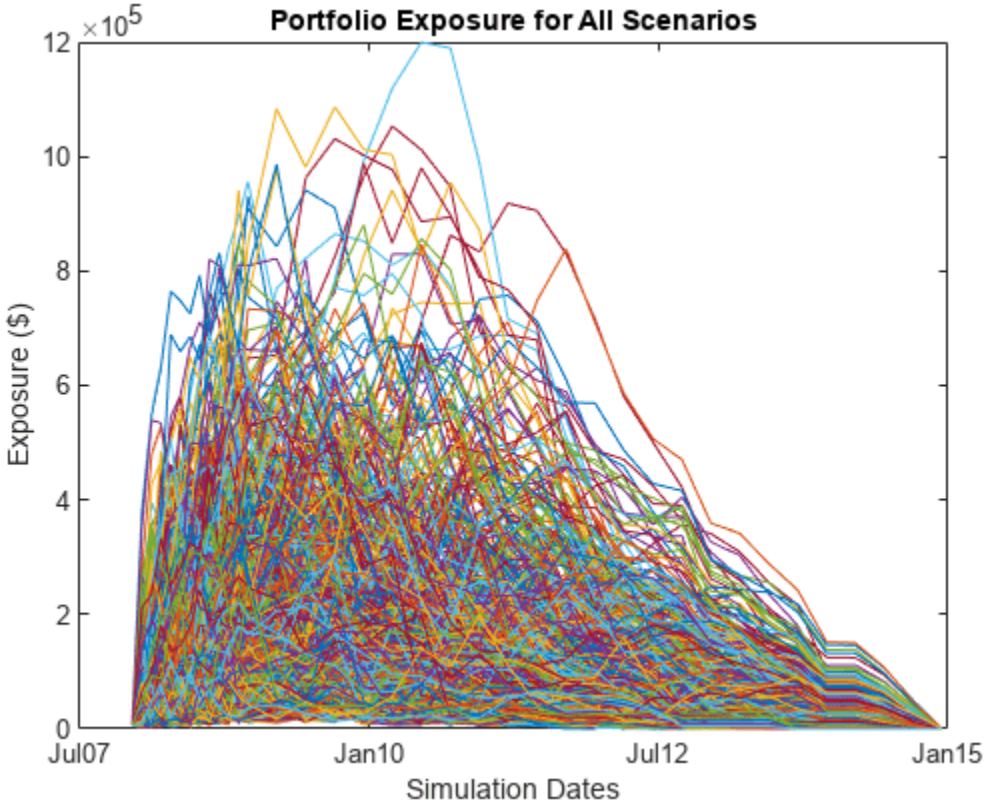
Unnetted contracts are indicated using a `NaN` in the `NettingID` vector. Exposure of an unnetted contract is equal to the market value of the contract if it has positive value, otherwise it is zero.

Contracts included in a netting agreement have their values aggregated together and can offset each other. See the references for more details on computing exposure from mark-to-market contract values.

```
[exposures, expcpty] = creditexposures(values,swaps.CounterpartyID, ...
    'NettingID',swaps.NettingID);
```

Plot the total portfolio exposure for each scenario in our simulation. Similar to the plot of contract values, the exposures for each scenario will approach zero as the swaps mature.

```
% View portfolio exposure over time
figure;
totalPortExposure = squeeze(sum(exposures,2));
plot(simulationDates,totalPortExposure);
title('Portfolio Exposure for All Scenarios');
datetick('x','mmmyy')
ylabel('Exposure ($)')
xlabel('Simulation Dates')
```



Exposure Profiles

Several exposure profiles are useful when analyzing the potential future exposure of a bank to a counterparty. Here you can compute several (non-discounted) exposure profiles per counterparty, as well as, for the entire portfolio.

- `PFE` (Potential Future Exposure): A high percentile (95%) of the distribution of exposures at any particular future date (also called Peak Exposure (PE))
- `MPFE` (Maximum Potential Future Exposure): The maximum PFE across all dates
- `EE` : (Expected Exposure): The mean (average) of the distribution of exposures at each date

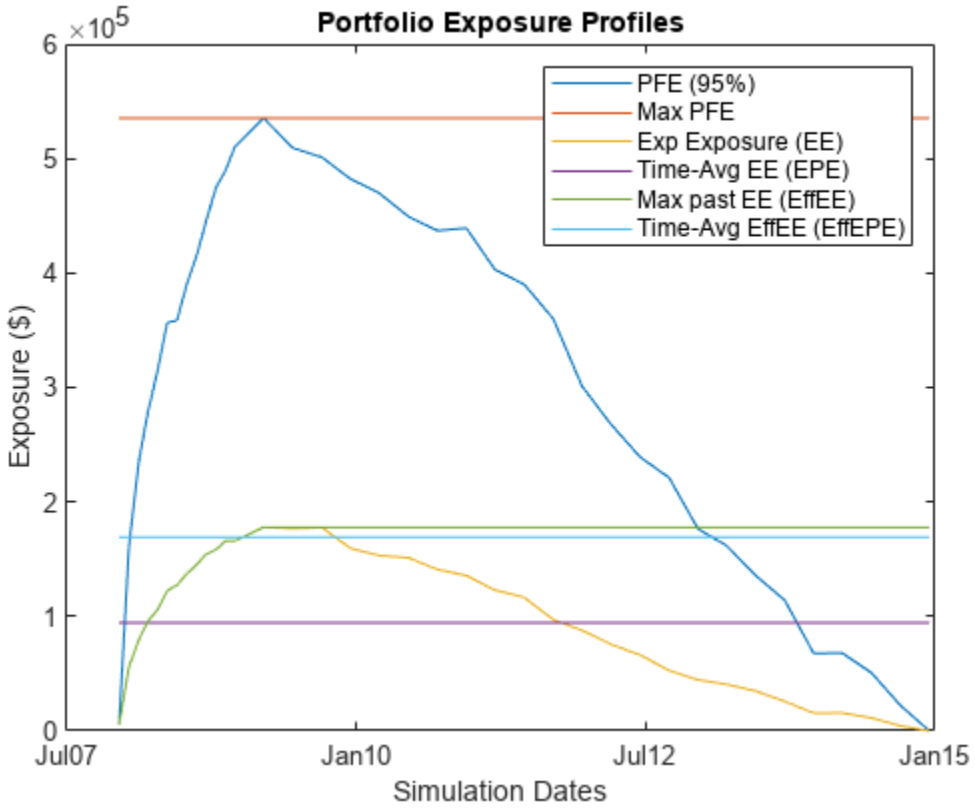
- `EPE` (Expected Positive Exposure): Weighted average over time of the expected exposure
- `EffEE` (Effective Expected Exposure): The maximum expected exposure at any time, *t*, or previous time
- `EffEPE` (Effective Expected Positive Exposure): The weighted average of the effective expected exposure

For further definitions, see for example the Basel II document in references.

```
% Compute entire portfolio exposure
portExposures = sum(exposures,2);

% Compute exposure profiles for each counterparty and entire portfolio
cpProfiles = exposureprofiles(simulationDates,exposures);
portProfiles = exposureprofiles(simulationDates,portExposures);
```

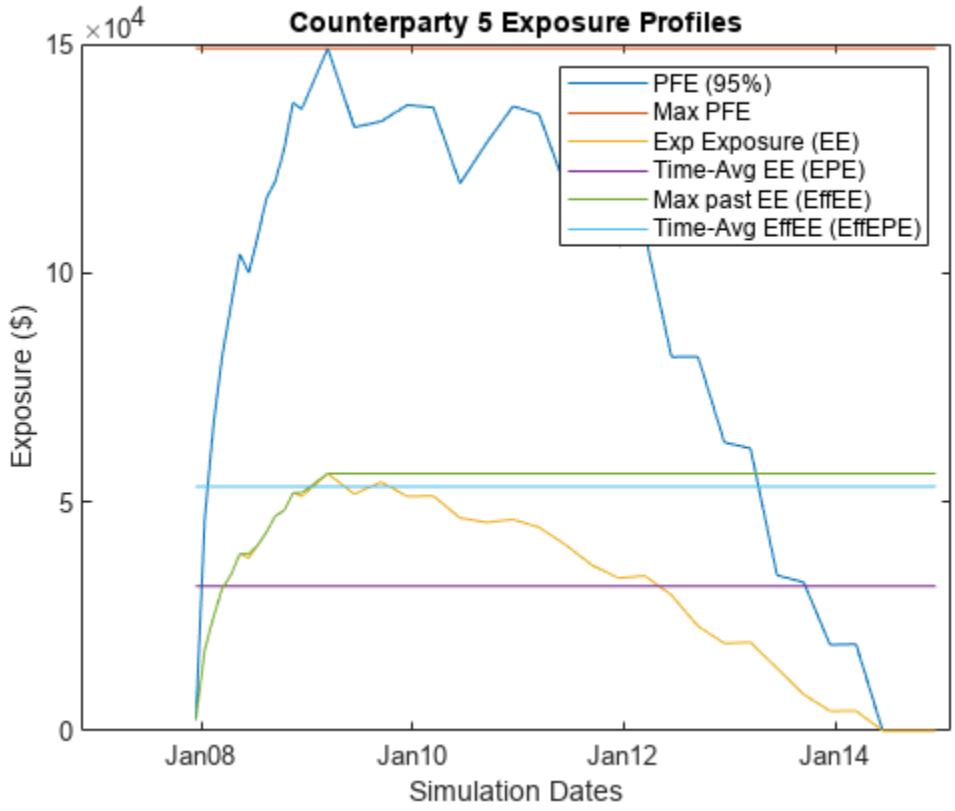
Visualize the exposure profiles, first for the entire portfolio, then for a particular counterparty.

```
% Visualize portfolio exposure profiles
figure;
plot(simulationDates,portProfiles.PFE, ...
    simulationDates,portProfiles.MPFE * ones(numDates,1), ...
    simulationDates,portProfiles.EE, ...
    simulationDates,portProfiles.EPE * ones(numDates,1), ...
    simulationDates,portProfiles.EffEE, ...
    simulationDates,portProfiles.EffEPE * ones(numDates,1));
legend({'PFE (95%)','Max PFE','Exp Exposure (EE)','Time-Avg EE (EPE)', ...
    'Max past EE (EffEE)','Time-Avg EffEE (EffEPE)'})

datetick('x','mmmyy')
title('Portfolio Exposure Profiles');
ylabel('Exposure ($)')
xlabel('Simulation Dates')
```



Visualize exposure profiles for a particular counterparty.

```
cpIdx = find(expcpty == 5);
figure;
plot(simulationDates,cpProfiles(cpIdx).PFE, ...
    simulationDates,cpProfiles(cpIdx).MPFE * ones(numDates,1), ...
    simulationDates,cpProfiles(cpIdx).EE, ...
    simulationDates,cpProfiles(cpIdx).EPE * ones(numDates,1), ...
    simulationDates,cpProfiles(cpIdx).EffEE, ...
    simulationDates,cpProfiles(cpIdx).EffEPE * ones(numDates,1));
legend({'PFE (95%)','Max PFE','Exp Exposure (EE)','Time-Avg EE (EPE)', ...
    'Max past EE (EffEE)','Time-Avg EffEE (EffEPE)'})

datetick('x','mmmyy','keeplimits')
title(sprintf('Counterparty %d Exposure Profiles',cpIdx));
ylabel('Exposure ($)')
xlabel('Simulation Dates')
```



Discounted Exposures

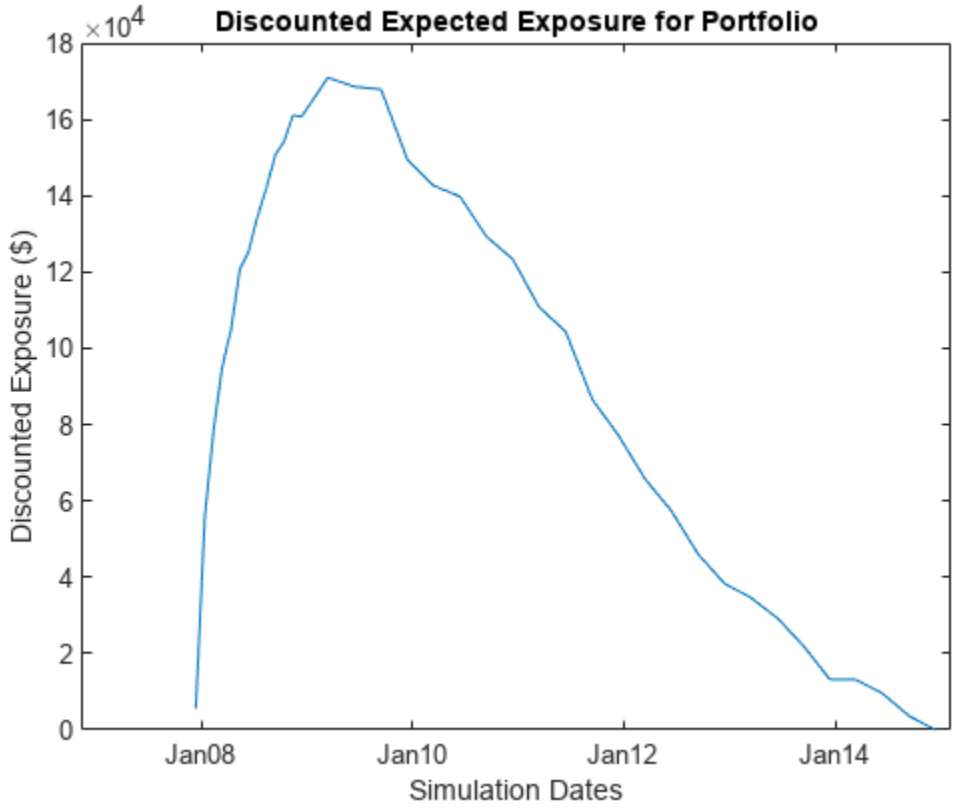
Compute the discounted expected exposures using the discount factors from each simulated interest-rate scenario. The discount factor for a given valuation date in a given scenario is the product of the incremental discount factors from one simulation date to the next, along with the interest-rate path of that scenario.

```
% Get discounted exposures per counterparty, for each scenario
discExp = zeros(size(exposures));
for i = 1:numScenarios
    discExp(:,:,i) = bsxfun(@times,dfactors(:,i),exposures(:,:,i));
end

% Discounted expected exposure
discProfiles = exposureprofiles(simulationDates,discExp, ...
    'ProfileSpec','EE');
```

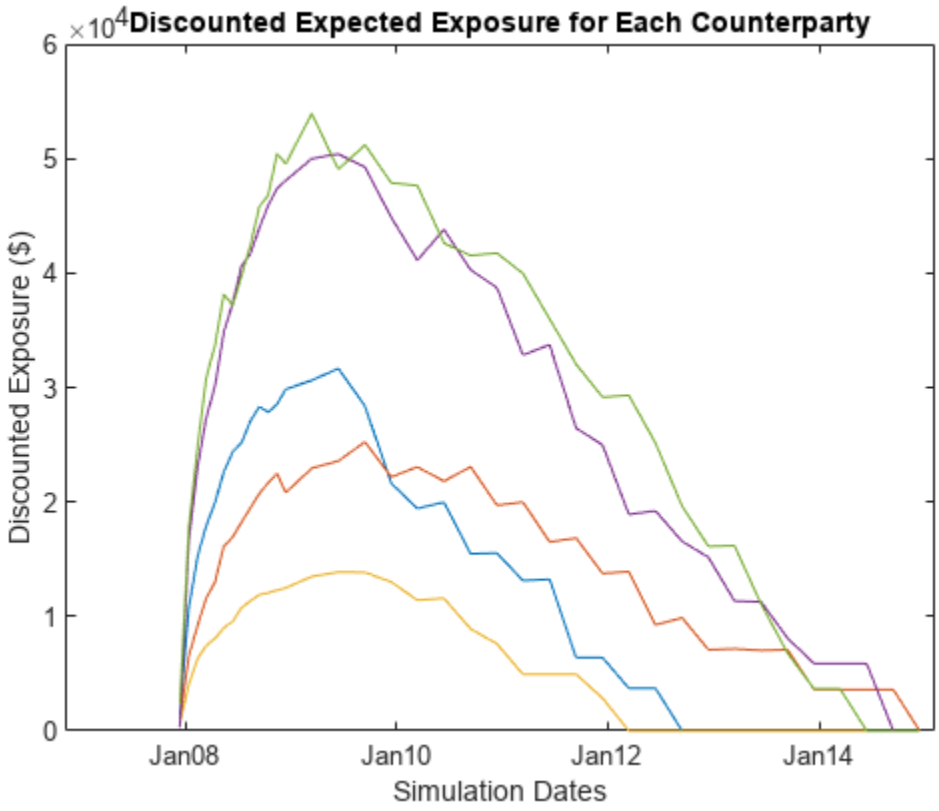
Plot the discounted expected exposures for the aggregate portfolio as well as for each counterparty.

```
% Aggregate the discounted EE for each counterparty into a matrix
discEE = [discProfiles.EE];

% Portfolio discounted EE
figure;
plot(simulationDates,sum(discEE,2))
datetick('x','mmmyy','keeplimits')
title('Discounted Expected Exposure for Portfolio');
```

```
ylabel('Discounted Exposure ($)')
xlabel('Simulation Dates')
```




```
% Counterparty discounted EE
figure;
plot(simulationDates,discEE)
datetick('x','mmmyy','keeplimits')
title('Discounted Expected Exposure for Each Counterparty');
ylabel('Discounted Exposure ($)')
xlabel('Simulation Dates')
```



Calibrating Probability of Default Curve for Each Counterparty

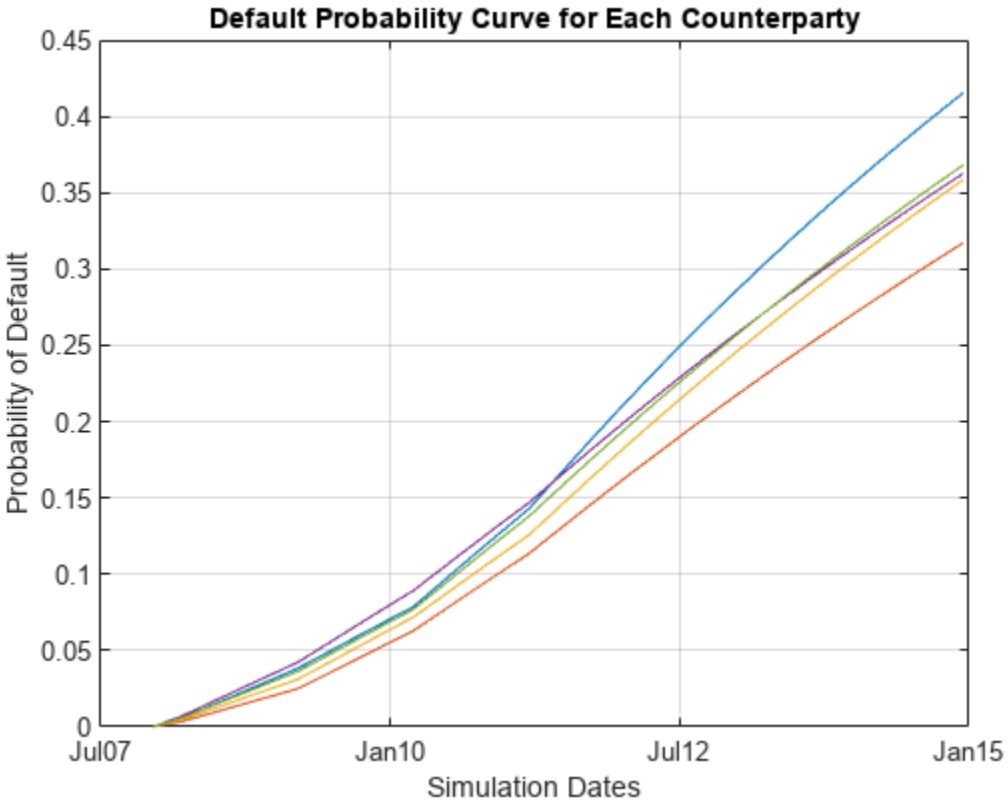
The default probability for a given counterparty is implied by the current market spreads of the counterparty's CDS. Use the function `cdsbootstrap` to generate the cumulative probability of default at each simulation date.

```
% Import CDS market information for each counterparty
CDS = readtable(swapFile,'Sheet','CDS Spreads');
disp(CDS);
```

Date	cp1	cp2	cp3	cp4	cp5
{'3/20/2008'}	140	85	115	170	140
{'3/20/2009'}	185	120	150	205	175
{'3/20/2010'}	215	170	195	245	210
{'3/20/2011'}	275	215	240	285	265
{'3/20/2012'}	340	255	290	320	310

```
CDSDates = datenum(CDS.Date);
CDSSpreads = table2array(CDS(:,2:end));

ZeroData = [RateSpec.EndDates RateSpec.Rates];

% Calibrate default probabilities for each counterparty
DefProb = zeros(length(simulationDates), size(CDSSpreads,2));
for i = 1:size(DefProb,2)
    probData = cdsbootstrap(ZeroData, [CDSDates CDSSpreads(:,i)], ...
```

```
        Settle, 'probDates', simulationDates);
    DefProb(:,i) = probData(:,2);
end

% Plot of the cumulative probability of default for each counterparty.
figure;
plot(simulationDates,DefProb)
title('Default Probability Curve for Each Counterparty');
xlabel('Date');
grid on;
ylabel('Cumulative Probability')
datetick('x','mmmyy')
ylabel('Probability of Default')
xlabel('Simulation Dates')
```



CVA Computation

The Credit Value (Valuation) Adjustment (CVA) formula is:

$$CVA = (1 - R)\int_0^T discEE(t)dPD(t)$$

Where `R` is the recovery, `discEE` the discounted expected exposure at time t, and `PD` the default probability distribution. This assumes the exposure is independent of default (no wrong-way risk), and it also assumes that the exposures were obtained using risk-neutral probabilities.

Approximate the integral with a finite sum over the valuation dates as:

$$CVA(approx) = (1 - R) \sum_{i=2}^{n} discEE(t_i)(PD(t_i) - PD(t_{i-1}))$$

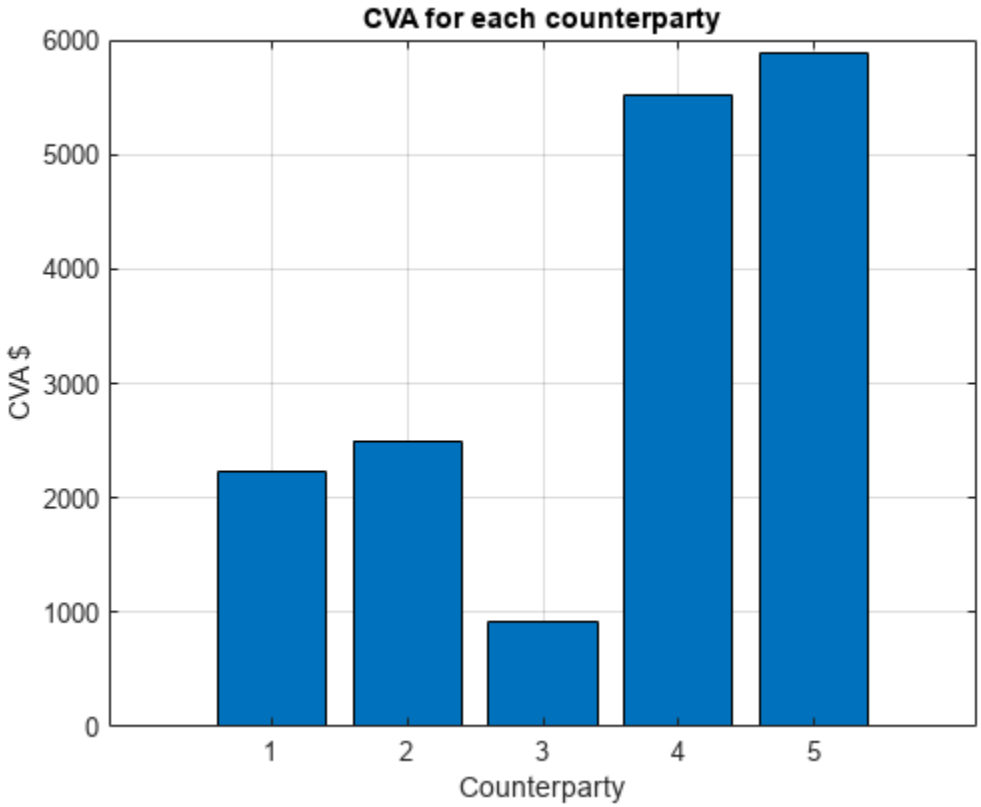
where `t_1` is today's date, `t_2`, ...,`t_`*n* the future valuation dates.

Assume that the CDS information corresponds to the counterparty with index `cpIdx`. The computed CVA is the present market value of our credit exposure to counterparty `cpIdx`. For this example, set the recovery rate at 40%.

```
Recovery = 0.4;
CVA = (1-Recovery) * sum(discEE(2:end,:) .* diff(DefProb));
for i = 1:numel(CVA)
    fprintf('CVA for counterparty %d = $%.2f\n',i,CVA(i));
end
```

```
CVA for counterparty 1 = $2229.38
CVA for counterparty 2 = $2498.71
CVA for counterparty 3 = $918.96
CVA for counterparty 4 = $5521.83
CVA for counterparty 5 = $5883.77
```

```
figure;
bar(CVA);
title('CVA for each counterparty');
xlabel('Counterparty');
ylabel('CVA $');
grid on;
```



References

1. Pykhtin, Michael, and Steven Zhu, *A Guide to Modeling Counterparty Credit Risk*, GARP, July/August 2007, issue 37, pp. 16-22.
2. Pykhtin, Michael, and Dan Rosen, *Pricing Counterparty Risk at the Trade Level and CVA*, 2010.
3. Basel II: https://www.bis.org/publ/bcbs128.pdf page 256.

See Also

`cdsbootstrap` | `cdsprice` | `cdsspread` | `cdsrpv01`

Related Examples

- “First-to-Default Swaps” on page 8-18
- “Credit Default Swap Option” on page 8-27

More About

- “Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94

External Websites

- Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

First-to-Default Swaps

This example shows how to price first-to-default (FTD) swaps under the homogeneous loss assumption.

A first-to-default swap is an instrument that pays a predetermined amount when (and if) the first of a basket of credit instruments defaults. The credit instruments in the basket are usually bonds. If you assume that the loss amount following a credit event is the same for all credits in the basket, you are under the *homogeneous loss* assumption. This assumption makes models simpler because any default in the basket triggers the same payment amount. This example is an implementation of the pricing methodology for these instruments, as described in O'Kane [2 on page 8-26]. There are two steps in the methodology:

- Compute the survival probability for the basket numerically.
- Use this survival curve and standard single-name credit-default swap (CDS) functionality to find FTD spreads and to price existing FTD swaps.

Fit Probability Curves to Market Data

Given CDS market quotes for each issuer in the basket, use `cdsbootstrap` to calibrate individual default probability curves for each issuer.

```
% Interest-rate curve
ZeroDates = datenum({'17-Jan-10','17-Jul-10','17-Jul-11','17-Jul-12',...
'17-Jul-13','17-Jul-14'});
ZeroRates = [1.35 1.43 1.9 2.47 2.936 3.311]'/100;
ZeroData = [ZeroDates ZeroRates];

% CDS spreads
% Each row in MarketSpreads corresponds to a different issuer; each
% column to a different maturity date (corresponding to MarketDates)
MarketDates = datenum({'20-Sep-10','20-Sep-11','20-Sep-12','20-Sep-14',...
'20-Sep-16'});
MarketSpreads = [
   160 195 230 285 330;
   130 165 205 260 305;
   150 180 210 260 300;
   165 200 225 275 295];
% Number of issuers equals number of rows in MarketSpreads
nIssuers = size(MarketSpreads,1);

% Settlement date
Settle = datenum('17-Jul-2009');
```

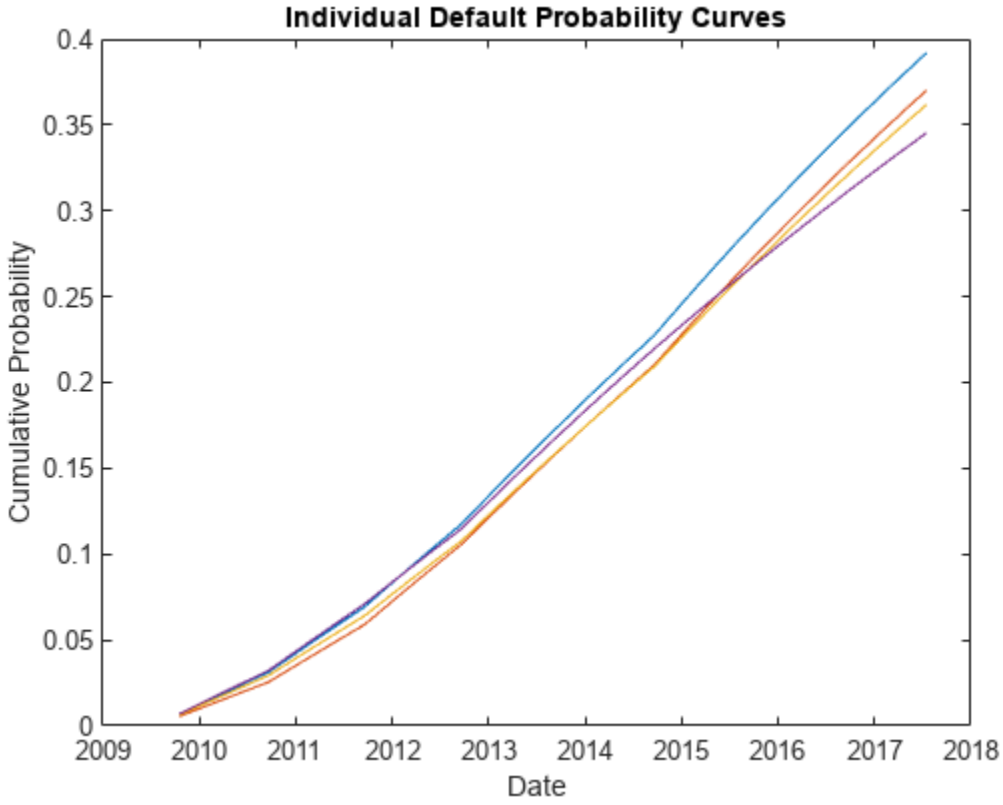
In practice, the time axis is discretized and the FTD survival curve is only evaluated at grid points. This example uses one point every three months. To request that `cdsbootstrap` returns default probability values over the specific grid points that you want, use the optional argument `'ProbDates'`. Add the original standard CDS market dates to the grid, otherwise the default probability information on those dates is interpolated using the two closest dates on the grid, and then the prices on market dates will be inconsistent with the original market data.

```
ProbDates = union(MarketDates,daysadd(Settle,360*(0.25:0.25:8),1));
nProbDates = length(ProbDates);
DefProb = zeros(nIssuers,nProbDates);
```

```
for ii = 1:nIssuers
   MarketData = [MarketDates MarketSpreads(ii,:)'];
   ProbData = cdsbootstrap(ZeroData,MarketData,Settle,...
      'ProbDates',ProbDates);
   DefProb(ii,:) = ProbData(:,2)';
end
```

Here are the calibrated default probability curves for each credit in the basket.

```
figure
plot(ProbDates',DefProb)
datetick
title('Individual Default Probability Curves')
ylabel('Cumulative Probability')
xlabel('Date')
```



Determine Latent Variable Thresholds

Latent variables are used in different credit risk contexts, with different interpretations. In some contexts, a latent variable is a proxy for a *change in the value of assets*, and the domain of this variable is binned, with each bin corresponding to a credit rating. The bins limits, or thresholds, are determined from credit migration matrices. In our context, the latent variable is associated to a *time to default*, and the thresholds determine bins in a discretized time grid where defaults may occur.

Formally, if the time to default of a particular issuer is denoted by τ, and we know its default probability function $P(t)$, a latent variable A and corresponding thresholds $C(t)$ satisfy

$$Pr(\tau \le t) = P(t) = Pr(A \le C(t))$$

or

$$Pr(s \le \tau \le t) = P(t) - P(s) = Pr(C(s) \le A \le C(t))$$

These relationships make latent variable approaches convenient for both simulations and analytical derivations. Both $P(t)$ and $C(t)$ are functions of time.

The choice of a distribution for the variable A determines the thresholds $C(t)$. In the standard latent variable model, the variable A is chosen to follow a standard normal distribution, from which

$$C(t) = \Phi^{-1}(P(t))$$

where Φ is the cumulative standard normal distribution.

Use the previous formula to determine the *default-time thresholds*, or simply *default thresholds*, corresponding to the default probabilities previously obtained for the credits in the basket.

```
DefThresh = norminv(DefProb);
```

Derive Survival Curve for the Basket

Following O'Kane [2 on page 8-26], you can use a one-factor latent variable model to derive expressions for the survival probability function of the basket.

Given parameters β_i for each issuer i, and given independent standard normal variables Z and ϵ_i, the one-factor latent variable model assumes that the latent variable A_i associated to issuer i satisfies

$$A_i = \beta_i * Z + \sqrt{1 - \beta_i^2} * \epsilon_i$$

This induces a correlation between issuers i and j of $\beta_i\beta_j$. All latent variables A_i share the common factor Z as a source of uncertainty, but each latent variable also has an idiosyncratic source of uncertainty ϵ_i. The larger the coefficient β_i, the more the latent variable resembles the common factor Z.

Using the latent variable model, you can derive an analytic formula for the survival probability of the basket. The probability that issuer i survives past time t_j, in other words, that its default time τ_i is greater than t_j is

$$Pr(\tau_i > t_j) = 1 - Pr(A_i \le C_i(t_j))$$

where $C_i(t_j)$ is the default threshold computed above for issuer i, for the j-th date in the discretization grid. Conditional on the value of the one-factor Z, the probability that all issuers survive past time t_j is

$$\begin{aligned} &Pr(\text{No defaults by time } t_j | Z) \\ &= Pr(\tau_i > t_j \text{ for all } i | Z) \\ &= \prod_i [1 - Pr(A_i \le C_i(t_j) | Z)] \end{aligned}$$

where the product is justified because all the ϵ_i's are independent. Therefore, conditional on Z, the A_i's are independent. The unconditional probability of no defaults by time t_j is the integral over all values of Z of the previous conditional probability

$$Pr(\text{No defaults by time } t_j) = \int_Z \prod_i [1 - Pr(A_i \le C_i(t_j) \mid Z)]\phi(Z)dZ$$

with $\phi(Z)$ the standard normal density.

By evaluating this one-dimensional integral for each point t_j in the grid, you get a discretization of the survival curve for the whole basket, which is the FTD survival curve.

The latent variable model can also be used to simulate default times, which is the back engine of many pricing methodologies for credit instruments. Loeffler and Posch [1 on page 8-26], for example, estimate the survival probability of a basket via simulation. In each simulated scenario a time to default is determined for each issuer. With some bookkeeping, the probability of having the first default on each bucket of the grid can be estimated from the simulation. The simulation approach is also discussed in O'Kane [2 on page 8-26]. Simulation is very flexible and applicable to many credit instruments. However, analytic approaches are preferred, when available, because they are much faster and more accurate than simulation.

To compute the FTD survival probabilities in this example, set all betas to the square root of a target correlation. Then you can loop over all dates in the time grid to compute the one-dimensional integral that gives the survival probability of the basket.

Regarding implementation, the conditional survival probability as a function of a scalar Z would be

```
condProb=@(Z)prod(normcdf((-DefThresh(:,jj)+beta*Z)./sqrt(1-beta.^2)));
```

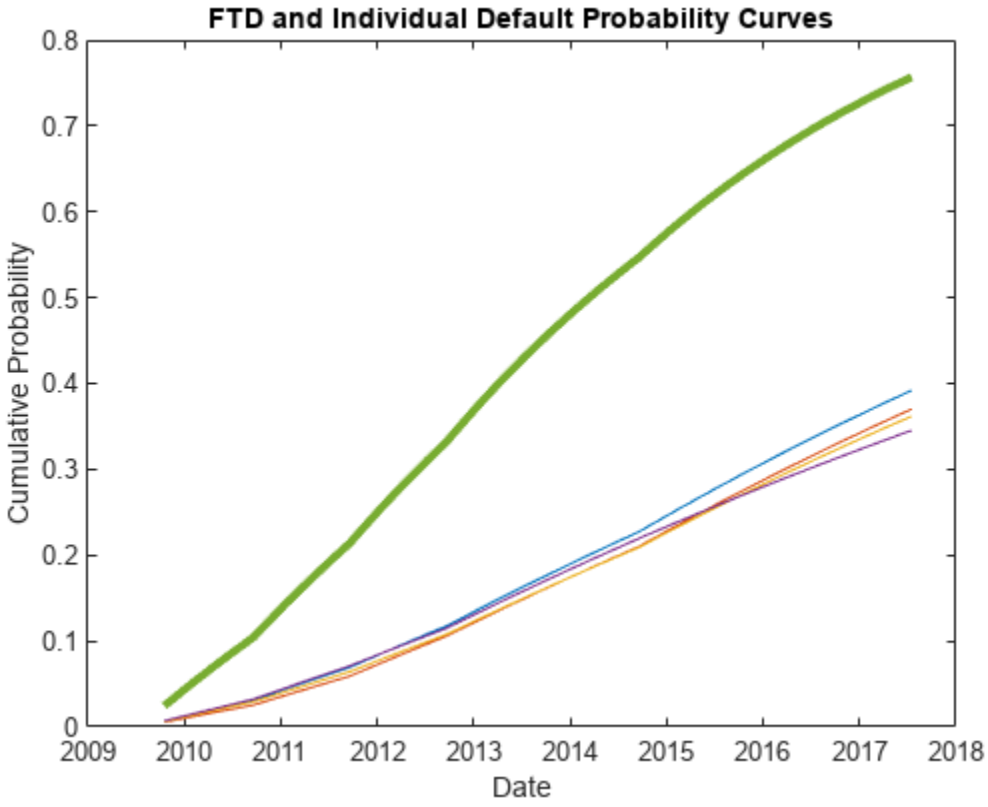
However, the integration function requires that the function handle of the integrand accepts vectors. Although a loop around the scalar version of the conditional probability would work, it is far more efficient to vectorize the conditional probability using `bsxfun`.

```
beta = sqrt(0.25)*ones(nIssuers,1);

FTDSurvProb = zeros(size(ProbDates));
for jj = 1:nProbDates
   % Vectorized conditional probability as a function of Z
   vecCondProb = @(Z)prod(normcdf(bsxfun(@rdivide,...
      -repmat(DefThresh(:,jj),1,length(Z))+bsxfun(@times,beta,Z),...
      sqrt(1-beta.^2))));
   % Truncate domain of normal distribution to [-5,5] interval
   FTDSurvProb(jj) = integral(@(Z)vecCondProb(Z).*normpdf(Z),-5,5);
end
FTDDefProb = 1-FTDSurvProb;
```

Compare the FTD probability to the default probabilities of the individual issuers.

```
figure
plot(ProbDates',DefProb)
datetick
hold on
plot(ProbDates,FTDDefProb,'LineWidth',3)
datetick
```

```
hold off
title('FTD and Individual Default Probability Curves')
ylabel('Cumulative Probability')
xlabel('Date')
```



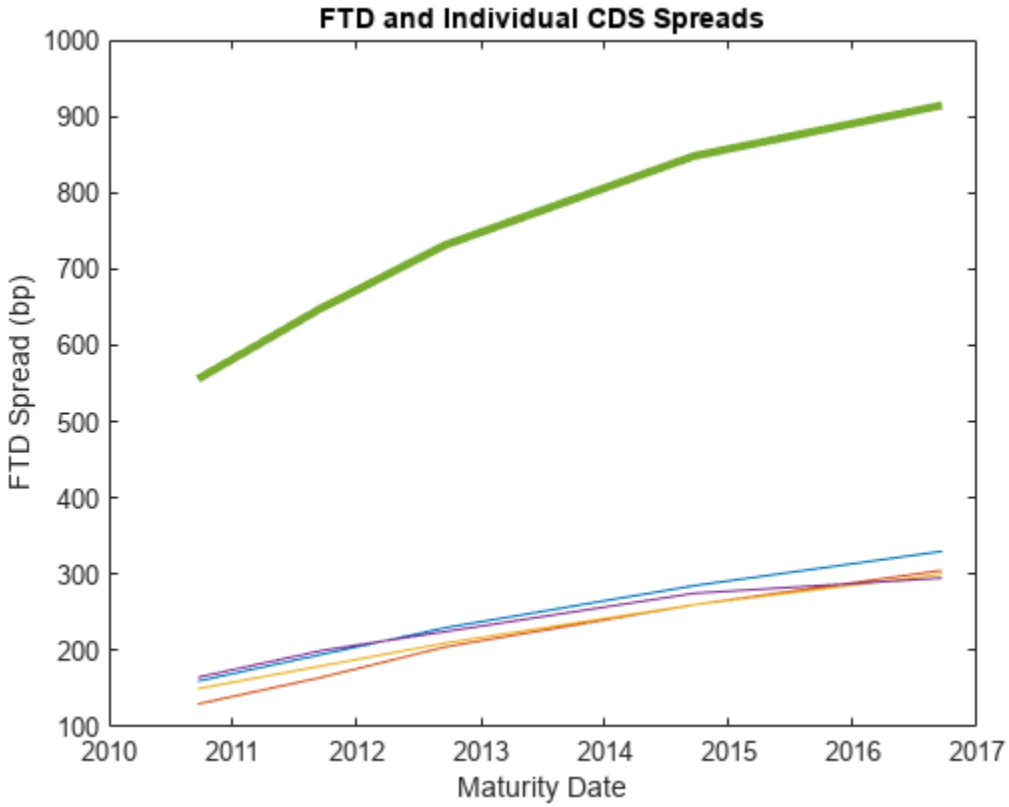
Find FTD Spreads and Price Existing FTD Swaps

Under the assumption that all instruments in the basket have the same recovery rate, or homogeneous loss assumption (see O'Kane [2 on page 8-26]), you get the spread for the FTD swap using the `cdsspread` function by passing the FTD probability data just computed.

```
Maturity = MarketDates;
ProbDataFTD = [ProbDates, FTDDefProb];
FTDSpread = cdsspread(ZeroData,ProbDataFTD,Settle,Maturity);
```

Compare the FTD spreads with the individual spreads.

```
figure
plot(MarketDates,MarketSpreads')
datetick
hold on
plot(MarketDates,FTDSpread,'LineWidth',3)
hold off
title('FTD and Individual CDS Spreads')
ylabel('FTD Spread (bp)')
xlabel('Maturity Date')
```



An existing FTD swap can be priced with `cdsprice`, using the same FTD probability.

```
Maturity0 = MarketDates(1); % Assume maturity on nearest market date
Spread0 = 540; % Spread of existing FTD contract
% Assume default values of recovery and notional
FTDPrice = cdsprice(ZeroData,ProbDataFTD,Settle,Maturity0,Spread0);
fprintf('Price of existing FTD contract: %g\n',FTDPrice)
```

```
Price of existing FTD contract: 17644.7
```

Analyze Sensitivity to Correlation

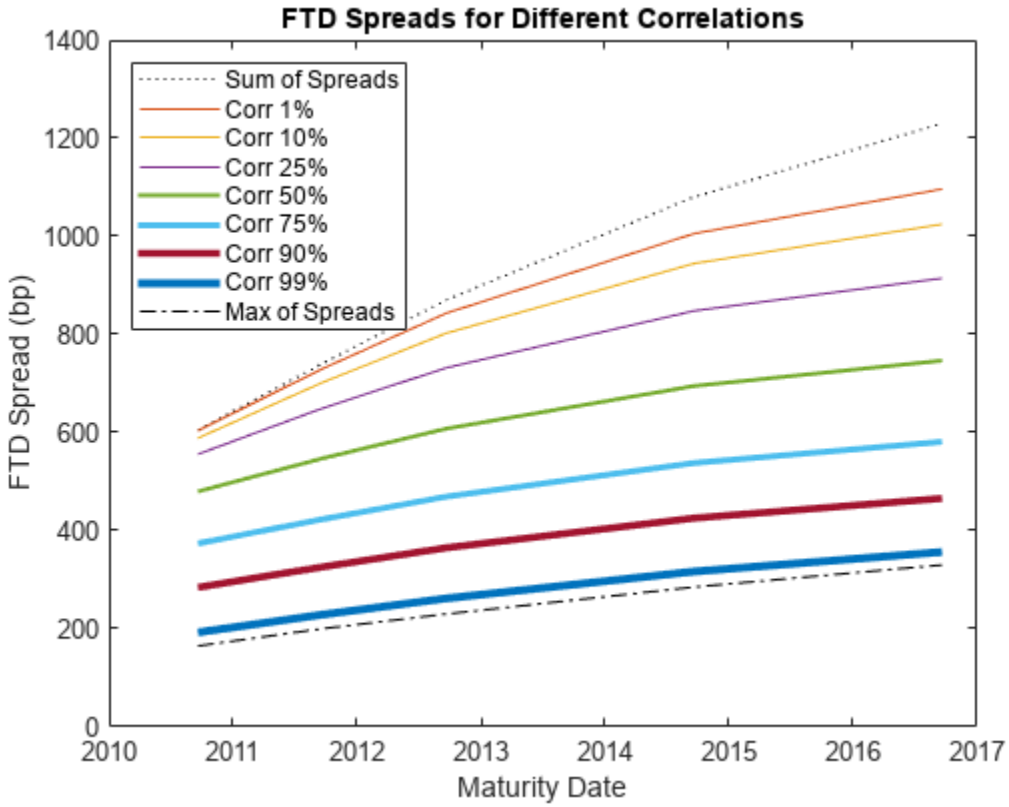
To illustrate the sensitivity of the FTD spreads to model parameters, calculate the market spreads for a range of correlation values.

```
corr = [0 0.01 0.10 0.25 0.5 0.75 0.90 0.99 1];
FTDSpreadByCorr = zeros(length(Maturity),length(corr));
FTDSpreadByCorr(:,1) = sum(MarketSpreads)';
FTDSpreadByCorr(:,end) = max(MarketSpreads)';

for ii = 2:length(corr)-1
   beta = sqrt(corr(ii))*ones(nIssuers,1);
   FTDSurvProb = zeros(length(ProbDates));
   for jj = 1:nProbDates
      % Vectorized conditional probability as a function of Z
      condProb = @(Z)prod(normcdf(bsxfun(@rdivide,...
         -repmat(DefThresh(:,jj),1,length(Z))+bsxfun(@times,beta,Z),...
         sqrt(1-beta.^2))));
```

```
      % Truncate domain of normal distribution to [-5,5] interval
      FTDSurvProb(jj) = integral(@(Z)condProb(Z).*normpdf(Z),-5,5);
   end
   FTDSurvProb = FTDSurvProb(:,1);
   FTDDefProb = 1-FTDSurvProb;
   ProbDataFTD = [ProbDates, FTDDefProb];
   FTDSpreadByCorr(:,ii) = cdsspread(ZeroData,ProbDataFTD,Settle,Maturity);
end
```

The FTD spreads lie in a band between the sum and the maximum of individual spreads. As the correlation increases to one, the FTD spreads decrease towards the maximum of the individual spreads in the basket (all credits default together). As the correlation decreases to zero, the FTD spreads approach the sum of the individual spreads (independent credits).

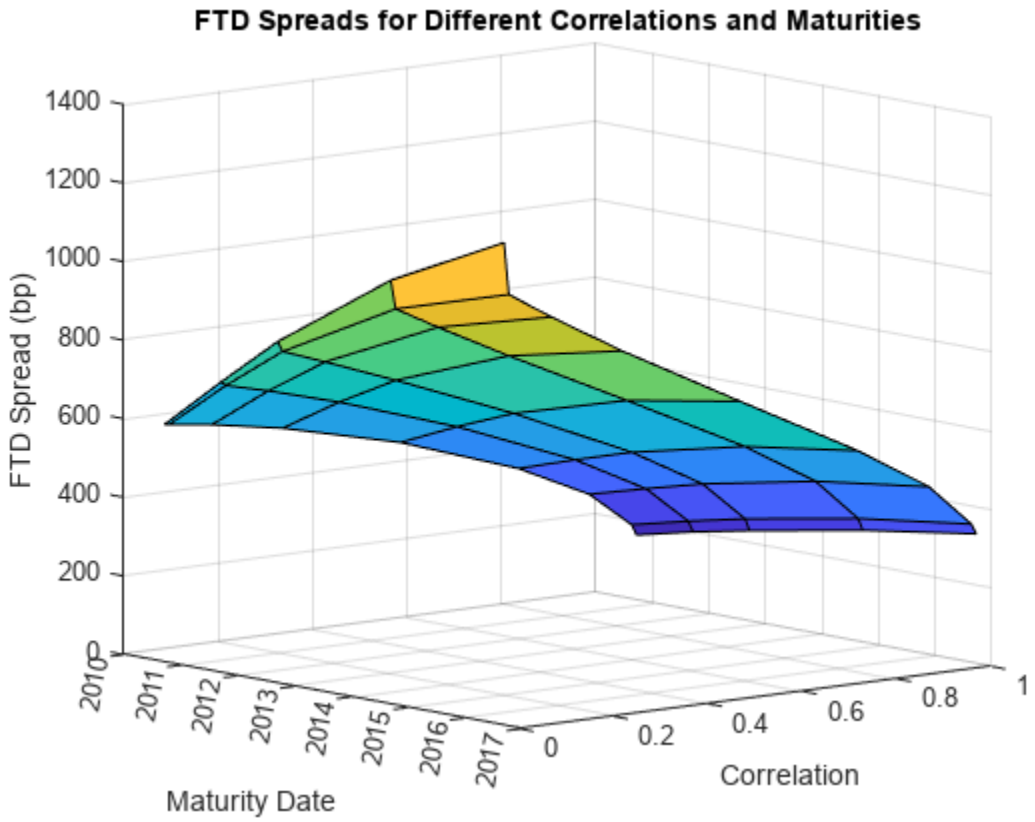
```
figure
legends = cell(1,length(corr));
plot(MarketDates,FTDSpreadByCorr(:,1),'k:')
legends{1} = 'Sum of Spreads';
datetick
hold on
for ii = 2:length(corr)-1
   plot(MarketDates,FTDSpreadByCorr(:,ii),'LineWidth',3*corr(ii))
   legends{ii} = ['Corr ' num2str(corr(ii)*100) '%'];
end
plot(MarketDates,FTDSpreadByCorr(:,end),'k-.')
legends{end} = 'Max of Spreads';

hold off
title('FTD Spreads for Different Correlations')
ylabel('FTD Spread (bp)')
xlabel('Maturity Date')
legend(legends,'Location','NW')
```



For short maturities and small correlations, the basket is effectively independent (the FTD spread is very close to the sum of individual spreads). The correlation effect becomes more significant for longer maturities.

Here is an alternative visualization of the dependency of FTD spreads on correlation.

```
figure
surf(corr,MarketDates,FTDSpreadByCorr)
datetick('y')
ax = gca;
ax.YDir = 'reverse';
view(-40,10)
title('FTD Spreads for Different Correlations and Maturities')
xlabel('Correlation')
ylabel('Maturity Date')
zlabel('FTD Spread (bp)')
```



References

[1] Loeffler, Gunter and Peter Posch. *Credit risk modeling using Excel and VBA.* Wiley Finance, 2007.

[2] O'Kane, Dominic. *Modelling single-name and multi-name Credit Derivatives.* Wiley Finance, 2008.

See Also

`cdsbootstrap` | `cdsprice` | `cdsspread` | `cdsrpv01`

Related Examples

- “Credit Default Swap Option” on page 8-27

More About

- “Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94

External Websites

- Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Credit Default Swap Option

A credit default swap (CDS) option, or credit default swaption, is a contract that provides the holder with the right, but not the obligation, to enter into a credit default swap in the future. CDS options can either be payer swaptions or receiver swaptions. If a payer swaption, the option holder has the right to enter into a CDS where they pay premiums; and, if a receiver swaption, the option holder receives premiums. Financial Instruments Toolbox software provides `cdsoptprice` or `CDSOption` for pricing payer and receiver credit default swaptions. Also, with some additional steps, `cdsoptprice` or `CDSOption` can be used for pricing multi-name CDS index options.

References

O'Kane, D., *Modelling Single-name and Multi-name Credit Derivatives*, Wiley, 2008.

See Also

`cdsoptprice` | `cdsspread` | `cdsrpv01` | `CDSOption` | `CDS`

Related Examples

- “Pricing a Single-Name CDS Option” on page 8-28
- “Pricing a CDS Index Option” on page 8-30
- “Credit Default Swap (CDS)”
- “Price Multiple CDS Option Instruments Using CDS Black Model and CDS Black Pricer” on page 8-46

More About

- “Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94

Pricing a Single-Name CDS Option

This example shows how to price a single-name CDS option using `cdsoptprice`. The function `cdsoptprice` is based on the Black's model as described in O'Kane (2008). The optional `knockout` argument for `cdsoptprice` supports two variations of the mechanics of a CDS option. CDS options can be knockout or non-knockout options.

- A knockout option cancels with no payments if there is a credit event before the option expiry date.
- A non-knockout option does not cancel if there is a credit event before the option expiry date. In this case, the option holder of a non-knockout payer swaption can take delivery of the underlying long protection CDS on the option expiry date and exercise the protection, delivering a defaulted obligation in return for par. This portion of protection from option initiation to option expiry is known as the front-end protection (FEP). While this distinction does not affect the receiver swaption, the price of a non-knockout payer swaption is obtained by adding the value of the FEP to the knockout payer swaption price.

Define the CDS instrument.

```
Settle = datenum('12-Jun-2012');
OptionMaturity = datenum('20-Sep-2012');
CDSMaturity = datenum('20-Sep-2017');
OptionStrike = 200;
SpreadVolatility = .4;
```

Define the zero rate.

```
Zero_Time = [.5 1 2 3 4 5]';
Zero_Rate = [.5 .75 1.5 1.7 1.9 2.2]'/100;
Zero_Dates = daysadd(Settle,360*Zero_Time,1);
ZeroData = [Zero_Dates Zero_Rate]
```

```
ZeroData = 6×2
10^5 ×

    7.3521    0.0000
    7.3540    0.0000
    7.3576    0.0000
    7.3613    0.0000
    7.3649    0.0000
    7.3686    0.0000
```

Define the market data.

```
Market_Time = [1 2 3 5 7 10]';
Market_Rate = [100 120 145 220 245 270]';
Market_Dates = daysadd(Settle,360*Market_Time,1);
MarketData = [Market_Dates Market_Rate];

ProbData = cdsbootstrap(ZeroData, MarketData, Settle)
```

```
ProbData = 6×2
10^5 ×
```

```
    7.3540    0.0000
    7.3576    0.0000
    7.3613    0.0000
    7.3686    0.0000
    7.3759    0.0000
    7.3868    0.0000
```

Define the CDS option.

```
[Payer,Receiver] = cdsoptprice(ZeroData, ProbData, Settle, OptionMaturity, ...
    CDSMaturity, OptionStrike, SpreadVolatility, 'Knockout', true);
fprintf('    Payer: %.0f   Receiver: %.0f  (Knockout)\n',Payer,Receiver);
```

```
    Payer: 196   Receiver: 23  (Knockout)
```

```
[Payer,Receiver] = cdsoptprice(ZeroData, ProbData, Settle, OptionMaturity, ...
    CDSMaturity, OptionStrike, SpreadVolatility, 'Knockout', false);
fprintf('    Payer: %.0f   Receiver: %.0f  (Non-Knockout)\n',Payer,Receiver);
```

```
    Payer: 224   Receiver: 23  (Non-Knockout)
```

See Also

`cdsoptprice` | `cdsspread` | `cdsrpv01`

Related Examples

- “Pricing a CDS Index Option” on page 8-30
- “Credit Default Swap (CDS)”

More About

- “Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94

Pricing a CDS Index Option

This example shows how to price CDS index options by using `cdsoptprice` with the forward spread adjustment. Unlike a single-name CDS, a CDS portfolio index contains multiple credits. When one or more of the credits default, the corresponding contingent payments are made to the protection buyer but the contract still continues with reduced coupon payments. Considering the fact that the CDS index option does not cancel when some of the underlying credits default before expiry, one might attempt to price CDS index options using the Black's model for non-knockout single-name CDS option. However, Black's model in this form is not appropriate for pricing CDS index options because it does not capture the exercise decision correctly when the strike spread (K) is very high, nor does it ensure put-call parity when (K) is not equal to the contractual spread (O'Kane, 2008).

However, with the appropriate modifications, Black's model for single-name CDS options used in `cdsoptprice` can provide a good approximation for CDS index options. While there are some variations in the way the Black's model is modified for CDS index options, they usually involve adjusting the forward spread F, the strike spread K, or both. Here we describe the approach of adjusting the forward spread only. In the Black's model for single-name CDS options, the forward spread F is defined as:

$$F = S(t, t_E, T) = \frac{S(t,T)RPV01(t,T) - S(t,t_E)RPV01(t,t_E)}{RPV01(t,t_E,T)}$$

where

S is the spread.

RPV01 is the risky present value of a basis point (see `cdsrpv01`).

t is the valuation date.

t_E is the option expiry date.

T is the CDS maturity date.

To capture the exercise decision correctly for CDS index options, we use the knockout form of the Black's model and adjust the forward spread to incorporate the *FEP* as follows:

$$F_{Adj} = F + \frac{FEP}{RPV01(t,t_E,T)}$$

with *FEP* defined as

$$FEP = (1 - R)Z(t,t_E)(1 - Q(t,t_E))$$

where

R is the recovery rate.

Z is the discount factor.

Q is the survival probability.

In `cdsoptprice`, forward spread adjustment can be made with the `AdjustedForwardSpread` parameter. When computing the adjusted forward spread, we can compute the spreads using `cdsspread` and the RPV01s using `cdsrpv01`.

Set up the data for the CDS index, its option, and zero curve. The underlying is a 5-year CDS index maturing on 20-Jun-2017 and the option expires on 20-Jun-2012. A flat index spread is assumed when bootstrapping the default probability curve.

```
% CDS index and option data
Recovery = .4;
Basis = 2;
Period = 4;
CDSMaturity = datenum('20-Jun-2017');
ContractSpread = 100;
IndexSpread = 140;
BusDayConvention = 'follow';
Settle = datenum('13-Apr-2012');
OptionMaturity = datenum('20-Jun-2012');
OptionStrike = 140;
SpreadVolatility = .69;

% Zero curve data
MM_Time = [1 2 3 6]';
MM_Rate = [0.004111 0.00563 0.00757 0.01053]';
MM_Dates = daysadd(Settle,30*MM_Time,1);
Swap_Time = [1 2 3 4 5 6 7 8 9 10 12 15 20 30]';
Swap_Rate = [0.01387 0.01035 0.01145 0.01318 0.01508 0.01700 0.01868 ...
    0.02012 0.02132 0.02237 0.02408 0.02564 0.02612 0.02524]';
Swap_Dates = daysadd(Settle,360*Swap_Time,1);

InstTypes = [repmat({'deposit'},size(MM_Time));repmat({'swap'},size(Swap_Time))];
Instruments = [repmat(Settle,size(InstTypes)) [MM_Dates;Swap_Dates] [MM_Rate;Swap_Rate]];

ZeroCurve = IRDataCurve.bootstrap('zero',Settle,InstTypes,Instruments);

% Bootstrap the default probability curve assuming a flat index spread.
MarketData = [CDSMaturity IndexSpread];
ProbDates = datemnth(OptionMaturity,(0:5*12)');
ProbData = cdsbootstrap(ZeroCurve, MarketData, Settle, 'ProbDates', ProbDates);
```

Compute the spot and forward RPV01s, which will be used later in the computation of the adjusted forward spread. For this purpose, we can use `cdsrpv01`.

```
% RPV01(t,T)
RPV01_CDSMaturity = cdsrpv01(ZeroCurve,ProbData,Settle,CDSMaturity)

% RPV01(t,t_E,T)
RPV01_OptionExpiryForward = cdsrpv01(ZeroCurve,ProbData,Settle,CDSMaturity,...
    'StartDate',OptionMaturity)

% RPV01(t,t_E) = RPV01(t,T) - RPV01(t,t_E,T)
RPV01_OptionExpiry = RPV01_CDSMaturity - RPV01_OptionExpiryForward
```

```
RPV01_CDSMaturity =

    4.7853


RPV01_OptionExpiryForward =

    4.5971


RPV01_OptionExpiry =

    0.1882
```

Compute the spot spreads using `cdsspread`.

```
% S(t,t_E)
Spread_OptionExpiry = cdsspread(ZeroCurve,ProbData,Settle,OptionMaturity,...
    'Period',Period,'Basis',Basis,'BusDayConvention',BusDayConvention,...
    'PayAccruedPremium',true,'recoveryrate',Recovery)

% S(t,T)
Spread_CDSMaturity = cdsspread(ZeroCurve,ProbData,Settle,CDSMaturity,...
```

```
    'Period',Period,'Basis',Basis,'BusDayConvention',BusDayConvention,...
    'PayAccruedPremium',true,'recoveryrate',Recovery)
```

```
Spread_OptionExpiry =

  139.9006


Spread_CDSMaturity =

  140.0000
```

The spot spreads and RPV01s are then used to compute the forward spread.

```
% F = S(t,t_E,T)
ForwardSpread = (Spread_CDSMaturity.*RPV01_CDSMaturity - ...
    Spread_OptionExpiry.*RPV01_OptionExpiry)./RPV01_OptionExpiryForward
```

```
ForwardSpread =

  140.0040
```

Compute the front-end protection (FEP).

```
FEP = 10000*(1-Recovery)*ZeroCurve.getDiscountFactors(OptionMaturity)*ProbData(1,2)
```

```
FEP =

   26.3108
```

Compute the adjusted forward spread.

```
AdjustedForwardSpread = ForwardSpread + FEP./RPV01_OptionExpiryForward
```

```
AdjustedForwardSpread =

  145.7273
```

Compute the option prices using `cdsoptprice` with the adjusted forward spread. Note again that the `Knockout` parameter should be set to be `true` because the FEP was already incorporated into the adjusted forward spread.

```
[Payer,Receiver] = cdsoptprice(ZeroCurve, ProbData, Settle, OptionMaturity, ...
    CDSMaturity, OptionStrike, SpreadVolatility,'Knockout',true,...
    'AdjustedForwardSpread', AdjustedForwardSpread,'PayAccruedPremium',true);
fprintf('    Payer: %.0f   Receiver: %.0f  \n',Payer,Receiver);
```

```
Payer: 92   Receiver: 66
```

See Also

`cdsoptprice` | `cdsspread` | `cdsrpv01` | `CDSOption`

Related Examples

- “Pricing a Single-Name CDS Option” on page 8-28
- “Credit Default Swap (CDS)”

More About

- “Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94

Wrong Way Risk with Copulas

This example shows an approach to modeling wrong-way risk for Counterparty Credit Risk using a Gaussian copula.

A basic approach to Counterparty Credit Risk (CCR) (see “Counterparty Credit Risk and CVA” on page 8-2 example) assumes that market and credit risk factors are independent of each other. A simulation of market risk factors drives the exposures for all contracts in the portfolio. In a separate step, Credit-Default Swap (CDS) market quotes determine the default probabilities for each counterparty. Exposures, default probabilities, and a given recovery rate are used to compute the Credit-Value Adjustment (CVA) for each counterparty, which is a measure of expected loss. The simulation of risk factors and the default probabilities are treated as independent of each other.

In practice, default probabilities and market factors are correlated. The relationship may be negligible for some types of instruments, but for others, the relationship between market and credit risk factors may be too important to be ignored when computing risk measures. When the probability of default of a counterparty and the exposure resulting from particular contract tend to increase together we say that the contract has wrong-way risk (WWR).

This example demonstrates an implementation of the wrong-way risk methodology described in Garcia Cespedes et al. (see References on page 8-41).

Exposures Simulation

Many financial institutions have systems that simulate market risk factors and value all the instruments in their portfolios at given simulation dates. These simulations are used to compute exposures and other risk measures. Because the simulations are computationally intensive, reusing them for subsequent risk analyses is important.

This example uses the data and the simulation results from the “Counterparty Credit Risk and CVA” on page 8-2 example, previously saved in the `ccr.mat` file. The `ccr.mat` file contains:

- `RateSpec`: The rate spec when contract values were calculated
- `Settle`: The settle date when contract values were calculated
- `simulationDates`: A vector of simulation dates
- `swaps`: A struct containing the swap parameters
- `values`: The `NUMDATES` x `NUMCONTRACT` x `NUMSCENARIOS` cube of simulated contract values over each date/scenario

This example looks at expected losses over a one-year time horizon only, so the data is cropped after one year of simulation. Simulation dates over the first year are at a monthly frequency, so the 13th simulation date is our one-year time horizon (the first simulation date is the settle date).

```
load ccr.mat

oneYearIdx = 13;
values = values(1:oneYearIdx,:,:);
dates = simulationDates(1:oneYearIdx);

numScenarios = size(values,3);
```

The credit exposures are computed from the simulated contract values. These exposures are monthly credit exposures per counterparty from the settle date to our one-year time horizon.

Since defaults can happen at any time during the one-year time period, it is common to model the exposure at default (EAD) based on the idea of expected positive exposure (EPE). The time-averaged exposure for each scenario is computed, which is called PE (positive exposure). The average of the PE's, including all scenarios, is the EPE, which can also be obtained from the `exposureprofiles` function.

The positive exposure matrix PE contains one row per simulated scenario and one column per counterparty. This is used as the EAD in this analysis.

```
% Compute counterparty exposures
[exposures, counterparties] = creditexposures(values,swaps.Counterparty, ...
    'NettingID',swaps.NettingID);
numCP = numel(counterparties);

% Compute PE (time-averaged exposures) per scenario
intervalWeights = diff(dates) / (dates(end) - dates(1));
exposureMidpoints = 0.5 * (exposures(1:end-1,:,:) + exposures(2:end,:,:));
weightedContributions = bsxfun(@times,intervalWeights,exposureMidpoints);
PE = squeeze(sum(weightedContributions))';

% Compute total portfolio exposure per scenario
totalExp = sum(PE,2);

% Display size of PE and totalExp
whos PE totalExp
```

```
  Name             Size            Bytes  Class     Attributes

  PE            1000x5             40000  double
  totalExp      1000x1              8000  double
```

Credit Simulation

A common approach for simulating credit defaults is based on a "one-factor model", sometimes called the "asset-value approach" (see Gupton et al., 1997 on page 8-41). This is an efficient way to simulate correlated defaults.

Each company *i* is associated with a random variable *Yi*, such that

$$Y_i = \beta_i Z + \sqrt{1 - \beta_i^2}\epsilon_i$$

where *Z* is the "one-factor", a standard normal random variable that represents a systematic credit risk factor whose values affect all companies. The correlation between company *i* and the common factor is given by *beta_i*, the correlation between companies *i* and *j* is *beta_i*beta_j*. The idiosyncratic shock *epsilon_i* is another standard normal variable that may reduce or increase the effect of the systematic factor, independently of what happens with any other company.

If the default probability for company *i* is *PDi*, a default occurs when

$$\Phi(Y_i) < PD_i$$

where Φ is the cumulative standard normal distribution.

The *Yi* variable is sometimes interpreted as asset returns, or sometimes referred to as a latent variable.

This model is a Gaussian copula that introduces a correlation between credit defaults. Copulas offer a particular way to introduce correlation, or more generally, co-dependence between two random variables whose co-dependence is unknown.

Use CDS spreads to bootstrap the one-year default probabilities for each counterparty. The CDS quotes come from the swap-portfolio spreadsheet used in the “Counterparty Credit Risk and CVA” on page 8-2 example.

```
% Import CDS market information for each counterparty
swapFile = 'cva-swap-portfolio.xls';
cds = readtable(swapFile,'Sheet','CDS Spreads');
cdsDates = datenum(cds.Date);
cdsSpreads = table2array(cds(:,2:end));

% Bootstrap default probabilities for each counterparty
zeroData = [RateSpec.EndDates RateSpec.Rates];
defProb = zeros(1, size(cdsSpreads,2));
for i = 1:numel(defProb)
    probData = cdsbootstrap(zeroData, [cdsDates cdsSpreads(:,i)], ...
        Settle, 'probDates', dates(end));
    defProb(i) = probData(2);
end
```

Now simulate the credit scenarios. Because defaults are rare, it is common to simulate a large number of credit scenarios.

The sensitivity parameter *beta* is set to `0.3` for all counterparties. This value can be calibrated or tuned to explore model sensitivities. See the References on page 8-41 for more information.

```
numCreditScen = 100000;
rng('default');

% Z is the single credit factor
Z = randn(numCreditScen,1);

% epsilon is the idiosyncratic factor
epsilon = randn(numCreditScen,numCP);

% beta is the counterparty sensitivity to the credit factor
beta = 0.3 * ones(1,numCP);

% Counterparty latent variables
Y = bsxfun(@times,beta,Z) + bsxfun(@times,sqrt(1 - beta.^2),epsilon);

% Default indicator
isDefault = bsxfun(@lt,normcdf(Y),defProb);
```

Correlating Exposure and Credit Scenarios

Now that there is a set of sorted portfolio exposure scenarios and a set of default scenarios, follow the approach in Garcia Cespedes et al. and use a Gaussian copula to generate correlated exposure-default scenario pairs.

Define a latent variable *Ye* that maps into the distribution of simulated exposures. *Ye* is defined as

$$Y_e = \rho Z + \sqrt{1 - \rho^2}\epsilon_e$$

where *Z* is the systemic factor computed in the credit simulation, *epsilon_e* is an independent standard normal variable and *rho* is interpreted as a market-credit correlation parameter. By construction, *Ye* is a standard normal variable correlated with *Z* with correlation parameter *rho*.

The mapping between *Ye* and the simulated exposures requires us to order the exposure scenarios in a meaningful way, based on some sortable criterion. The criterion can be any meaningful quantity, for example, it could be an underlying risk factor for the contract values (such as an interest rate), the total portfolio exposure, and so on.

In this example, use the total portfolio exposure (`totalExp`) as the exposure scenario criterion to correlate the credit factor with the total exposure. If *rho* is negative, low values of the credit factor *Z* tend to get linked to high values of *Ye*, hence high exposures. This means negative values of *rho* introduce WWR.

To implement the mapping between *Ye* and the exposure scenarios, sort the exposure scenarios by the `totalExp` values. Suppose that the number of exposure scenarios is *S* (`numScenarios`). Given *Ye*, find the value *j* such that

$$\frac{j-1}{S} \le \Phi(Y_e) < \frac{j}{S}$$

and select the scenario *j* from the sorted exposure scenarios.

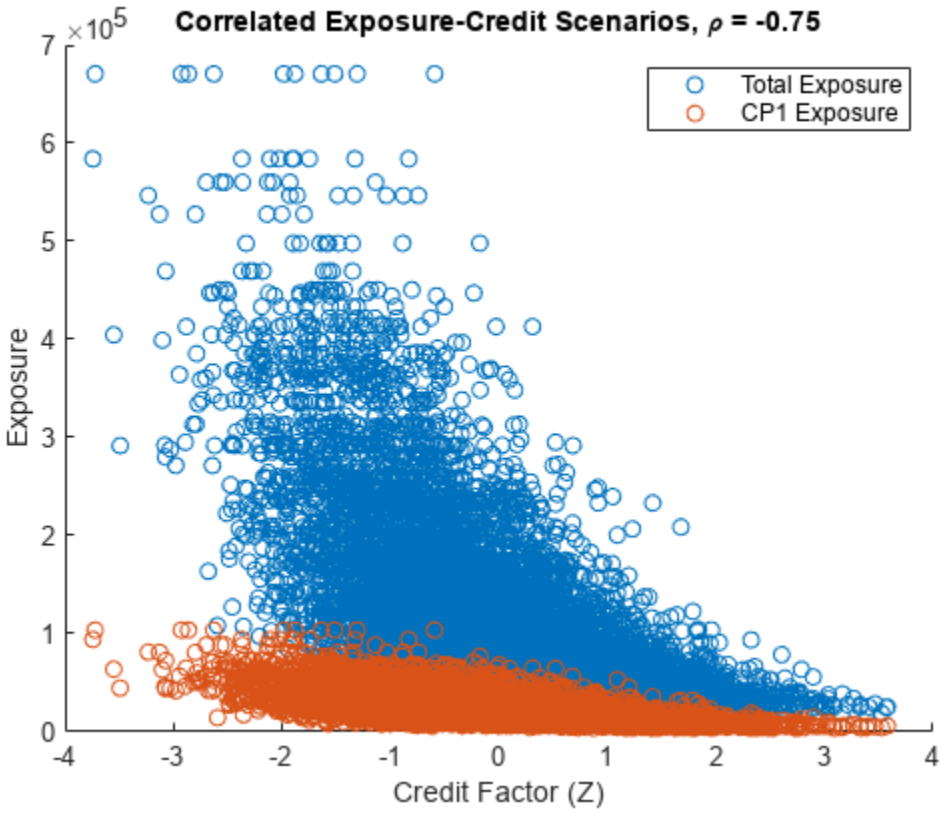
Ye is correlated to the simulated exposures and *Z* is correlated to the simulated defaults. The correlation *rho* between *Ye* and *Z* is, therefore, the correlation link between the exposures and the credit simulations.

```
% Sort the total exposure
[~,totalExpIdx] = sort(totalExp);

% Scenario cut points
cutPoints = 0:1/numScenarios:1;

% epsilonExp is the idiosyncratic factor for the latent variable
epsilonExp = randn(numCreditScen,1);

% Set a market-credit correlation value
rho = -0.75;

% Latent variable
Ye = rho * Z + sqrt(1 - rho^2) * epsilonExp;

% Find corresponding exposure scenario
binidx = discretize(normcdf(Ye),cutPoints);
scenIdx = totalExpIdx(binidx);
totalExpCorr = totalExp(scenIdx);
PECorr = PE(scenIdx,:);
```

The following plot shows the correlated exposure-credit scenarios for the total portfolio exposure as well as for the first counterparty. Because of the negative correlation, negative values of the credit factor *Z* correspond to high exposure levels (wrong-way risk).

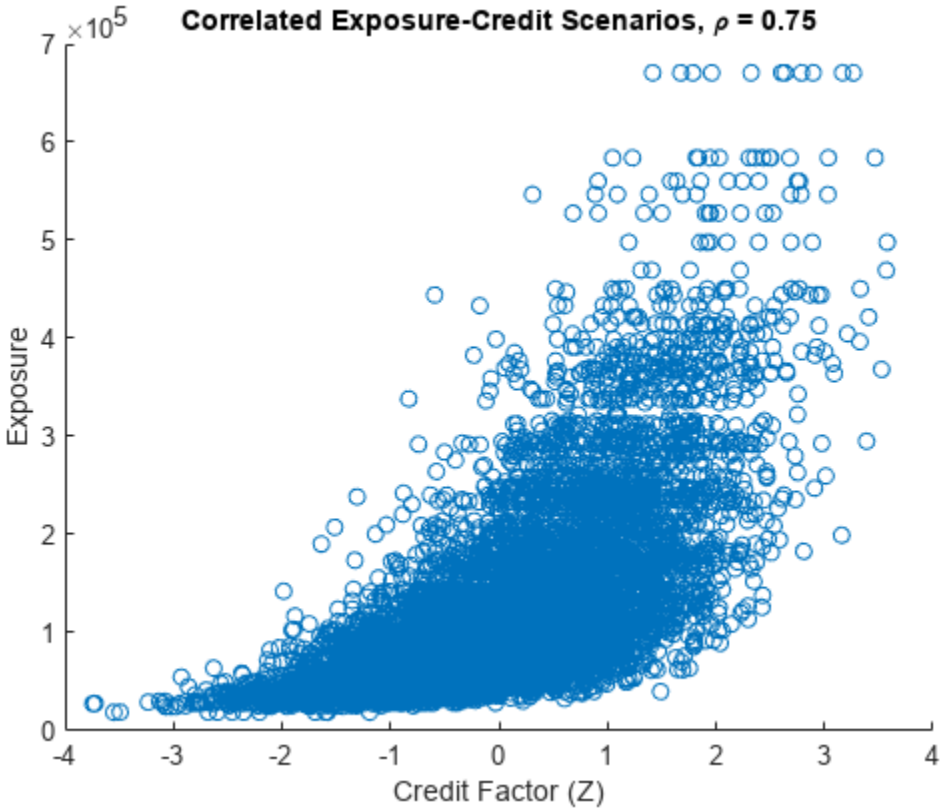
```
% We only plot up to 10000 scenarios
numScenPlot = min(10000,numCreditScen);
figure;
scatter(Z(1:numScenPlot),totalExpCorr(1:numScenPlot))
```

```
hold on
scatter(Z(1:numScenPlot),PECorr(1:numScenPlot,1))
xlabel('Credit Factor (Z)')
ylabel('Exposure')
title(['Correlated Exposure-Credit Scenarios, \rho = ' num2str(rho)])
legend('Total Exposure','CP1 Exposure')
hold off
```



For positive values of *rho,* the relationship between the credit factor and the exposures is reversed (right-way risk).

```
rho = 0.75;
Ye = rho * Z + sqrt(1 - rho^2) * epsilonExp;
binidx = discretize(normcdf(Ye),cutPoints);
scenIdx = totalExpIdx(binidx);
totalExpCorr = totalExp(scenIdx);

figure;
scatter(Z(1:numScenPlot),totalExpCorr(1:numScenPlot))
xlabel('Credit Factor (Z)')
ylabel('Exposure')
title(['Correlated Exposure-Credit Scenarios, \rho = ' num2str(rho)])
```



Sensitivity to Correlation

You can explore the sensitivity of the exposures or other risk measures to a range of values for *rho*.

For each value of *rho*, compute the total losses per credit scenario as well as the expected losses per counterparty. This example assumes a 40% recovery rate.

```
Recovery = 0.4;
rhoValues = -1:0.1:1;

totalLosses = zeros(numCreditScen,numel(rhoValues));
expectedLosses = zeros(numCP, numel(rhoValues));

for i = 1:numel(rhoValues)

    rho = rhoValues(i);

    % Latent variable
    Ye = rho * Z + sqrt(1 - rho^2) * epsilonExp;

    % Find corresponding exposure scenario
    binidx = discretize(normcdf(Ye),cutPoints);
    scenIdx = totalExpIdx(binidx);
    simulatedExposures = PE(scenIdx,:);

    % Compute actual losses based on exposures and default events
    losses = isDefault .* simulatedExposures * (1-Recovery);
```

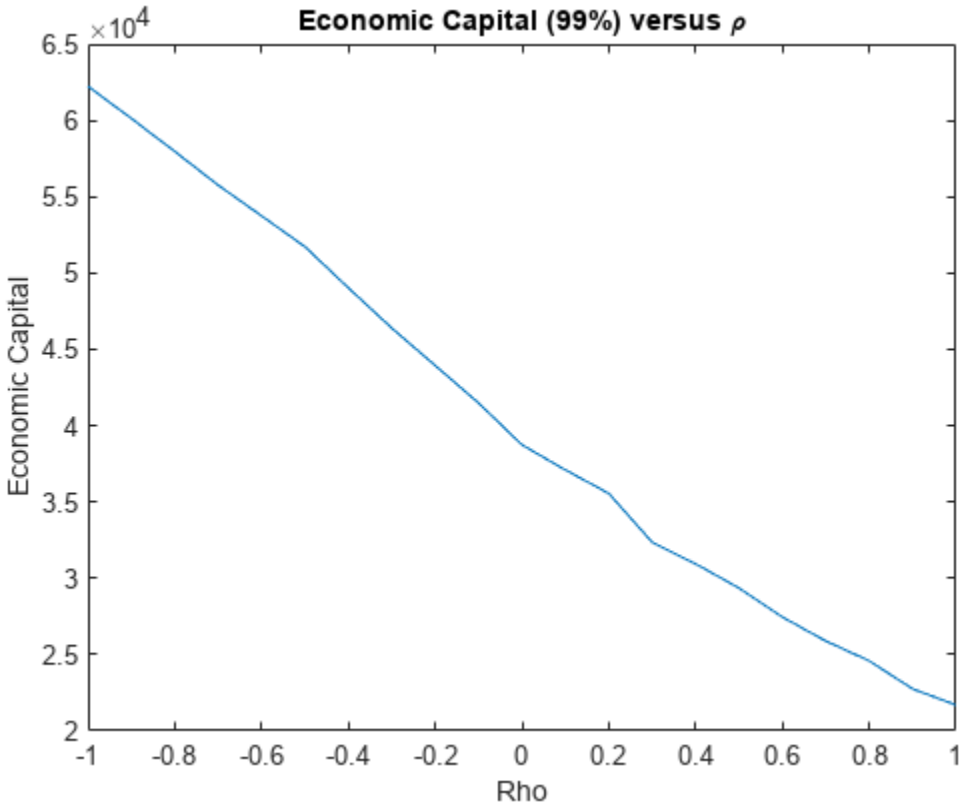
```
    totalLosses(:,i) = sum(losses,2);

    % We compute the expected losses per counterparty
    expectedLosses(:,i) = mean(losses)';
end
displayExpectedLosses(rhoValues, expectedLosses)
```

```
               Expected Losses
 Rho    CP1      CP2      CP3     CP4       CP5
-------------------------------------------------
-1.0   604.10   260.44   194.70  1234.17   925.95-0.9   583.67   250.45   189.02  1158.65   897.9
```

You can visualize the sensitivity of the Economic Capital (EC) to the market-credit correlation parameter. Define EC as the difference between a percentile q of the distribution of losses, minus the expected loss.

Negative values of *rho* result in higher capital requirements because of WWR.

```
pct = 99;
ec = prctile(totalLosses,pct) - mean(totalLosses);

figure;
plot(rhoValues,ec)
title('Economic Capital (99%) versus \rho')
xlabel('Rho');
ylabel('Economic Capital');
```



Final Remarks

This example implements a copula-based approach to WWR, following Garcia Cespedes et al. The methodology can efficiently reuse existing exposures and credit simulations, and the sensitivity to the market-credit correlation parameter can be efficiently computed and conveniently visualized for all correlation values.

The single-parameter copula approach presented here can be extended for a more thorough exploration of the WWR of a portfolio. For example, different types of copulas can be applied, and different criteria can be used to sort the exposure scenarios. Other extensions include simulating multiple systemic credit risk variables (a multi-factor model), or switching from a one-year to a multi-period framework to calculate measures such as credit value adjustment (CVA), as in Rosen and Saunders (see References on page 8-41).

References

1. Garcia Cespedes, J. C. "Effective Modeling of Wrong-Way Risk, Counterparty Credit Risk Capital, and Alpha in Basel II." *The Journal of Risk Model Validation*, Volume 4 / Number 1, pp. 71-98, Spring 2010.
2. Gupton, G., C. Finger, and M. Bathia. *CreditMetrics™ - Technical Document*. J.P. Morgan, New York, 1997.
3. Rosen, D., and D. Saunders. "CVA the Wrong Way." *Journal of Risk Management in Financial Institutions*. Vol. 5, No. 3, pp. 252-272, 2012.

Local Functions

```
function displayExpectedLosses(rhoValues, expectedLosses)
fprintf('               Expected Losses\n');
fprintf(' Rho    CP1      CP2      CP3     CP4       CP5\n');
fprintf('-------------------------------------------------\n');
for i = 1:numel(rhoValues)
    % Display expected loss
    fprintf('% .1f%9.2f%9.2f%9.2f%9.2f%9.2f', rhoValues(i), expectedLosses(:,i));
end
end
```

See Also

`cdsbootstrap` | `cdsprice` | `cdsspread` | `cdsrpv01`

Related Examples

- “First-to-Default Swaps” on page 8-18
- “Credit Default Swap Option” on page 8-27

More About

- “Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94

External Websites

- Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Bootstrapping a Default Probability Curve from Credit Default Swaps

This example shows how to bootstrap a default probability curve for CDS instruments.

Create a ratecurve Object for a Zero Curve

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2017,9,15);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2017
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Market CDS Spreads and a Vector of Market CDS Instruments

Define the market CDS spreads and use `fininstrument` to create a vector of market CDS instrument objects.

```
SpreadTimes = [1 2 3 4 5 7 10 20 30]';
Spread = [140 175 210 265 310 360 410 460 490]';
MarketDates = datemnth(Settle,12*SpreadTimes);

NumMarketInst = length(MarketDates);
ContractSpreadBP = 50.*ones(NumMarketInst,1);

MarketCDSInstruments(NumMarketInst,1) = fininstrument("cds", ...
    'ContractSpread', ContractSpreadBP(end), 'Maturity', MarketDates(end));
for k = 1:NumMarketInst
    MarketCDSInstruments(k,1) = fininstrument("cds", ...
        'ContractSpread', ContractSpreadBP(k), 'Maturity', MarketDates(k));
end
MarketCDSInstruments
```

```
MarketCDSInstruments=9×1 object
  9x1 CDS array with properties:

    ContractSpread
    Maturity
    Period
    Basis
```

```
    RecoveryRate
    BusinessDayConvention
    Holidays
    PayAccruedPremium
    Notional
    Name
```

Bootstrap a Default Probability Curve

Use `defprobstrip`, `hazardrates`,and `survprobs` to analyse a default probability curve for the market `CDS` instruments.

```
DefaultProbCurve = defprobstrip(ZeroCurve, MarketCDSInstruments, Spread)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                 Settle: 15-Sep-2017
                  Basis: 2
                  Dates: [9x1 datetime]
    DefaultProbabilities: [9x1 double]
```

```
HazardRates = hazardrates(DefaultProbCurve)
```

```
HazardRates = 9×1

    0.0233
    0.0352
    0.0474
    0.0751
    0.0879
    0.0887
    0.1023
    0.1059
    0.2271
```

```
SurvivalProbabilities = survprobs(DefaultProbCurve, MarketDates)
```

```
SurvivalProbabilities = 9×1

    0.9766
    0.9424
    0.8981
    0.8322
    0.7612
    0.6358
    0.4658
    0.1590
    0.0159
```

See Also

Functions
`CDS` | `finmodel` | `finpricer`

Related Examples

- “Price Multiple CDS Option Instruments Using CDS Black Model and CDS Black Pricer” on page 8-46

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53

Bootstrap Default Probability Curve from Market CDS Instruments

This example shows how to use `defprobstrip` to bootstrap a `defprobcurve` object based on market CDS instruments.

Create `ratecurve` Object for Zero-Rate Curve

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2017,9,15);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates);
```

Market CDS Spreads and Vector of Market CDS Instruments

Define the market CDS spreads and use `fininstrument` to create a vector of market CDS instrument objects.

```
SpreadTimes = [1 2 3 4 5 7 10 20 30]';
Spread = [140 175 210 265 310 360 410 460 490]';
MarketDates = datemnth(Settle,12*SpreadTimes);

NumMarketInst = length(MarketDates);
ContractSpreadBP = zeros(NumMarketInst,1);

MarketCDSInstruments(NumMarketInst,1) = fininstrument("cds", ...
      'ContractSpread', ContractSpreadBP(end), 'Maturity', MarketDates(end));
  for k = 1:NumMarketInst
      MarketCDSInstruments(k,1) = fininstrument("cds", ...
          'ContractSpread', ContractSpreadBP(k), 'Maturity', MarketDates(k));
  end
```

Use `defprobstrip` to create a `defprobcurve` object.

```
DefaultProbCurve = defprobstrip(ZeroCurve,MarketCDSInstruments, Spread)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                  Settle: 15-Sep-2017
                   Basis: 2
                   Dates: [9x1 datetime]
    DefaultProbabilities: [9x1 double]
```

Price Multiple CDS Option Instruments Using CDS Black Model and CDS Black Pricer

This example shows the workflow to price multiple `CDSOption` instruments using a `CDSBlack` model and a `CDSBlack` pricer.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2021,9,20);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero", Settle, ZeroDates ,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 20-Sep-2021
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `defprobcurve` Object

Create a `defprobcurve` object using `defprobcurve`.

```
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle, ProbDates, DefaultProbabilities)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                 Settle: 20-Sep-2021
                  Basis: 2
                  Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Create `CDS` Instrument Object

Use `fininstrument` to create an underlying `CDS` instrument object.

```
ContractSpreadBP = 0; % Contractual spread is determined on ExerciseDate
CDS = fininstrument("CDS",'Maturity',datetime(2027,9,20),'ContractSpread',ContractSpreadBP)
```

```
CDS =
  CDS with properties:
```

```
          ContractSpread: 0
                Maturity: 20-Sep-2027
                  Period: 4
                   Basis: 2
            RecoveryRate: 0.4000
    BusinessDayConvention: "actual"
                Holidays: NaT
        PayAccruedPremium: 1
                Notional: 10000000
                    Name: ""
```

Create CDSOption Instrument Objects

Use `fininstrument` to create multiple `CDSOption` instrument objects.

```
ExerciseDate = datetime(2021, 12, 20);
Strikes = [30:2:90]';
PayerCDSOptions = fininstrument("CDSOption",'Strike',Strikes,'ExerciseDate',ExerciseDate,'Option
```

```
PayerCDSOptions=31×1 object
  16x1 CDSOption array with properties:

    OptionType
    Strike
    Knockout
    AdjustedForwardSpread
    ExerciseDate
    CDS
    Name
      ⋮
```

```
ReceiverCDSOptions = fininstrument("CDSOption",'Strike',Strikes,'ExerciseDate',ExerciseDate,'Opti
```

```
ReceiverCDSOptions=31×1 object
  16x1 CDSOption array with properties:

    OptionType
    Strike
    Knockout
    AdjustedForwardSpread
    ExerciseDate
    CDS
    Name
      ⋮
```

Price CDSOption Instruments

Assuming a flat volatility structure across strikes, first use `finmodel` to create a `CDSBlack` model object. Then use `finpricer` to create a `CDSBlack` pricer object. Use `price` to compute the prices for the `CDSOption` instruments.

```
SpreadVolatility = 0.3;
CDSOptionModel = finmodel("CDSBlack",'SpreadVolatility',SpreadVolatility)
```

```
CDSOptionModel =
  CDSBlack with properties:
```

```
    SpreadVolatility: 0.3000
```

```
CDSOptionpricer = finpricer("analytic",'Model',CDSOptionModel,'DiscountCurve',ZeroCurve,'DefaultP
```

```
CDSOptionpricer =
  CDSBlack with properties:

                     Model: [1x1 finmodel.CDSBlack]
             DiscountCurve: [1x1 ratecurve]
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

```
PayerPrices = price(CDSOptionpricer,PayerCDSOptions)
```

```
PayerPrices = 31×1

  171.7269
  160.6802
  149.6346
  138.5931
  127.5648
  116.5716
  105.6576
   94.8983
   84.4061
   74.3266
      ⋮
```

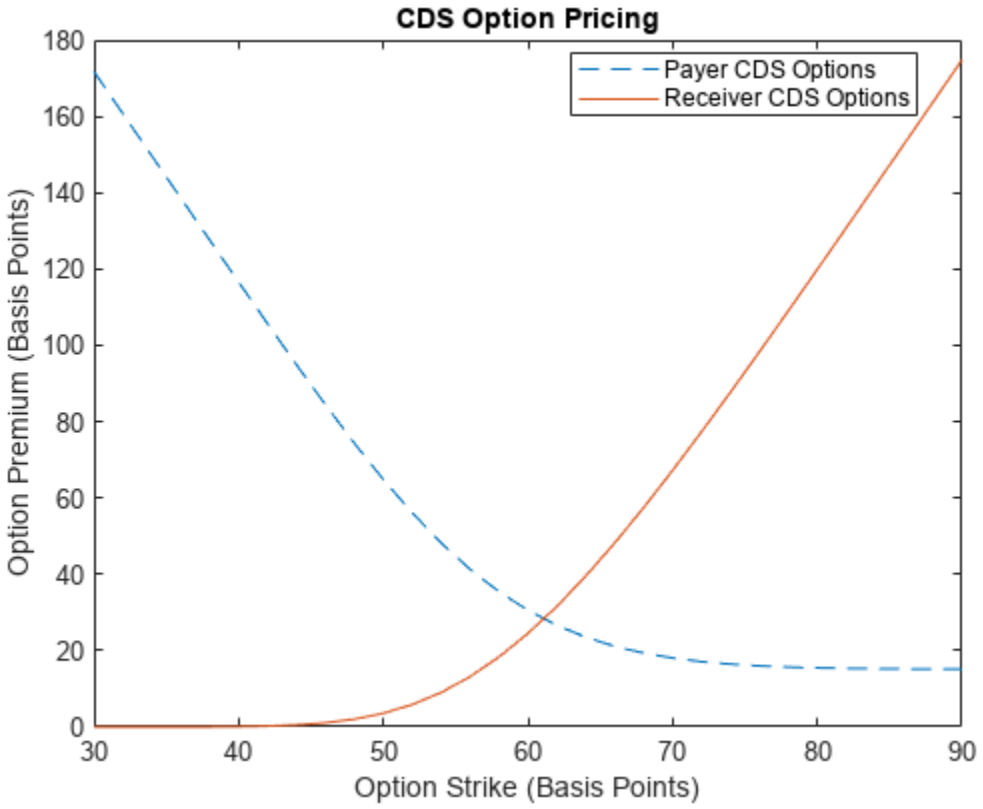
```
ReceiverPrices = price(CDSOptionpricer,ReceiverCDSOptions)
```

```
ReceiverPrices = 31×1

    0.0000
    0.0003
    0.0016
    0.0070
    0.0256
    0.0794
    0.2123
    0.4999
    1.0547
    2.0221
      ⋮
```

Plot CDS Option Prices

Plot the payer and receiver CDS option prices.

```
figure;
plot(Strikes, PayerPrices, '--', Strikes, ReceiverPrices)
title('CDS Option Pricing')
xlabel('Option Strike (Basis Points)')
ylabel('Option Premium (Basis Points)')
legend('Payer CDS Options','Receiver CDS Options','Location','best')
```



See Also

Functions
`CDS` | `CDSOption` | `finmodel` | `finpricer`

Related Examples

- “Bootstrapping a Default Probability Curve from Credit Default Swaps” on page 8-42

More About

- “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
- “Choose Instruments, Models, and Pricers” on page 1-53

Interest-Rate Curve Objects

Interest-Rate Curve Objects and Workflow

In this section...

Class Structure

Financial Instruments Toolbox class structure supports interest-rate curve objects. The class structure supports four classes.

Class Structure

Class Name	Description
`IRDataCurve`	Creates a representation of an interest-rate curve with dates and data. `IRDataCurve` is created directly by specifying dates and corresponding interest rates or discount factors, or you can bootstrap an `IRDataCurve` object from market data.
`IRFunctionCurve`	Creates a representation of an interest-rate curve with a function. `IRFunctionCurve` is created directly by specifying a function handle, or you can fit a function to market data using functions of the `IRFunctionCurve` object.
`IRBootstrapOptions`	The `IRBootstrapOptions` object lets you specify options relating to the bootstrapping of an `IRDataCurve` object.
`IRFitOptions`	The `IRFitOptions` object lets you specify options relating to the fitting process for an `IRFunctionCurve` object.

Workflow Using Interest-Rate Curve Objects

The supported workflow model for using interest-rate curve objects is:

1. Create an interest-rate curve based on an `IRDataCurve` object or an `IRFunctionCurve` object.
   - To create an `IRDataCurve` object:
     - Use vectors of dates and data with interpolation methods.
     - Use bootstrapping based on market instruments.

     For more information on creating an `IRDataCurve` object, see “Creating an IRDataCurve Object” on page 9-6.
   - To create an `IRFunctionCurve` object:
     - Specify a function handle.
     - Fit a function using the Nelson-Siegel model, Svensson model, or smoothing spline model.
     - Fit a custom function.
2. Use functions of the `IRDataCurve` or `IRFunctionCurve` objects to extract forward, zero, discount factor, or par yield curves for the interest-rate curve object.

3 Convert an interest-rate curve from an `IRDataCurve` or `IRFunctionCurve` object to a `RateSpec` structure. This `RateSpec` structure is identical to the `RateSpec` produced by the function `intenvset`. Using the `RateSpec` for an interest-rate curve object, you can then use Financial Instruments Toolbox functions to model an interest-rate structure and price. Alternatively, you can convert the `RateSpec` to a `ratecurve` object (see “Convert RateSpec to a ratecurve Object” on page 1-49) and then use the Financial Instruments Toolbox object-based framework for pricing instruments.

See Also

`ratecurve` | `parametercurve`

Related Examples

- “Creating Interest-Rate Curve Objects” on page 9-4

More About

- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Creating Interest-Rate Curve Objects

Depending on your data and purpose for analysis, you can create an interest-rate curve object by using an `IRDataCurve` or `IRFunctionCurve` object.

To create an `IRDataCurve` object, you can:

- Use `IRDataCurve` to create an `IRDataCurve` object using vector of dates and data with interpolation methods.
- Use the object function `bootstrap` using market instruments.

For more information on creating an `IRDataCurve` object, see “Creating an IRDataCurve Object” on page 9-6.

Using an `IRDataCurve` object, you can use the following functions to determine:

- Forward rate curve — `getForwardRates`
- Zero rate curve — `getZeroRates`
- Discount rate curve — `getDiscountFactors`
- Par yield curve — `getParYields`

Alternatively, to create an `IRFunctionCurve` object, you can:

- Use `IRFunctionCurve` to create an `IRFunctionCurve` object and directly specify a function handle.
- Use `IRFunctionCurve` object functions:
  - `fitNelsonSiegel` fits a “Fitting IRFunctionCurve Object Using Nelson-Siegel Method” on page 9-16 to market data for bonds.
  - `fitSvensson` fits a “Fitting IRFunctionCurve Object Using Svensson Method” on page 9-17 to market data for bonds.
  - `fitSmoothingSpline` fits a “Fitting IRFunctionCurve Object Using Smoothing Spline Method” on page 9-19 function to market data for bonds.
  - `fitFunction` custom fits an interest-rate curve object to market data for bonds.

Using an `IRFunctionCurve` object, you can use the following functions to determine:

- Forward rate curve — `getForwardRates`
- Zero rate curve — `getZeroRates`
- Discount rate curve — `getDiscountFactors`
- Par yield curve — `getParYields`

In addition, you can convert an `IRDataCurve` object or `IRFunctionCurve` object to a `RateSpec` structure. For more information, see “Converting an IRDataCurve or IRFunctionCurve Object” on page 9-30.

See Also

Related Examples

- “Creating an IRDataCurve Object” on page 9-6

More About

- “Interest-Rate Curve Objects and Workflow” on page 9-2

Creating an IRDataCurve Object

To create an `IRDataCurve` object, see the following options:

In this section...
“Use IRDataCurve with Dates and Data” on page 9-6
“Bootstrap IRDataCurve Based on Market Instruments” on page 9-7

Use IRDataCurve with Dates and Data

Use `IRDataCurve` with vectors of dates and data to create an interest-rate curve object. When constructing the `IRDataCurve` object, you can also use optional inputs to define how the interest-rate curve is constructed from the dates and data.

Example

In this example, you create the vectors for `Dates` and `Data` for an interest-rate curve.

```
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = daysadd(today,[360 2*360 3*360 5*360 7*360 10*360 20*360 30*360],1);
```

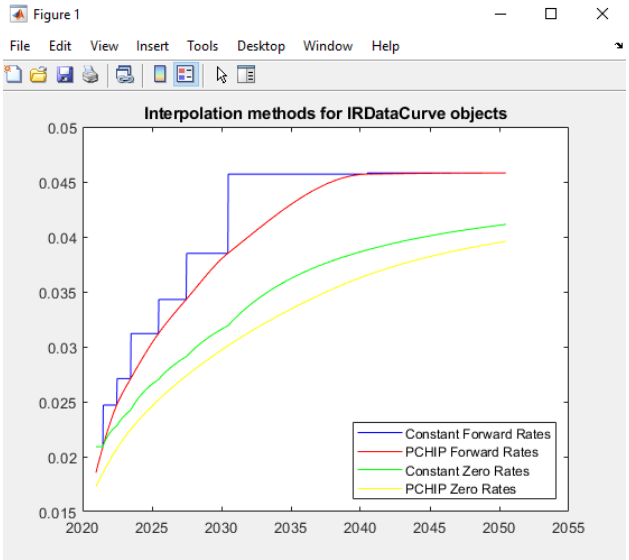
Use `IRDataCurve` to build interest-rate objects based on the `constant` and `pchip` interpolation methods.

```
irdc_const = IRDataCurve('Forward',today,Dates,Data,'InterpMethod','constant');
irdc_pchip = IRDataCurve('Forward',today,Dates,Data,'InterpMethod','pchip');
```

Plot the forward and zero rate curves for the two `IRDataCurve` objects based on `constant` and `pchip` interpolation methods.

```
PlottingDates = daysadd(today,180:10:360*30,1);
plot(PlottingDates, getForwardRates(irdc_const, PlottingDates),'b')
hold on
plot(PlottingDates, getForwardRates(irdc_pchip, PlottingDates),'r')
plot(PlottingDates, getZeroRates(irdc_const, PlottingDates),'g')
plot(PlottingDates, getZeroRates(irdc_pchip, PlottingDates),'yellow')
legend({'Constant Forward Rates','PCHIP Forward Rates','Constant Zero Rates',...
'PCHIP Zero Rates'},'location','SouthEast')
title('Interpolation methods for IRDataCurve objects')
datetick
```

The plot demonstrates the relationship of the forward and zero rate curves.



Bootstrap IRDataCurve Based on Market Instruments

Use the bootstrapping function, based on market instruments, to create an interest-rate curve object. When bootstrapping, you also have the option to define a range of interpolation methods (`linear`, `spline`, `constant`, and `pchip`).

Example 1

In this example, you bootstrap a swap curve from deposits, Eurodollar Futures and swaps. The input market data for this example is hard-coded and specified as two cell arrays of data; one cell array indicates the type of instrument and the other contains the `Settle`, `Maturity` values and a market quote for the instrument. For deposits and swaps, the quote is a rate; for the EuroDollar futures, the quote is a price. Although bonds are not used in this example, a bond would also be quoted with a price.

```
InstrumentTypes = {'Deposit';'Deposit';'Deposit';'Deposit';'Deposit'; ...
    'Futures';'Futures'; ...
    'Futures';'Futures';'Futures'; ...
    'Futures';'Futures';'Futures'; ...
    'Futures';'Futures';'Futures'; ...
    'Futures';'Futures';'Futures'; ...
    'Swap';'Swap';'Swap';'Swap';'Swap';'Swap';'Swap'};

Instruments = [datenum('08/10/2007'),datenum('08/17/2007'),.0532063; ...
    datenum('08/10/2007'),datenum('08/24/2007'),.0532000; ...
    datenum('08/10/2007'),datenum('09/17/2007'),.0532000; ...
    datenum('08/10/2007'),datenum('10/17/2007'),.0534000; ...
    datenum('08/10/2007'),datenum('11/17/2007'),.0535866; ...
    datenum('08/08/2007'),datenum('19-Dec-2007'),9485; ...
    datenum('08/08/2007'),datenum('19-Mar-2008'),9502; ...
    datenum('08/08/2007'),datenum('18-Jun-2008'),9509.5; ...
    datenum('08/08/2007'),datenum('17-Sep-2008'),9509; ...
    datenum('08/08/2007'),datenum('17-Dec-2008'),9505.5; ...
    datenum('08/08/2007'),datenum('18-Mar-2009'),9501; ...
    datenum('08/08/2007'),datenum('17-Jun-2009'),9494.5; ...
    datenum('08/08/2007'),datenum('16-Sep-2009'),9489; ...
    datenum('08/08/2007'),datenum('16-Dec-2009'),9481.5; ...
    datenum('08/08/2007'),datenum('17-Mar-2010'),9478; ...
    datenum('08/08/2007'),datenum('16-Jun-2010'),9474; ...
    datenum('08/08/2007'),datenum('15-Sep-2010'),9469.5; ...
```

```
    datenum('08/08/2007'),datenum('15-Dec-2010'),9464.5; ...
    datenum('08/08/2007'),datenum('16-Mar-2011'),9462.5; ...
    datenum('08/08/2007'),datenum('15-Jun-2011'),9456.5; ...
    datenum('08/08/2007'),datenum('21-Sep-2011'),9454; ...
    datenum('08/08/2007'),datenum('21-Dec-2011'),9449.5; ...
    datenum('08/08/2007'),datenum('08/08/2014'),.0530; ...
    datenum('08/08/2007'),datenum('08/08/2017'),.0545; ...
    datenum('08/08/2007'),datenum('08/08/2019'),.0551; ...
    datenum('08/08/2007'),datenum('08/08/2022'),.0559; ...
    datenum('08/08/2007'),datenum('08/08/2027'),.0565; ...
    datenum('08/08/2007'),datenum('08/08/2032'),.0566; ...
    datenum('08/08/2007'),datenum('08/08/2037'),.0566];
```

`bootstrap` is called as a function of the `IRDataCurve` object. Inputs to this function include the curve `Type` (`zero` or `forward`), `Settle` date, `InstrumentTypes`, and `Instrument` data. The `bootstrap` function also supports optional arguments, including an interpolation method, compounding, basis, and an options structure for bootstrapping. For example, you are passing in an `IRBootstrapOptions` object which includes information for the `ConvexityAdjustment` to forward rates.

```
IRsigma = .01;
CurveSettle = datenum('08/10/2007');
bootModel = IRDataCurve.bootstrap('Forward', CurveSettle, ...
InstrumentTypes, Instruments,'InterpMethod','pchip',...
'Compounding',-1,'IRBootstrapOptions',...
IRBootstrapOptions('ConvexityAdjustment',@(t) .5*IRsigma^2.*t.^2))
```

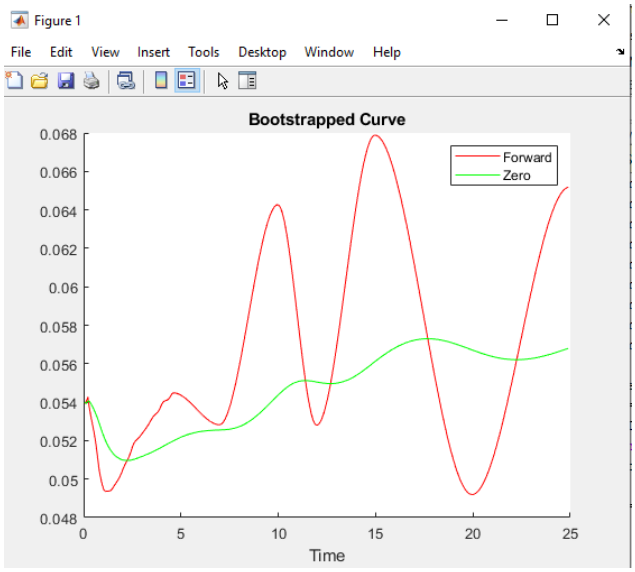
```
bootModel =

IRDataCurve

             Type: Forward
           Settle: 733264 (10-Aug-2007)
      Compounding: -1
            Basis: 0 (actual/actual)
     InterpMethod: pchip
            Dates: [29x1 double]
             Data: [29x1 double]
```

The `bootstrap` function uses an Optimization Toolbox function to solve for any bootstrapped rates.

Plot the forward and zero curves.

```
PlottingDates = (CurveSettle+20:30:CurveSettle+365*25)';
TimeToMaturity = yearfrac(CurveSettle,PlottingDates);
BootstrappedForwardRates = getForwardRates(bootModel, PlottingDates);
BootstrappedZeroRates = getZeroRates(bootModel, PlottingDates);

figure
hold on
plot(TimeToMaturity,BootstrappedForwardRates,'r')
plot(TimeToMaturity,BootstrappedZeroRates,'g')
title('Bootstrapped Curve')
xlabel('Time')
legend({'Forward','Zero'})
```

The plot demonstrates the forward and zero rate curves for the market data.



Example 2

In this example, you bootstrap a swap curve from deposits, Eurodollar futures, and swaps. The input market data for this example is hard-coded and specified as two cell arrays of data; one cell array indicates the type of instrument and the other cell array contains the `Settle`, `Maturity` values and a market quote for the instrument. This example of bootstrapping also demonstrates the use of an `InstrumentBasis` for each `Instrument` type.

```
InstrumentTypes = {'Deposit';'Deposit';...
'Futures';'Futures';'Futures';'Futures';'Futures';...
'Swap';'Swap';'Swap';'Swap';};

Instruments = [datenum('08/10/2007'),datenum('09/17/2007'),.0532000; ...
datenum('08/10/2007'),datenum('11/17/2007'),.0535866; ...
datenum('08/08/2007'),datenum('19-Dec-2007'),9485; ...
datenum('08/08/2007'),datenum('19-Mar-2008'),9502; ...
datenum('08/08/2007'),datenum('18-Jun-2008'),9509.5; ...
datenum('08/08/2007'),datenum('17-Sep-2008'),9509; ...
datenum('08/08/2007'),datenum('17-Dec-2008'),9505.5; ...
datenum('08/08/2007'),datenum('18-Mar-2009'),9501; ...
datenum('08/08/2007'),datenum('08/08/2014'),.0530; ...
datenum('08/08/2007'),datenum('08/08/2019'),.0551; ...
datenum('08/08/2007'),datenum('08/08/2027'),.0565; ...
datenum('08/08/2007'),datenum('08/08/2037'),.0566];

CurveSettle = datenum('08/10/2007');
```

The `bootstrap` function is called as a function of the `IRBootstrapOptions` object. Inputs to the `bootstrap` function include the curve `Type` (`zero` or `forward`), `Settle` date, `InstrumentTypes`, and `Instrument` data. The `bootstrap` function also supports optional arguments, including an interpolation method, compounding, basis, and an options structure for bootstrapping. In this example, you are passing an additional `Basis` value for each instrument type.

```
bootModel=IRDataCurve.bootstrap('Forward',CurveSettle,InstrumentTypes, ...
Instruments,'InterpMethod','pchip','InstrumentBasis',[repmat(2,8,1);repmat(0,4,1)])
```

```
bootModel =

    IRDataCurve

           Type: Forward
```

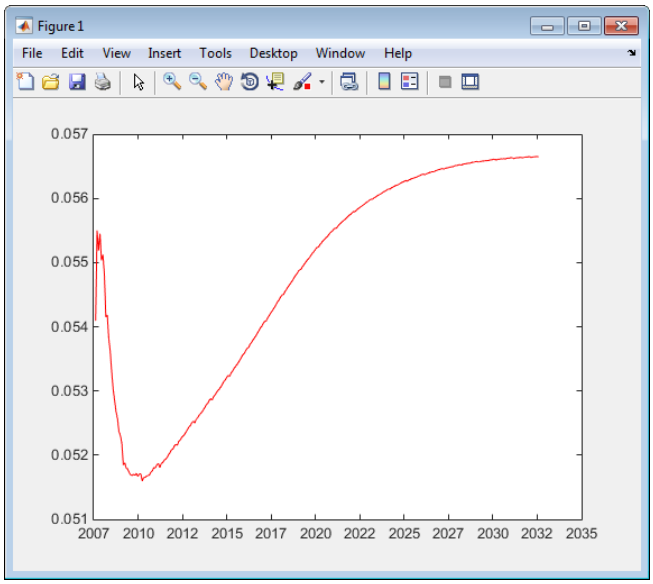
```
      Settle: 733264 (10-Aug-2007)
 Compounding: 2
       Basis: 0 (actual/actual)
InterpMethod: pchip
       Dates: [12x1 double]
        Data: [12x1 double]
```

The `bootstrap` function uses an Optimization Toolbox function to solve for any bootstrapped rates.

Plot the par yields curve using the `getParYields` function.

```
PlottingDates = (datenum('08/11/2007'):30:CurveSettle+365*25)';
plot(PlottingDates, getParYields(bootModel, PlottingDates),'r')
datetick
```

The plot demonstrates the par yields curve for the market data.



See Also

`IRBootstrapOptions` | `IRDataCurve` | `IRFunctionCurve` | `IRFitOptions`

Related Examples

- “Creating Interest-Rate Curve Objects” on page 9-4
- “Bootstrapping a Swap Curve” on page 2-141
- “Dual Curve Bootstrapping” on page 9-12
- “Creating an IRFunctionCurve Object” on page 9-16
- “Fitting Interest-Rate Curve Functions” on page 2-144

More About

- “Interest-Rate Curve Objects and Workflow” on page 9-2
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

External Websites

- Calibration and Simulation Best Practices: Multifactor Interest Rate Models for Risk Applications (30 min 00 sec)

Dual Curve Bootstrapping

This example shows how to bootstrap a forward curve using a different curve for discounting.

Define the Data

Data is needed for both the forward and discount curve. For this particular example, it is assumed that the data is provided for EONIA (the discount curve) and EURIBOR (the forward curve). However, this approach can be used in any case where the curve to be built is different than the curve used for discounting cash flows. While the data in this example is hardcoded, it could also be imported into MATLAB with Datafeed Toolbox™ or Database Toolbox™.

```
Settle = datenum('20-Aug-2013');

% Deposit data
EONIADepositRates = [.0007 .00067]';
EONIADepositMat = datenum({'3-Sep-2013','20-Sep-2013'});
EONIADepositBasis = 2; % act/360
EONIADepositPeriod = 0;

% FRA
EONIAFRARates = [.00025 .0003 .00043 .00054]';
EONIAFRAStartDate = datenum({'11-Sep-2013','9-Oct-2013','13-Nov-2013','11-Dec-2013'});
EONIAFRAEndDate = datenum({'9-Oct-2013','13-Nov-2013','11-Dec-2013','11-Jan-2014'});
EONIAFRABasis = 2; % act/360
EONIAFRAPeriod = 0;

% Swap data
EONIASwapRates = [.0003 .001 .002 .004 .008 .012 .0155 .018 .0193 .02]';
EONIASwapMat = datemnth(Settle,12*[2:5 7 10 15 20 25 30]');
EONIASwapBasis = 5; % 30/360 ISDA
EONIASwapPeriod = 1;

% EURIBOR Deposit data
EURIBORDepositRates = [.0022 .0021 .002 .0019]';
EURIBORDepositMat = datenum({'3-Sep-2013','20-Sep-2013','21-Oct-2013','20-Nov-2013'});
EURIBORDepositBasis = 2; % act/360
EURIBORDepositPeriod = 0;

% EURIBOR Futures
EURIBORFRARates = [9982 9978 9976 9975]';
EURIBORFRAStartDate = datenum({'18-Dec-2013','19-Mar-2014','18-Jun-2014','17-Sep-2014'});
EURIBORFRAEndDate = datenum({'18-Mar-2014','19-Jun-2014','18-Sep-2014','17-Dec-2014'});
EURIBORFRABasis = 2; % act/360
EURIBORFRAPeriod = 4;

% EURIBOR Swap data
EURIBORSwapRates = [.0026 .0044 .0062 .0082 .012 .015 .018 .02 .021 .0215]';
EURIBORSwapMat = datemnth(Settle,12*[2:5 7 10 15 20 25 30]');
EURIBORSwapBasis = 5; % 30/360 ISDA
EURIBORSwapPeriod = 1;
```

Create an EONIA Discount Curve

Build the EONIA curve. This is essentially the same as the single curve case.

```
CurveType = 'zero';
CurveCompounding = 1;
CurveBasis = 3; % act/365

nEONIADeposits = length(EONIADepositMat);
nEONIAFRA = length(EONIAFRAEndDate);
nEONIASwaps = length(EONIASwapMat);

EONIAInstrumentTypes = [repmat({'deposit'},nEONIADeposits,1);
    repmat({'fra'},nEONIAFRA,1);repmat({'swap'},nEONIASwaps,1)];

EONIAPeriod = [repmat(EONIADepositPeriod,nEONIADeposits,1);
    repmat(EONIAFRAPeriod,nEONIAFRA,1);repmat(EONIASwapPeriod,nEONIASwaps,1)];

EONIABasis = [repmat(EONIADepositBasis,nEONIADeposits,1);
    repmat(EONIAFRABasis,nEONIAFRA,1);repmat(EONIASwapBasis,nEONIASwaps,1)];

EONIAInstrumentData = [[repmat(Settle,[nEONIADeposits 1]);EONIAFRAStartDate;repmat(Settle,[nEONIASwaps 1])] ...
```

```
    [EONIADepositMat;EONIAFRAEndDate;EONIASwapMat] ...
    [EONIADepositRates;EONIAFRARates;EONIASwapRates]];

EONIACurve = IRDataCurve.bootstrap(CurveType,Settle,EONIAInstrumentTypes,...
    EONIAInstrumentData,'Compounding',CurveCompounding,'Basis',CurveBasis,...
    'InstrumentPeriod',EONIAPeriod,'InstrumentBasis',EONIABasis)
```

```
EONIACurve =

             Type: zero
           Settle: 735466 (20-Aug-2013)
      Compounding: 1
            Basis: 3 (actual/365)
     InterpMethod: linear
            Dates: [16x1 double]
             Data: [16x1 double]
```

Create an EURIBOR Forward Curve

The EURIBOR forward curve is built first using a single curve approach.

```
nEURIBORDeposits = length(EURIBORDepositMat);
nEURIBORFRA = length(EURIBORFRAEndDate);
nEURIBORSwaps = length(EURIBORSwapMat);

EURIBORInstrumentTypes = [repmat({'deposit'},nEURIBORDeposits,1);
    repmat({'futures'},nEURIBORFRA,1);repmat({'swap'},nEURIBORSwaps,1)];

EURIBORPeriod = [repmat(EURIBORDepositPeriod,nEURIBORDeposits,1);
    repmat(EURIBORFRAPeriod,nEURIBORFRA,1);repmat(EURIBORSwapPeriod,nEURIBORSwaps,1)];

EURIBORBasis = [repmat(EURIBORDepositBasis,nEURIBORDeposits,1);
    repmat(EURIBORFRABasis,nEURIBORFRA,1);repmat(EURIBORSwapBasis,nEURIBORSwaps,1)];

EURIBORInstrumentData = [repmat(Settle,size(EURIBORInstrumentTypes)) ...
    [EURIBORDepositMat;EURIBORFRAEndDate;EURIBORSwapMat] ...
    [EURIBORDepositRates;EURIBORFRARates;EURIBORSwapRates]];

EURIBORCurve_Single = IRDataCurve.bootstrap(CurveType,Settle,EURIBORInstrumentTypes,...
    EURIBORInstrumentData,'Compounding',CurveCompounding,'Basis',CurveBasis,...
    'InstrumentPeriod',EURIBORPeriod,'InstrumentBasis',EURIBORBasis)
```

```
EURIBORCurve_Single =

             Type: zero
           Settle: 735466 (20-Aug-2013)
      Compounding: 1
            Basis: 3 (actual/365)
     InterpMethod: linear
            Dates: [18x1 double]
             Data: [18x1 double]
```

Build the EURIBOR Curve with the EONIA Curve

Next, build a curve using the EONIA curve as a discounting curve. To do this, specify the EONIA curve as an optional input argument.

```
EURIBORCurve = IRDataCurve.bootstrap(CurveType,Settle,EURIBORInstrumentTypes,...
    EURIBORInstrumentData,'DiscountCurve',EONIACurve,'Compounding',...
    CurveCompounding,'Basis',CurveBasis,'InstrumentPeriod',EURIBORPeriod,...
    'InstrumentBasis',EURIBORBasis)
```

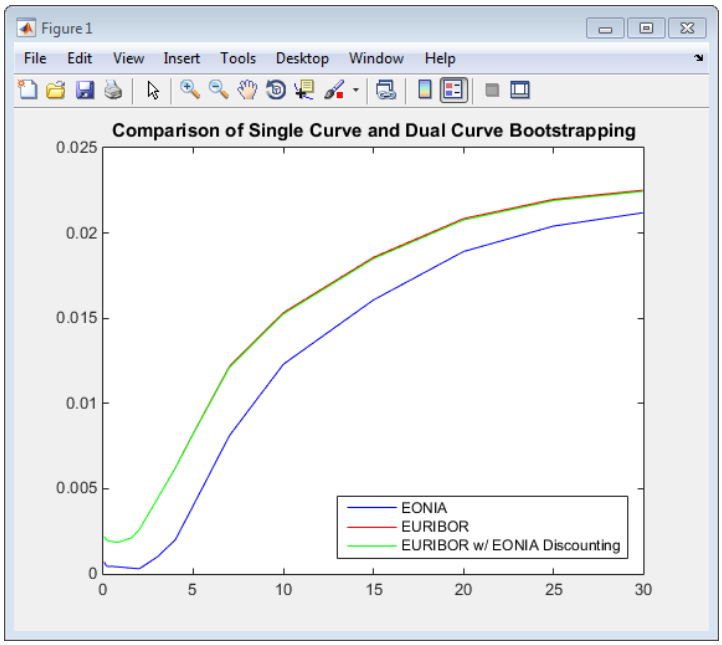
```
EURIBORCurve =

             Type: zero
           Settle: 735466 (20-Aug-2013)
      Compounding: 1
            Basis: 3 (actual/365)
```

```
     InterpMethod: linear
            Dates: [18x1 double]
             Data: [18x1 double]
```

Plot the Results

Plot the results to compare the curves.

```
PlottingDates = (Settle+20:30:Settle+365*30)';
TimeToMaturity = yearfrac(Settle,PlottingDates);

figure
plot(TimeToMaturity, getZeroRates(EONIACurve, PlottingDates),'b')
hold on
plot(TimeToMaturity, getZeroRates(EURIBORCurve_Single, PlottingDates),'r')
plot(TimeToMaturity, getZeroRates(EURIBORCurve, PlottingDates),'g')
title('Comparison of Single Curve and Dual Curve Bootstrapping')
legend({'EONIA','EURIBOR','EURIBOR w/ EONIA Discounting'},'location','southeast')
```



As expected, the difference between the two different EURIBOR curves is small but nontrivial.

Bibliography

This example draws from the following papers and journal articles:

[1] Ametrano, F, and Bianchetti, M. *Everything You Always Wanted to Know About Multiple Interest Rate Curve Bootstrapping but Were Afraid to Ask*. (April 2, 2013), available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2219548.

[2] Bianchetti, M. *Two Curves, One Price*. Risk Magazine, pages 74–80, August 2010.

[3] Fujii, M, Shimada, Y, Takahashi, A. *A Note on Construction of Multiple Swap Curves with and without Collateral*. (January 2, 2010), CARF Working Paper Series No. CARF-F-154, available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1440633.

[4] Mercurio, Fabio. *Interest Rates and The Credit Crunch: New Formulas and Market Models*. (February 5, 2009), Bloomberg Portfolio Research Paper No. 2010-01-FRONTIERS.

[5] Nashikkar, A. *Understanding OIS Discounting.*, Barclays Capital Interest Rate Strategy, February 24, 2011.

See Also

`IRDataCurve` | `bootstrap` | `floatbyzero` | `swapbyzero` | `getZeroRates`

Related Examples

- “Creating Interest-Rate Curve Objects” on page 9-4
- “Bootstrapping a Swap Curve” on page 2-141
- “Fitting Interest-Rate Curve Functions” on page 2-144

More About

- “Interest-Rate Curve Objects and Workflow” on page 9-2
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

External Websites

- Calibration and Simulation Best Practices: Multifactor Interest Rate Models for Risk Applications (30 min 00 sec)

Creating an IRFunctionCurve Object

To create an `IRFunctionCurve` object, see the following options:

Fitting IRFunctionCurve Object Using a Function Handle

You can use `IRFunctionCurve` with a MATLAB function handle to define an interest-rate curve. For more information on defining a function handle, see the MATLAB Programming Fundamentals documentation.

Example

This example uses a `FunctionHandle` argument with a value `@(t) t.^2` to create an interest-rate curve.

```
rr = IRFunctionCurve('Zero',today,@(t) t.^2)
```

```
rr =

  Properties:
    FunctionHandle: @(t)t.^2
              Type: 'Zero'
            Settle: 733600
       Compounding: 2
             Basis: 0
```

Fitting IRFunctionCurve Object Using Nelson-Siegel Method

Use the function, `fitNelsonSiegel`, for the Nelson-Siegel model that fits the empirical form of the yield curve with a prespecified functional form of the spot rates which is a function of the time to maturity of the bonds.

The Nelson-Siegel model represents a dynamic three-factor model: level, slope, and curvature. However, the Nelson-Siegel factors are unobserved, or latent, which allows for measurement error, and the associated loadings have economic restrictions (forward rates are always positive, and the discount factor approaches zero as maturity increases). For more information, see “Zero-coupon yield curves: technical documentation,” *BIS Papers*, Bank for International Settlements, Number 25, October 2005.

Example

This example uses `IRFunctionCurve` to model the default-free term structure of interest rates in the United Kingdom.

Load the data.

```
load ukdata20080430
```

Convert repo rates to be equivalent zero-coupon bonds.

```
RepoCouponRate = repmat(0,size(RepoRates));
RepoPrice = bndprice(RepoRates, RepoCouponRate, RepoSettle, RepoMaturity);
```

Aggregate the data.

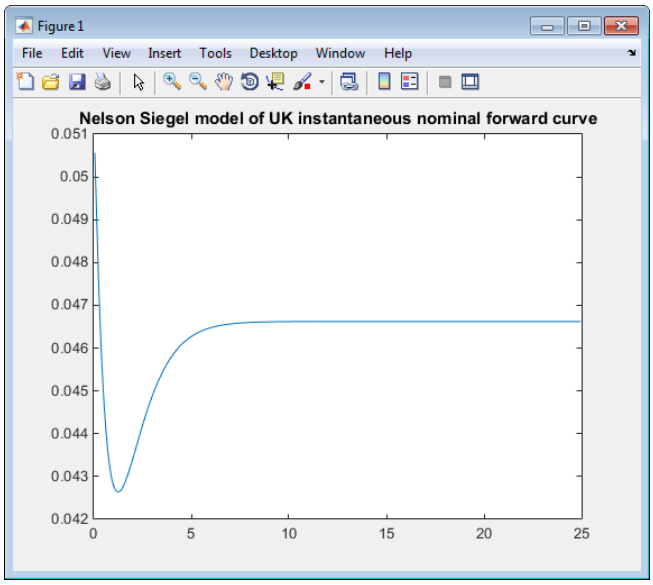
```
Settle = [RepoSettle;BondSettle];
Maturity = [RepoMaturity;BondMaturity];
CleanPrice = [RepoPrice;BondCleanPrice];
CouponRate = [RepoCouponRate;BondCouponRate];
Instruments = [Settle Maturity CleanPrice CouponRate];
InstrumentPeriod = [repmat(0,6,1);repmat(2,31,1)];
CurveSettle = datenum('30-Apr-2008');
```

The `IRFunctionCurve` object provides the capability to fit a Nelson-Siegel curve to observed market data with the `fitNelsonSiegel` function. The fitting is done by calling the function `lsqnonlin`. The `fitNelsonSiegel` function has required inputs of `Type`, `Settle`, and a matrix of instrument data.

```
NSModel = IRFunctionCurve.fitNelsonSiegel('Zero',CurveSettle,...
Instruments,'Compounding',-1,'InstrumentPeriod',InstrumentPeriod);
```

Plot the Nelson-Siegel interest-rate curve for forward rates.

```
PlottingDates = CurveSettle+20:30:CurveSettle+365*25;
TimeToMaturity = yearfrac(CurveSettle,PlottingDates);
NSForwardRates = getForwardRates(NSModel, PlottingDates);
plot(TimeToMaturity,NSForwardRates)
title('Nelson Siegel model of UK instantaneous nominal forward curve')
```



Fitting IRFunctionCurve Object Using Svensson Method

Use the function, `fitSvensson`, for the Svensson model to improve the flexibility of the curves and the fit for a Nelson-Siegel model. In 1994, Svensson extended Nelson and Siegel's function by adding a further term that allows for a second "hump." The extra precision is achieved at the cost of adding two more parameters, $\beta 3$ and $\tau 2$, which have to be estimated.

Example

In this example of using the `fitSvensson` function, an `IRFitOptions` structure, previously defined using `IRFitOptions`, is used. Thus, you must specify `FitType`, `InitialGuess`, `UpperBound`, `LowerBound`, and the `OptOptions` optimization parameters for `lsqnonlin`.

Load the data.

```
load ukdata20080430
```

Convert repo rates to be equivalent zero coupon bonds.

```
RepoCouponRate = repmat(0,size(RepoRates));
RepoPrice = bndprice(RepoRates, RepoCouponRate, RepoSettle, RepoMaturity);
```

Aggregate the data.

```
Settle = [RepoSettle;BondSettle];
Maturity = [RepoMaturity;BondMaturity];
CleanPrice = [RepoPrice;BondCleanPrice];
CouponRate = [RepoCouponRate;BondCouponRate];
Instruments = [Settle Maturity CleanPrice CouponRate];
InstrumentPeriod = [repmat(0,6,1);repmat(2,31,1)];
CurveSettle = datenum('30-Apr-2008');
```

Define `OptOptions` for `IRFitOptions`.

```
OptOptions = optimoptions('lsqnonlin','MaxFunEvals',1000);
fIRFitOptions = IRFitOptions([5.82 -2.55 -.87 0.45 3.9 0.44],...
'FitType','durationweightedprice','OptOptions',OptOptions,...
'LowerBound',[0 -Inf -Inf -Inf 0 0],'UpperBound',[Inf Inf Inf Inf Inf Inf]);
```

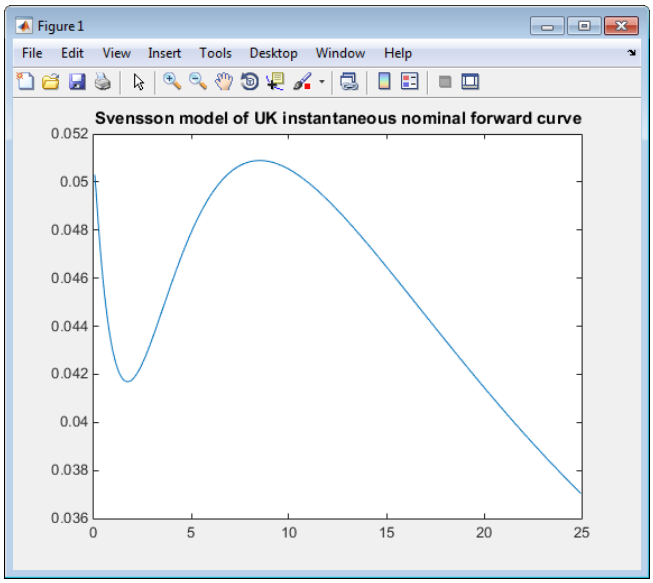
Fit the interest-rate curve using a Svensson model.

```
SvenssonModel = IRFunctionCurve.fitSvensson('Zero',CurveSettle,...
Instruments,'IRFitOptions', fIRFitOptions, 'Compounding',-1,...
'InstrumentPeriod',InstrumentPeriod)

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.

SvenssonModel =

             Type: Zero
           Settle: 733528 (30-Apr-2008)
      Compounding: -1
            Basis: 0 (actual/actual)
```

The status message, output from `lsqnonlin`, indicates that the optimization to find parameters for the Svensson equation terminated successfully.

Plot the Svensson interest-rate curve for forward rates.

```
PlottingDates = CurveSettle+20:30:CurveSettle+365*25;
TimeToMaturity = yearfrac(CurveSettle,PlottingDates);
SvenssonForwardRates = getForwardRates(SvenssonModel, PlottingDates);
plot(TimeToMaturity,SvenssonForwardRates)
title('Svensson model of UK instantaneous nominal forward curve')
```



Fitting IRFunctionCurve Object Using Smoothing Spline Method

Use the function, `fitSmoothingSpline`, to model the term structure with a spline, specifically, the term structure represents the forward curve with a cubic spline.

Note You must have a license for Curve Fitting Toolbox software to use the `fitSmoothingSpline` function.

Example

The `IRFunctionCurve` object is used to fit a smoothing spline representation of the forward curve with a penalty function. Required inputs for `fitSmoothingSpline` are `Type`, `Settle`, the matrix of `Instruments`, and `Lambdafun`, a function handle containing the penalty function

Load the data.

```
load ukdata20080430
```

Convert repo rates to be equivalent zero coupon bonds.

```
RepoCouponRate = repmat(0,size(RepoRates));
RepoPrice = bndprice(RepoRates, RepoCouponRate, RepoSettle, RepoMaturity);
```

Aggregate the data.

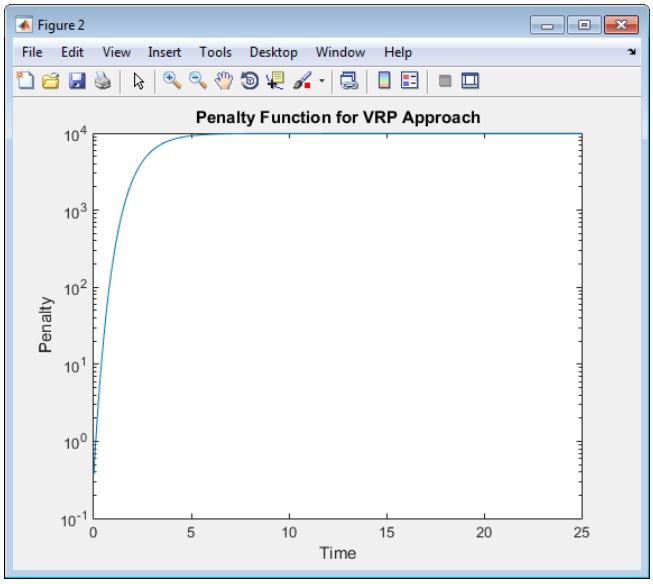
```
Settle = [RepoSettle;BondSettle];
Maturity = [RepoMaturity;BondMaturity];
CleanPrice = [RepoPrice;BondCleanPrice];
CouponRate = [RepoCouponRate;BondCouponRate];
Instruments = [Settle Maturity CleanPrice CouponRate];
InstrumentPeriod = [repmat(0,6,1);repmat(2,31,1)];
CurveSettle = datenum('30-Apr-2008');
```

Choose parameters for `Lambdafun`.

```
L = 9.2;
S = -1;
mu = 1;
```

Define the `Lambdafun` penalty function.

```
lambdafun = @(t) exp(L - (L-S)*exp(-t/mu));
t = 0:.1:25;
y = lambdafun(t);
figure
semilogy(t,y);
title('Penalty Function for VRP Approach')
ylabel('Penalty')
xlabel('Time')
```

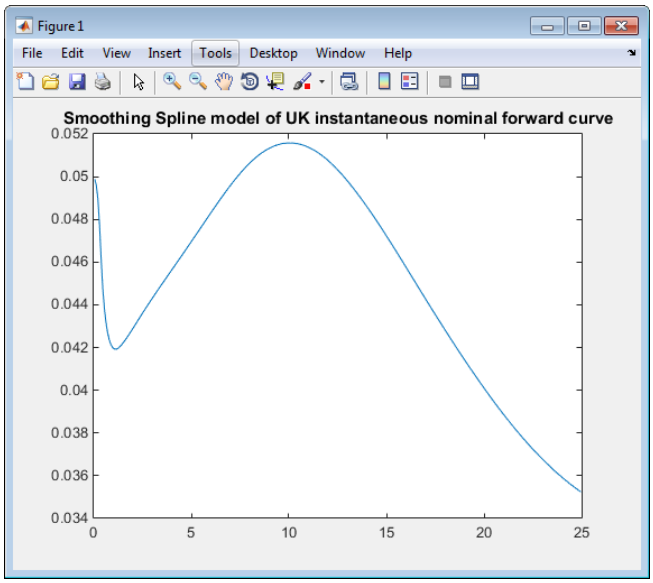


Use the `fitSmoothingSpline` function to fit the interest-rate curve and model the `Lambdafun` penalty function.

```
VRPModel = IRFunctionCurve.fitSmoothingSpline('Forward',CurveSettle,...
Instruments,lambdafun,'Compounding',-1, 'InstrumentPeriod',InstrumentPeriod);
```

Plot the smoothing spline interest-rate curve for forward rates.

```
PlottingDates = CurveSettle+20:30:CurveSettle+365*25;
TimeToMaturity = yearfrac(CurveSettle,PlottingDates);
VRPForwardRates = getForwardRates(VRPModel, PlottingDates);
plot(TimeToMaturity,VRPForwardRates)
title('Smoothing Spline model of UK instantaneous nominal forward curve')
```



Using fitFunction to Create Custom Fitting Function

When using an `IRFunctionCurve` object, you can create a custom fitting function with the `fitFunction` function. To use `fitFunction`, you must define a `FunctionHandle`. In addition, you must also use `IRFitOptions` to define an `IRFitOptions` object to support an `InitialGuess` for the parameters of the curve function.

Example

The following example demonstrates the use of `fitFunction` with a `FunctionHandle` and an `IRFitOptions` object.

```
Settle = repmat(datenum('30-Apr-2008'),[6 1]);
Maturity = [datenum('07-Mar-2009');datenum('07-Mar-2011');...
datenum('07-Mar-2013');datenum('07-Sep-2016');...
datenum('07-Mar-2025');datenum('07-Mar-2036')];

CleanPrice = [100.1;100.1;100.8;96.6;103.3;96.3];
CouponRate = [0.0400;0.0425;0.0450;0.0400;0.0500;0.0425];
Instruments = [Settle Maturity CleanPrice CouponRate];
CurveSettle = datenum('30-Apr-2008');
```

Define the `FunctionHandle`.

```
functionHandle = @(t,theta) polyval(theta,t);
```

Define the `OptOptions` for `IRFitOptions`.

```
OptOptions = optimoptions('lsqnonlin','display','iter');
```

Define `fitFunction`.

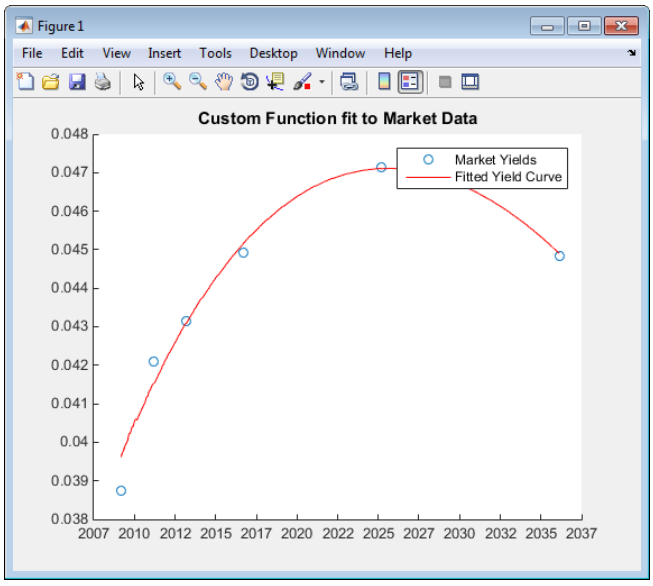
```
CustomModel = IRFunctionCurve.fitFunction('Zero', CurveSettle, ...
functionHandle,Instruments, IRFitOptions([.05 .05 .05],'FitType','price',...
'OptOptions',OptOptions));
```

```
                                        Norm of      First-order
 Iteration  Func-count     f(x)          step          optimality   CG-iterations
     0          4         38036.7                     4.92e+04
     1          8         38036.7            10       4.92e+04            0
     2         12         38036.7           2.5       4.92e+04            0
```

```
     3          16         38036.7           0.625        4.92e+04            0
     4          20         38036.7         0.15625        4.92e+04            0
     5          24         30741.5       0.0390625        1.72e+05            0
     6          28         30741.5        0.078125        1.72e+05            0
     7          32         30741.5       0.0195312        1.72e+05            0
     8          36         28713.6      0.00488281        2.33e+05            0
     9          40         20323.3      0.00976562        9.47e+05            0
    10          44         20323.3       0.0195312        9.47e+05            0
    11          48         20323.3      0.00488281        9.47e+05            0
    12          52         20323.3       0.0012207        9.47e+05            0
    13          56         19698.8    0.000305176        1.08e+06            0
    14          60           17493    0.000610352           7e+06            0
    15          64           17493      0.0012207           7e+06            0
    16          68           17493    0.000305176           7e+06            0
    17          72         15455.1    7.62939e-05        2.25e+07            0
    18          76         15455.1    0.000177499        2.25e+07            0
    19          80         13317.1     3.8147e-05        3.18e+07            0
    20          84         12865.3    7.62939e-05        7.83e+07            0
    21          88         11779.8    7.62939e-05        7.58e+06            0
    22          92         11747.6    0.000152588        1.45e+05            0
    23          96         11720.9    0.000305176        2.33e+05            0
    24         100         11667.2    0.000610352        1.48e+05            0
    25         104         11558.6      0.0012207        3.55e+05            0
    26         108         11335.5      0.0024414        1.57e+05            0
    27         112         10863.8      0.0048828        6.36e+05            0
    28         116         9797.14      0.0097656        2.53e+05            0
    29         120         6882.83       0.0195312        9.18e+05            0
    30         124         6882.83       0.0373993        9.18e+05            0
    31         128         3218.45      0.0093498        1.96e+06            0
    32         132         612.703       0.0186996        3.01e+06            0
    33         136         13.0998       0.0253882        3.05e+06            0
    34         140       0.0762922      0.00154002        5.05e+04            0
    35         144       0.0731652    3.61102e-06            29.9            0
    36         148       0.0731652    6.32335e-08           0.063            0

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.
```

Plot the custom function that is defined using `fitFunction`.

```
Yields = bndyield(CleanPrice,CouponRate,Settle(1),Maturity);
scatter(Maturity,Yields);
PlottingPoints = min(Maturity):30:max(Maturity);
hold on;
plot(PlottingPoints, getParYields(CustomModel, PlottingPoints),'r');
datetick
legend('Market Yields','Fitted Yield Curve')
title('Custom Function fit to Market Data')
```



See Also

`IRBootstrapOptions` | `IRDataCurve` | `IRFunctionCurve` | `IRFitOptions`

Related Examples

- “Creating Interest-Rate Curve Objects” on page 9-4
- “Bootstrapping a Swap Curve” on page 2-141
- “Dual Curve Bootstrapping” on page 9-12
- “Creating an IRDataCurve Object” on page 9-6
- “Converting an IRDataCurve or IRFunctionCurve Object” on page 9-30
- “Analyze Inflation-Indexed Instruments” on page 2-132
- “Fitting Interest-Rate Curve Functions” on page 9-24

More About

- “Interest-Rate Curve Objects and Workflow” on page 9-2
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

External Websites

- Calibration and Simulation Best Practices: Multifactor Interest Rate Models for Risk Applications (30 min 00 sec)

Fitting Interest-Rate Curve Functions

This example shows how to use `IRFunctionCurve` objects to model the term structure of interest rates (also referred to as the yield curve). This can be contrasted with modeling the term structure with vectors of dates and data and interpolating between the points (which can currently be done with the function `prbyzero`). The term structure can refer to at least three different curves: the discount curve, zero curve, or forward curve.

The `IRFunctionCurve` object allows you to model an interest-rate curve as a function.

This example explores using an `IRFunctionCurve` object to model the default-free term structure of interest rates in the United Kingdom. Three different forms for the term structure are implemented and are discussed in more detail later:

- Nelson-Siegel
- Svensson
- Smoothing Cubic Spline with a so-called Variable Roughness Penalty (VRP)

Choosing the Data

The first question in modeling the yield curve is what data should be used. To model a default-free yield curve, default-free, option-free market instruments must be used. The most significant component of the data is UK Government Bonds (known as Gilts). Historical data is retrieved from the following site:

https://www.dmo.gov.uk

Repo data is used to construct the short end of the yield curve. Repo data is retrieved from the following site:

https://www.ukfinance.org.uk/

Note also that the data must be specified as a matrix where the columns are `Settle`, `Maturity`, `CleanPrice`, and `CouponRate` and that instruments must be bonds or synthetically converted to bonds.

Market data for a close date of April 30, 2008, has been downloaded and saved to the following data file (`ukdata20080430`), which is loaded into MATLAB® with the following command:

```
% Load the data
load ukdata20080430

% Convert repo rates to be equivalent zero coupon bonds
RepoCouponRate = repmat(0,size(RepoRates));
RepoPrice = bndprice(RepoRates, RepoCouponRate, RepoSettle, RepoMaturity);

% Aggregate the data
Settle = [RepoSettle;BondSettle];
Maturity = [RepoMaturity;BondMaturity];
CleanPrice = [RepoPrice;BondCleanPrice];
CouponRate = [RepoCouponRate;BondCouponRate];
Instruments = [Settle Maturity CleanPrice CouponRate];
InstrumentPeriod = [repmat(0,6,1);repmat(2,31,1)];
```

```
CurveSettle = datenum('30-Apr-2008');
```

Fit Nelson-Siegel Model to Market Data

The Nelson-Siegel model proposes that the instantaneous forward curve can be modeled with the following:

$$f = \beta_0 + \beta_1 e^{\frac{-m}{\tau}} + \beta_2 e^{\frac{-m}{\tau}} \frac{m}{\tau}$$

This can be integrated to derive an equation for the zero curve (see [6] for more information on the equations and the derivation):

$$s = \beta_0 + (\beta_1 + \beta_2)\frac{\tau}{m}(1 - e^{\frac{-m}{\tau}}) - \beta_2 e^{\frac{-m}{\tau}}$$

See [1 on page 9-28] for more information.

The `IRFunctionCurve` object provides the capability to fit a Nelson Siegel curve to observed market data with the `fitNelsonSiegel` method. The fitting is done by calling the Optimization Toolbox™ function `lsqnonlin`.

The `fitNelsonSiegel` function has required inputs for Curve `Type`, Curve `Settle`, and a matrix of instrument data.

Optional input arguments, specified in name-value pair argument, are:

- `IRFitOptions` structure: Provides the capability to choose which quantity to be minimized (price, yield, or duration weighted price) and other optimization parameters (for example, upper and lower bounds for parameters).
- Curve `Compounding` and `Basis` (day-count convention)
- Additional instrument parameters, `Period`, `Basis`, `FirstCouponDate`, and so on.

```
NSModel = IRFunctionCurve.fitNelsonSiegel('Zero',CurveSettle,...
    Instruments,'InstrumentPeriod',InstrumentPeriod);
```

Fit Svensson Model

A very similar model to the Nelson-Siegel model is the Svensson model, which adds two additional parameters to account for greater flexibility in the term structure. This model proposes that the forward rate can be modeled with the following form:

$$f = \beta_0 + \beta_1 e^{\frac{-m}{\tau_1}} + \beta_2 e^{\frac{-m}{\tau_1}} \frac{m}{\tau_1} + \beta_3 e^{\frac{-m}{\tau_2}} \frac{m}{\tau_2}$$

As above, this can be integrated to derive an equation for the zero curve:

$$s = \beta_0 + \beta_1 (1 - e^{\frac{-m}{\tau_1}})(-\frac{\tau_1}{m}) + \beta_2 ((1 - e^{\frac{-m}{\tau_1}})\frac{\tau_1}{m} - e^{\frac{m}{\tau_1}}) + \beta_3 ((1 - e^{\frac{-m}{\tau_2}})\frac{\tau_2}{m} - e^{\frac{m}{\tau_2}})$$

See [2 on page 9-28] for more information.

Fitting the parameters to this model proceeds in a similar fashion to the Nelson-Siegel model using the `fitSvensson` function.

```
SvenssonModel = IRFunctionCurve.fitSvensson('Zero',CurveSettle,...
    Instruments,'InstrumentPeriod',InstrumentPeriod);
```

Fit Smoothing Spline

The term structure can also be modeled with a spline, specifically, one way to model the term structure is by representing the forward curve with a cubic spline. To ensure that the spline is sufficiently smooth, a penalty is imposed relating to the curvature (second derivative) of the spline:

$$\sum_{i=1}^{N}[\frac{P_i - \widehat{P}_i(f)}{D_i}]^2 + \int_0^M \lambda_t(m)[f''(m)]^2 dm$$

where the first term is the difference between the observed price *P* and the predicted price, *P_hat*, (weighted by the bond's duration, *D*) summed over all bonds in the data set, and the second term is the penalty term (where *lambda* is a penalty function and *f* is the spline).

See [3 on page 9-28], [4 on page 9-28], [5 on page 9-29] below.

There have been different proposals for the specification of the penalty function *lambda*. One approach, advocated by [4 on page 9-28], and currently used by the UK Debt Management Office, is a penalty function of the following form:

$$log(\lambda(m)) = L - (L - S)e^{\frac{-m}{\mu}}$$

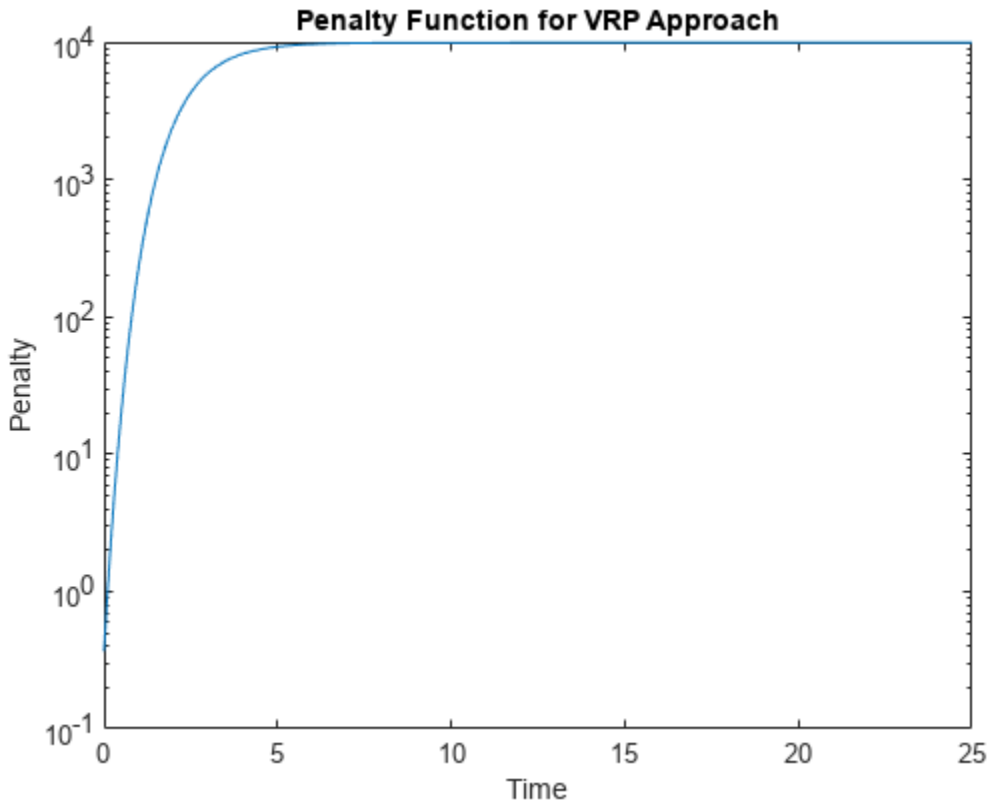
The parameters *L*, *S*, and *mu* are typically estimated from historical data.

The `IRFunctionCurve` object can be used to fit a smoothing spline representation of the forward curve with a penalty function using the function `fitSmoothingSpline`.

Required inputs, like for the functions above, are a `CurveType`, Curve`Settle`, `Instruments` matrix, and a function handle (`Lambdafun`) containing the penalty function.

The optional parameters are similar to `fitNelsonSiegel` and `fitSvensson`.

```
% Parameters chosen to be roughly similar to [4] below.
L = 9.2;
S = -1;
mu = 1;

lambdafun = @(t) exp(L - (L-S)*exp(-t/mu)); % Construct penalty function
t = 0:.1:25; % Construct data to plot penalty function
y = lambdafun(t);
figure
semilogy(t,y);
title('Penalty Function for VRP Approach')
ylabel('Penalty')
xlabel('Time')
```

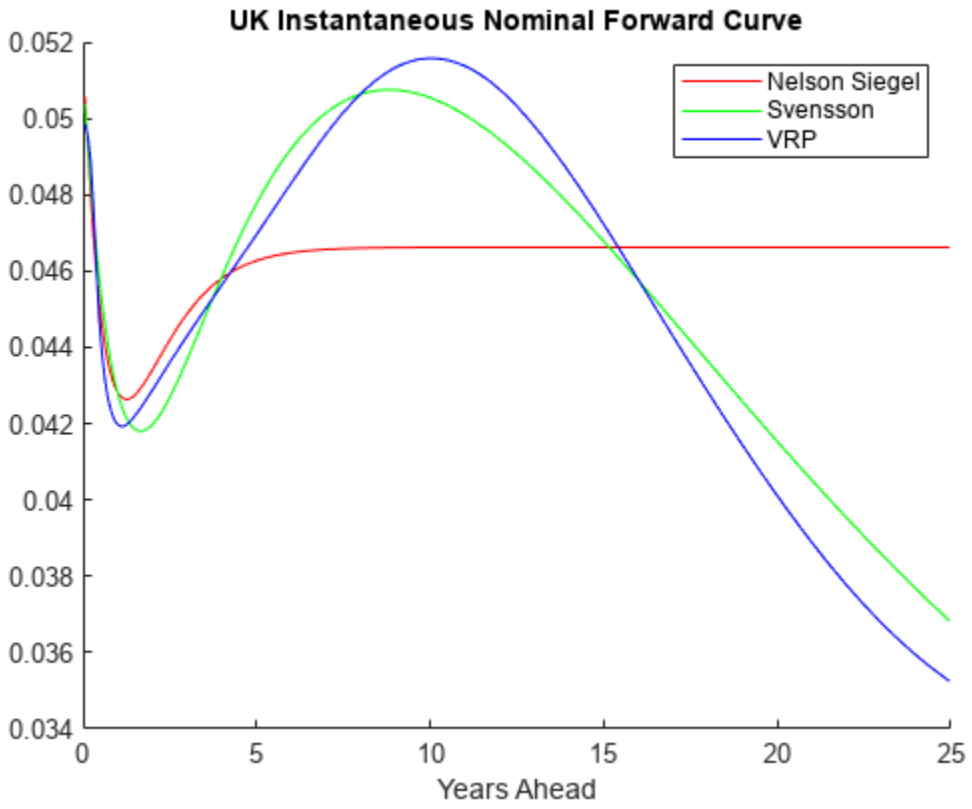



```
VRPModel = IRFunctionCurve.fitSmoothingSpline('Forward',CurveSettle,...
    Instruments,lambdafun,'Compounding',-1,...
    'InstrumentPeriod',InstrumentPeriod);
```

Use Fitted Curves and Plot Results

Once a curve is created, functions are used to extract the Forward and Zero Rates and the Discount Factors. This curve can also be converted into a `RateSpec` structure using the `toRateSpec` function. The `RateSpec` can then be used with many other functions in the Financial Instruments Toolbox™

```
PlottingDates = CurveSettle+20:30:CurveSettle+365*25;
TimeToMaturity = yearfrac(CurveSettle,PlottingDates);

NSForwardRates = NSModel.getForwardRates(PlottingDates);
SvenssonForwardRates = SvenssonModel.getForwardRates(PlottingDates);
VRPForwardRates = VRPModel.getForwardRates(PlottingDates);

figure
hold on
plot(TimeToMaturity,NSForwardRates,'r')
plot(TimeToMaturity,SvenssonForwardRates,'g')
plot(TimeToMaturity,VRPForwardRates,'b')
title('UK Instantaneous Nominal Forward Curve')
xlabel('Years Ahead')
legend({'Nelson Siegel','Svensson','VRP'})
```



Compare with this Link

This link provides a live look at the derived yield curve published by the UK

https://www.bankofengland.co.uk

Bibliography

This example is based on the following papers and journal articles:

[1] Nelson, C.R., Siegel, A.F. "Parsimonious Modelling of Yield Curves." *Journal of Business.* 60, pp 473-89, 1987.

[2] Svensson, L.E.O. "Estimating and Interpreting Forward Interest Rates: Sweden 1992-4." International Monetary Fund, IMF Working Paper, 1994/114, 1994.

[3] Fisher, M., Nychka, D., Zervos, D. "Fitting the Term Structure of Interest Rates with Smoothing Splines." Board of Governors of the Federal Reserve System, Federal Reserve Board Working Paper, 95-1, 1995.

[4] Anderson, N., Sleath, J. "New Estimates of the UK Real and Nominal Yield Curves." *Bank of England Quarterly Bulletin.* November, pp 384-92, 1999.

[5] Waggoner, D. "Spline Methods for Extracting Interest Rate Curves from Coupon Bond Prices." Federal Reserve Board Working Paper, 97-10, 1997.

[6] "Zero-Coupon Yield Curves: Technical Documentation." BIS Papers No. 25, October 2005.

[7] Bolder, D.J., Gusba,S. "Exponentials, Polynomials, and Fourier Series: More Yield Curve Modelling at the Bank of Canada." Working Papers 02-29, Bank of Canada, 2002.

[8] Bolder, D.J., Streliski, D. "Yield Curve Modelling at the Bank of Canada." Technical Reports 84, Bank of Canada, 1999.

See Also

`IRBootstrapOptions` | `IRDataCurve` | `IRFunctionCurve` | `IRFitOptions`

Related Examples

- “Creating Interest-Rate Curve Objects” on page 9-4
- “Bootstrapping a Swap Curve” on page 2-141
- “Dual Curve Bootstrapping” on page 9-12
- “Creating an IRDataCurve Object” on page 9-6
- “Converting an IRDataCurve or IRFunctionCurve Object” on page 9-30
- “Analyze Inflation-Indexed Instruments” on page 2-132

More About

- “Interest-Rate Curve Objects and Workflow” on page 9-2
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

External Websites

- Calibration and Simulation Best Practices: Multifactor Interest Rate Models for Risk Applications (30 min 00 sec)

Converting an IRDataCurve or IRFunctionCurve Object

In this section...

Introduction

The `IRDataCurve` and `IRFunctionCurve` objects for interest-rate curves support conversion to:

- A `RateSpec` structure.

  The `RateSpec` generated from an `IRDataCurve` or `IRFunctionCurve` object, using the `toRateSpec` function, is identical to the `RateSpec` structure created with `intenvset` using Financial Instruments Toolbox software.
- A vector of dates and data from an `IRDataCurve` object

  The vector of dates and data is acceptable to `prbyzero`, `bkcall`, `bkput`, `tfutbyprice`, and `tfutbyyield` or any function that requires a term structure of interest rates.

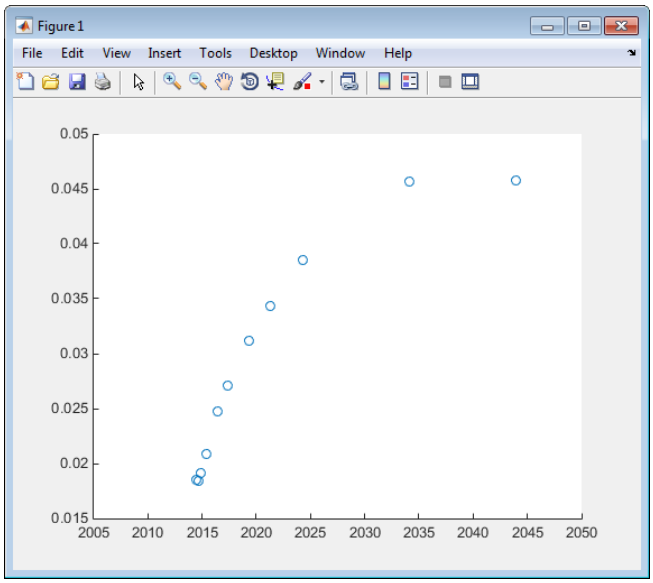
Using the toRateSpec Function

To convert an `IRDataCurve` or `IRFunctionCurve` object to a `RateSpec` structure, you must first create an interest-rate curve object. Then, use the `toRateSpec` function for an `IRDataCurve` object or the`toRateSpec` function for an `IRFunctionCurve` object.

Example

Create a data vector from the following data: `https://www.ustreas.gov/offices/domestic-finance/debt-management/interest-rate/yield.shtml`.

```
Data = [1.85 1.84 1.91 2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = daysadd(today,[30 90 180 360 2*360 3*360 5*360 7*360 10*360 20*360 30*360],2);
scatter(Dates,Data)
datetick
```



Create an `IRDataCurve` interest-rate curve object.

```
rr = IRDataCurve('Zero',today,Dates,Data);
```

Convert to a `RateSpec`.

```
toRateSpec(rr, today+30:30:today+365)
```

```
ans =
          FinObj: 'RateSpec'
      Compounding: 2
             Disc: [12x1 double]
            Rates: [12x1 double]
         EndTimes: [12x1 double]
       StartTimes: [12x1 double]
         EndDates: [12x1 double]
       StartDates: 733569
    ValuationDate: 733569
            Basis: 0
     EndMonthRule: 1
```

Using Vector of Dates and Data

You can use the `getZeroRates` function for an `IRDataCurve` object with a `Dates` property to create a vector of dates and data acceptable for `prbyzero` in Financial Toolbox software and `bkcall`, `bkput`, `tfutbyprice`, and `tfutbyyield` in Financial Instruments Toolbox software.

Example

This is an example of using an `IRDataCurve` object with the `getZeroRates` function with `prbyzero`.

```
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = daysadd(today,[360 2*360 3*360 5*360 7*360 10*360 20*360 30*360],1);
irdc = IRDataCurve('Zero',today,Dates,Data,'InterpMethod','pchip');
Maturity = daysadd(today,8*360,1);
CouponRate = .055;
```

```
ZeroDates = daysadd(today,180:180:8*360,1);
ZeroRates = getZeroRates(irdc, ZeroDates);
BondPrice = prbyzero([Maturity CouponRate], today, ZeroRates, ZeroDates)
```

```
BondPrice =
  113.9250
```

See Also

IRBootstrapOptions | IRDataCurve | IRFunctionCurve | IRFitOptions

Related Examples

- “Creating an IRFunctionCurve Object” on page 9-16
- “Dual Curve Bootstrapping” on page 9-12
- “Analyze Inflation-Indexed Instruments” on page 2-132
- “Fitting Interest-Rate Curve Functions” on page 9-24

More About

- “Interest-Rate Curve Objects and Workflow” on page 9-2
- “Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

Numerix Workflows

Working with Simple Numerix Trades

This example shows how to price a callable reverse floater using Numerix CROSSASSET.

Initialize Numerix environment.

```
import com.numerix.integration.*;
import com.numerix.integration.implementation.*;

n = numerix('i:\NumeriX_java_10_3_0\data')
```

```
n =

              Path: 'i:\NumeriX_java_10_3_0\data'
    RepositoryPath: 'i:\NumeriX_java_10_3_0\data\Repository'
        Repository: [1x1 com.numerix.integration.implementation.FileSystemRepository]
           Context: [1x1 com.numerix.integration.implementation.LocalCalculationContext]
       LookupsPath: 'i:\NumeriX_java_10_3_0\data\Data\LookupRules'
       MarketsPath: 'i:\NumeriX_java_10_3_0\data\Data\Markets'
       FixingsPath: 'i:\NumeriX_java_10_3_0\data\Data\Fixings'
        TradesPath: 'i:\NumeriX_java_10_3_0\data\Data\Trades'
        Parameters: [1x1 com.numerix.integration.implementation.CalculationParameters]
```

Create a market.

```
quotes = java.util.HashMap;
quotes.put('IR.USD-LIBOR-3M.SWAP-1Y.MID', 0.0066056);
quotes.put('IR.USD-LIBOR-3M.SWAP-10Y.MID', 0.022465005);
quotes.put('IR.USD-LIBOR-3M.SWAP-20Y.MID', 0.027544995);
market = Market('EOD_14-NOV-2011', DateExtensions.date('14-Nov-2011'), quotes.entrySet);
```

Define a trade instance for a callable reverse floater based on instrument template located in the `Repository`.

```
tradeDescriptor = 'TRADE.IR.CALLABLEREVERSEFLOATER';
tradeParameters = java.util.HashMap;
tradeParameters.put('Trade ID','1001');
tradeParameters.put('Quote Type', 'MID');
tradeParameters.put('Currency', 'USD');
tradeParameters.put('Notional', 1000000.0);
tradeParameters.put('Effective Date', DateExtensions.date('1-Dec-2011'));
tradeParameters.put('Termination Date', DateExtensions.date('1-Dec-2021'));
tradeParameters.put('IR Index', 'LIBOR');
tradeParameters.put('IR Index Tenor', '3M');
tradeParameters.put('Structured Freq', '3M');
tradeParameters.put('Structured Side', 'Receive');
tradeParameters.put('Structured Coupon Floor', 0.0);
tradeParameters.put('Structured Coupon UpBd', 0.08);
tradeParameters.put('StructuredCoupon Multiplier', 1.4);
tradeParameters.put('Structured Coupon Cap', 0.05);
tradeParameters.put('Structured Basis', 'ACT/360');
tradeParameters.put('Funding Freq', '3M');
tradeParameters.put('Funding Side', 'Pay');
tradeParameters.put('Funding Spread', 0.003);
tradeParameters.put('Funding Basis', 'ACT/360');
tradeParameters.put('Call Start Date', DateExtensions.date('1-Dec-2013'));
tradeParameters.put('Call End Date', DateExtensions.date('1-Dec-2020'));
tradeParameters.put('Option Side', 'Short');
tradeParameters.put('Option Type', 'Right to Terminate');
tradeParameters.put('Call Frequency', '3M');
tradeParameters.put('Model', 'IR.USD-LIBOR-3M.MID.DET');
tradeParameters.put('Method', 'BackwardAnalytic');
```

Create the trade instance.

```
trade = RepositoryExtensions.createTradeInstance(n.Repository, tradeDescriptor, tradeParameters)
```

Price the trade.

```
results = CalculationContextExtensions.calculate(n.Context, trade, market, Request.getAll);
```

Parse the results for MATLAB and display.

```
r = n.parseResults(results)
disp([r.Name r.Category r.Currency r.Data])
```

```
r =

    Category: {13x1 cell}
    Currency: {13x1 cell}
        Name: {13x1 cell}
        Data: {13x1 cell}

    'Reporting Currency'           'Price'       ''       'USD'
    'Structured Cashflow Log'      'Cashflow'    ''        {41x20 cell}
    'Structured Leg PV Accrued'    'Price'       'USD'    [          0]
    'PV'                           'Price'       'USD'    [ 6.4133e+04]
    'Structured Leg PV Clean'      'Price'       'USD'    [ 4.2637e+05]
    'Option PV'                    'Price'       'USD'    [-1.3220e+05]
    'Funding Cashflow Log'         'Cashflow'    ''        {41x20 cell}
    'Structured Leg PV'            'Price'       'USD'    [ 4.2637e+05]
    'Funding Leg PV'               'Price'       'USD'    [-2.3004e+05]
    'Funding Leg PV Accrued'       'Price'       'USD'    [          0]
    'Funding Leg PV Clean'         'Price'       'USD'    [-2.3004e+05]
    'Yield Risk Report'            ''            ''        { 4x30 cell}
    'Messages'                     ''            ''        { 4x1  cell}
```

See Also

`numerix` | `parseResults` | `numerixCrossAsset`

Related Examples

- “Working with Advanced Numerix Trades” on page 10-4
- “Use Numerix to Price Cash Deposits” on page 10-8
- “Use Numerix for Interest-Rate Risk Assessment” on page 10-10

External Websites

- `https://www.numerix.com/product/crossasset`

Working with Advanced Numerix Trades

This example shows how to price multiple trades from MATLAB using Numerix CROSSASSET.

Initialize Numerix environment.

```
import com.numerix.integration.*;
import com.numerix.integration.implementation.*;

n = numerix('i:\NumeriX_java_10_3_0\data')
```

```
n =

             Path: 'i:\NumeriX_java_10_3_0\data'
    RepositoryPath: 'i:\NumeriX_java_10_3_0\data\Repository'
        Repository: [1x1 com.numerix.integration.implementation.FileSystemRepository]
           Context: [1x1 com.numerix.integration.implementation.LocalCalculationContext]
       LookupsPath: 'i:\NumeriX_java_10_3_0\data\Data\LookupRules'
       MarketsPath: 'i:\NumeriX_java_10_3_0\data\Data\Markets'
       FixingsPath: 'i:\NumeriX_java_10_3_0\data\Data\Fixings'
        TradesPath: 'i:\NumeriX_java_10_3_0\data\Data\Trades'
        Parameters: [1x1 com.numerix.integration.implementation.CalculationParameters]
```

Specify the hybrid model for multiple trades.

```
hySpec = HybridModelSpecification;
hySpec.addHW1F('IR-USD', 'USD', 'LIBOR', '3M', 'MeanReversion(0.5),DiagonalSwaption(ATM, 10Y)');
hySpec.addHW1F('IR-EUR', 'EUR', 'EURIBOR', '6M', 'MeanReversion(0.5),DiagonalSwaption(ATM, 10Y)');
hySpec.addFXBlack('FX-USDEUR', 'USD', 'EUR', 'LIBOR', '3M', 'EURIBOR', '6M', 'StrikeFXEuropean(ATM, 10Y)');
% 5 Specify the factor correlations.
hyCorrelations = HybridModelCorrelationMatrix(hySpec);
hyCorrelations.add('IR-USD', 'IR-EUR', 0.5);
hyCorrelations.add('IR-USD', 'FX-USDEUR', 0.25);
hyCorrelations.add('IR-EUR', 'FX-USDEUR', 0.25);

% Specify the model parameters.
hybridModelParameters = java.util.HashMap;
hybridModelParameters.put('Quote Type', 'MID');
hybridModelParameters.put('Payout Currency', 'USD');
hybridModelParameters.put('Specification', hySpec);
hybridModelParameters.put('Correlations', hyCorrelations);
```

Specify exposure calculation parameters.

```
observationDates = CustomObservationSchedule;
observationDates.add(DateExtensions.date(2011, 12, 1));
for y = 2012:2013
  for m = 1:12
    observationDates.add(DateExtensions.date(y, m, 1));
  end
end

exposureParameters = java.util.HashMap;
exposureParameters.put('Model ID', 'HYBRID');
exposureParameters.put('Observation Dates', observationDates);
```

Define the first trade instance.

```
tradeParameters1 = java.util.HashMap;
tradeParameters1.put('Trade ID', 'RVFL1001');
tradeParameters1.put('Quote Type', 'MID');
tradeParameters1.put('Currency', 'USD');
tradeParameters1.put('Notional', 1000000.0);
tradeParameters1.put('Effective Date', DateExtensions.date('1-Dec-2011'));
tradeParameters1.put('Termination Date', DateExtensions.date('1-Dec-2021'));
tradeParameters1.put('IR Index', 'LIBOR');
tradeParameters1.put('IR Index Tenor', '3M');
tradeParameters1.put('Structured Freq', '3M');
tradeParameters1.put('Structured Side', 'Receive');
tradeParameters1.put('Structured Coupon Floor', 0.0);
tradeParameters1.put('Structured Coupon UpBd', 0.08);
tradeParameters1.put('StructuredCoupon Multiplier', 1.4);
tradeParameters1.put('Structured Coupon Cap', 0.05);
tradeParameters1.put('Structured Basis', 'ACT/360');
```

```
tradeParameters1.put('Funding Freq', '3M');
tradeParameters1.put('Funding Side', 'Pay');
tradeParameters1.put('Funding Spread', 0.003);
tradeParameters1.put('Funding Basis', 'ACT/360');
tradeParameters1.put('Call Start Date', DateExtensions.date('1-Dec-2013'));
tradeParameters1.put('Call End Date', DateExtensions.date('1-Dec-2020'));
tradeParameters1.put('Option Side', 'Short');
tradeParameters1.put('Option Type', 'Right to Terminate');
tradeParameters1.put('Call Frequency', '3M');
tradeParameters1.put('Model', 'HYBRID');
tradeParameters1.put('Method', 'BackwardMC');
tradeInstance1 = RepositoryExtensions.createTradeInstance(n.Repository, 'TRADE.IR.CALLABLEREVERSEFLOATER', tradeParameters1);
```

Define the second trade instance.

```
tradeParameters2 = java.util.HashMap;
tradeParameters2.put('Trade ID', 'CASHDEP1001');
tradeParameters2.put('Quote Type', 'MID');
tradeParameters2.put('Currency', 'USD');
tradeParameters2.put('Coupon Rate', 0.05);
tradeParameters2.put('Yield', 0.044);
tradeParameters2.put('Notional', 100.0);
tradeParameters2.put('Effective Date', DateExtensions.date('1-Apr-2012'));
tradeParameters2.put('Maturity', DateExtensions.date('1-Apr-2013'));
tradeParameters2.put('IR Index', 'LIBOR');
tradeParameters2.put('IR Index Tenor', '3M');
tradeParameters2.put('Model', 'HYBRID');
tradeParameters2.put('Method', 'BACKWARDMC');
tradeInstance2 = RepositoryExtensions.createTradeInstance(n.Repository, 'IR.CASHDEPOSIT', tradeParameters2);
```

Create the third trade instance.

```
tradeParameters3 = java.util.HashMap;
tradeParameters3.put('Trade ID', 'FXFWD1001');
tradeParameters3.put('Quote Type', 'MID');
tradeParameters3.put('Base Currency', 'USD');
tradeParameters3.put('Term Currency', 'EUR');
tradeParameters3.put('Delivery Date', DateExtensions.date('1-Jun-2012'));
tradeParameters3.put('Contract FX Forward Rate', 80.5);
tradeParameters3.put('Base Notional', 10000000.0);
tradeParameters3.put('Base IR Index', 'LIBOR');
tradeParameters3.put('Term IR Index', 'EURIBOR');
tradeParameters3.put('Base IR Index Tenor', '3m');
tradeParameters3.put('Term IR Index Tenor', '6m');
tradeParameters3.put('Calendar', 'NewYork Target');
tradeParameters3.put('Spot Lag', '2bd');
tradeParameters3.put('Model', 'HYBRID');
tradeParameters3.put('Method', 'BACKWARDMC');
tradeInstance3 = RepositoryExtensions.createTradeInstance(n.Repository, 'FX.FXFORWARD', tradeParameters3);
```

Set `tradeInstances` for all three trade instances.

```
tradeInstances = java.util.ArrayList;
tradeInstances.add(tradeInstance1);
tradeInstances.add(tradeInstance2);
tradeInstances.add(tradeInstance3);
n.Parameters.setInstances(tradeInstances);
```

Add a custom lookup so these trade instances reference the hybrid model.

```
n.Parameters.getLookups.add(0,ExactLookupRule('HYBRID','MODEL.HYBRID',hybridModelParameters.entrySet));
```

Add another custom lookup so that exposure report has parameters defined.

```
n.Parameters.getLookups.add(1,ExactLookupRule('RISK.REPORT.EXPOSURE','REPORT.EXPOSURE',exposureParameters.entrySet));
```

Perform the calculation.

```
results = n.Context.calculate(n.Parameters, Request.getExposure);
```

Parse the results for MATLAB and display.

```
r = n.parseResults(results)

disp([r.Trade(2) r.Market(2)])
disp([r.Results{2}.Name r.Results{2}.Category r.Results{2}.Currency r.Results{2}.Data])
```

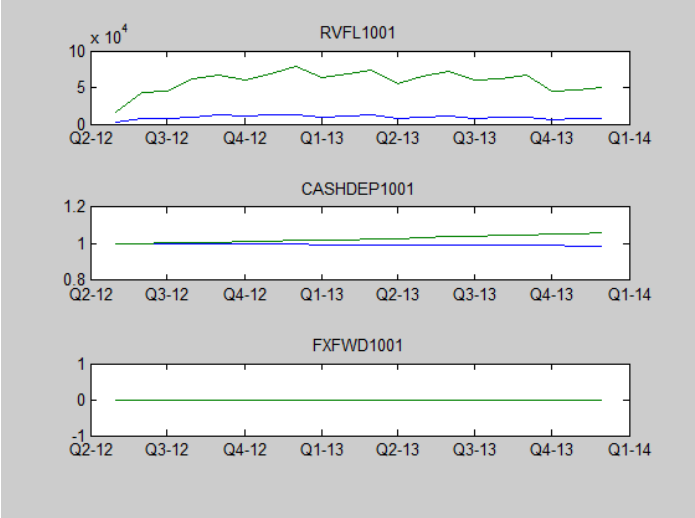
```
disp([r.Results{2}.Name{1}])
disp([r.Results{2}.Data{1}])
```

```
r =

      Trade: {3x1 cell}
     Market: {3x1 cell}
    Results: {3x1 cell}

 'CASHDEP1001'    'EOD'

    'Exposure'                     ''    ''    {21x501 cell}
    'Exposure.Discount Factors'    ''    ''    {21x501 cell}
    'Messages'                     ''    ''    {12x1   cell}

Exposure
  Columns 1 through 3

    'DATE'                          'VALUE 1'             'VALUE 2'
    'Tue May 01 13:00:00 EDT 2012'    [104.198166609924]    [103.386222783828]
    'Fri Jun 01 13:00:00 EDT 2012'    [ 104.09953599675]    [102.117465067435]
    'Sun Jul 01 13:00:00 EDT 2012'    [105.524567506006]    [100.055731577867]
    'Wed Aug 01 13:00:00 EDT 2012'    [105.787455961524]    [100.318762976796]
    'Sat Sep 01 13:00:00 EDT 2012'    [104.417483614373]    [100.764337265155]
    'Mon Oct 01 13:00:00 EDT 2012'    [104.692275556824]    [100.980213613911]
    'Thu Nov 01 13:00:00 EDT 2012'    [104.443818312902]    [101.478508725115]
    'Sat Dec 01 12:00:00 EST 2012'    [104.736646932343]    [101.679769557039]
    'Tue Jan 01 12:00:00 EST 2013'    [104.577562970494]    [102.423339265735]
    'Fri Feb 01 12:00:00 EST 2013'    [ 104.28994278039]    [103.117326879887]
    'Fri Mar 01 12:00:00 EST 2013'    [ 104.70469459715]    [104.232180198939]
    'Mon Apr 01 13:00:00 EDT 2013'    [ 105.07334321718]    [ 105.05089338769]
    'Wed May 01 13:00:00 EDT 2013'    [                0]    [                0]
    'Sat Jun 01 13:00:00 EDT 2013'    [                0]    [                0]
    'Mon Jul 01 13:00:00 EDT 2013'    [                0]    [                0]
    'Thu Aug 01 13:00:00 EDT 2013'    [                0]    [                0]
    'Sun Sep 01 13:00:00 EDT 2013'    [                0]    [                0]
    'Tue Oct 01 13:00:00 EDT 2013'    [                0]    [                0]
    'Fri Nov 01 13:00:00 EDT 2013'    [                0]    [                0]
    'Sun Dec 01 12:00:00 EST 2013'    [                0]    [                0]

.

.

.

  Columns 499 through 501

    'VALUE 498'           'VALUE 499'           'VALUE 500'
    [ 105.36273206453]    [104.335982034187]    [104.141595030057]
    [105.904822463264]    [104.238089023172]    [104.276676080686]
    [103.893060436208]    [103.613968079212]    [106.188617261199]
    [103.183889382889]    [105.499763150412]    [105.440275818983]
    [103.310404527817]    [105.233622768447]    [105.267337892552]
    [103.274239052394]    [104.716952177783]    [ 104.33099834332]
    [103.583983117053]    [104.710250522521]    [105.501004542869]
    [103.379982561438]    [105.146939039653]    [104.681616459661]
    [103.821169954095]    [105.567274949306]    [104.835971977691]
    [104.016530403399]    [105.254054161819]    [104.842156238753]
    [104.481475787501]    [105.197179985119]    [104.962752610848]
    [105.061984636083]    [105.077227736476]    [105.077766765965]
    [                0]    [                0]    [                0]
    [                0]    [                0]    [                0]
    [                0]    [                0]    [                0]
    [                0]    [                0]    [                0]
    [                0]    [                0]    [                0]
    [                0]    [                0]    [                0]
    [                0]    [                0]    [                0]
    [                0]    [                0]    [                0]
```

Plot the results for the second trade instance, `CASHDEP1001`, with the corresponding Exposure Discount Factors.

```
figure('Tag','NumerixAdvancedRiskExample');

for ii=1:3
   % Get dates
   dates = cell2mat(r.Results{ii}.Data{expIndex}(2:end,1));
   dates = dates(:,4:end);
   dates = floor(datenum(dates));
```

```
    % Get exposures
    mtm = cell2mat(r.Results{ii}.Data{expIndex}(2:end,2:end))';
    exposures = max(0,mtm);  % Exposure at contract level, no netting
    EE = mean(exposures);  % Expected Exposure
    PFE = prctile(exposures,95);  % Potential Future Exposure
    subplot(3,1,ii)
    plot(dates,EE,dates,PFE)
    title(r.Trade{ii})
    datetick
end
```



See Also

`numerix` | `parseResults` | `numerixCrossAsset`

Related Examples

- “Working with Simple Numerix Trades” on page 10-2
- “Use Numerix to Price Cash Deposits” on page 10-8
- “Use Numerix for Interest-Rate Risk Assessment” on page 10-10

External Websites

- https://www.numerix.com/product/crossasset

Use Numerix to Price Cash Deposits

This example shows how to use the Numerix CROSSASSET API to price a cash deposit from MATLAB. The trade parameters are read from the `Cashdeposit1.csv` in the `Numerix Data Trades` folder.

Initialize Numerix environment.

```
import com.numerix.integration.*;
import com.numerix.integration.implementation.*;

n = numerix('i:\NumeriX_java_10_3_0\data')
```

```
n =

              Path: 'i:\NumeriX_java_10_3_0\data'
    RepositoryPath: 'i:\NumeriX_java_10_3_0\data\Repository'
        Repository: [1x1 com.numerix.integration.implementation.FileSystemRepository]
           Context: [1x1 com.numerix.integration.implementation.LocalCalculationContext]
       LookupsPath: 'i:\NumeriX_java_10_3_0\data\Data\LookupRules'
       MarketsPath: 'i:\NumeriX_java_10_3_0\data\Data\Markets'
       FixingsPath: 'i:\NumeriX_java_10_3_0\data\Data\Fixings'
        TradesPath: 'i:\NumeriX_java_10_3_0\data\Data\Trades'
        Parameters: [1x1 com.numerix.integration.implementation.CalculationParameters]
```

Create a market.

```
market = Market('EOD_16-APR-2012', DateExtensions.date('16-APR-2012'), []);
```

Read the `Cashdeposit1.csv` file from the `Numerix Trades` folder.

```
[~,~,tradeInfo] = xlsread([n.TradesPath '\Cashdeposit1.csv'])
```

```
tradeInfo =

    'Template'          'String'      'TRADE.IR.CASHDEPOSIT'
    'Trade ID'          'ID'          'CASHDEP1001'
    'Quote Type'        'String'      'MID'
    'Effective Date'    'Date'        '4/1/2012'
    'Maturity'          'Date'        '4/1/2013'
    'Notional'          'Double'      [                100]
    'Currency'          'Currency'    'USD'
    'Coupon Rate'       'Double'      [             0.0500]
    'Yield'             'Double'      [             0.0440]
    'IR Index'          'String'      'Libor'
    'IR Index Tenor'    'Tenor'       '3m'
```

Define a trade instance from the imported `CASHDEP1001` instrument.

```
tradeDescriptor = tradeInfo{1,3};
tradeParameters = java.util.HashMap;
numTradeInfoFields = size(tradeInfo,1);
for i = 2:numTradeInfoFields
  switch tradeInfo{i,2}
    case {'DATE','Date'}
      tradeParameters.put(tradeInfo{i,1},DateExtensions.date(datestr(tradeInfo{i,3},'dd-mmm-yyyy')));
    otherwise
      tradeParameters.put(tradeInfo{i,1},tradeInfo{i,3});
  end
end
```

Create the trade instance.

```
trade = RepositoryExtensions.createTradeInstance(n.Repository, tradeDescriptor, tradeParameters);
```

Price the trade.

```
results = CalculationContextExtensions.calculate(n.Context, trade, market, Request.getAll);
```

Parse the results for MATLAB and display.

```
r = n.parseResults(results)
disp([r.Name r.Category r.Currency r.Data])
```

```
 r =

    Category: {9x1 cell}
    Currency: {9x1 cell}
        Name: {9x1 cell}
        Data: {9x1 cell}

    'Modified Duration'     'Price'    ''       [   0.9349]
    'Accrued Interest'      'Price'    'USD'    [   0.2083]
    'Reporting Currency'    'Price'    ''       'USD'
    'PV'                    'Price'    'USD'    [ 100.7607]
    'Instrument'            'Price'    ''       [1x85 char]
    'Clean Price'           'Price'    'USD'    [ 100.5524]
    'Convexity'             'Price'    ''       [   1.7481]
    'YTM'                   'Price'    ''                []
    'Messages'              ''         ''                []
```

See Also

`numerix` | `parseResults` | `numerixCrossAsset`

Related Examples

- “Working with Simple Numerix Trades” on page 10-2
- “Working with Advanced Numerix Trades” on page 10-4
- “Use Numerix for Interest-Rate Risk Assessment” on page 10-10

External Websites

- https://www.numerix.com/product/crossasset

Use Numerix for Interest-Rate Risk Assessment

This example shows how to use the Numerix CROSSASSET API for interest-rate curve stripping for risk assessment.

Initialize Numerix environment.

```
import com.numerix.integration.*;
import com.numerix.integration.implementation.*;

n = numerix('i:\NumeriX_java_10_3_0\data')
```

```
n =

             Path: 'i:\NumeriX_java_10_3_0\data'
    RepositoryPath: 'i:\NumeriX_java_10_3_0\data\Repository'
        Repository: [1x1 com.numerix.integration.implementation.FileSystemRepository]
           Context: [1x1 com.numerix.integration.implementation.LocalCalculationContext]
       LookupsPath: 'i:\NumeriX_java_10_3_0\data\Data\LookupRules'
       MarketsPath: 'i:\NumeriX_java_10_3_0\data\Data\Markets'
       FixingsPath: 'i:\NumeriX_java_10_3_0\data\Data\Fixings'
        TradesPath: 'i:\NumeriX_java_10_3_0\data\Data\Trades'
        Parameters: [1x1 com.numerix.integration.implementation.CalculationParameters]
```

Specify the current market associated with the Numerix CROSSASSET environment.

```
markets = get(n.Parameters,'Markets');
currentMarket = markets.get(0);
outInstance = RefObject(currentMarket);
```

Define the interest-rate curve key `IR.USD-LIBOR-3M.MID`.

```
n.Context.tryResolveId('IR.USD-LIBOR-3M.MID',outInstance);
currentInstance = outInstance.argvalue;
```

Set the instance and market.

```
n.Parameters.setMarkets(java.util.Arrays.asList(currentMarket));
n.Parameters.setInstances(java.util.Arrays.asList(currentInstance));
```

Calculate the interest-rate curve stripping.

```
results = n.Context.calculate(n.Parameters,Request.getAll);
```

The calculation returns the results from stripping the interest-rate curve for `IR.USD-LIBOR-3M.MID`. Parse the results for MATLAB and display.

```
% IR.USD-LIBOR-3M.MID.
r = n.parseResults(results)

disp([r.Instance r.Market])
disp([r.Results{1}.Name r.Results{1}.Category r.Results{1}.Currency r.Results{1}.Data])
disp([r.Results{1}.Name{1}])
disp([r.Results{1}.Data{1}])
```

```
r =

    Instance: {'IR.USD-LIBOR-3M.MID'}
      Market: {'EOD'}
     Results: {[1x1 struct]}

 'IR.USD-LIBOR-3M.MID'    'EOD'

'Curve Info'    ''    ''    {30x3 cell}
'Messages'      ''    ''    { 7x1 cell}

Curve Info

  'KEY'                                 'DATE'                           'DISCOUNTFACTOR'
    'CASH RATE 16-APR-2012 17-APR-2012'    'Tue Apr 17 13:00:00 EDT 2012'    [        1.0000]
    'CASH RATE 16-APR-2012 18-APR-2012'    'Wed Apr 18 13:00:00 EDT 2012'    [        1.0000]
    'CASH RATE 16-APR-2012 23-APR-2012'    'Mon Apr 23 13:00:00 EDT 2012'    [        1.0000]
```

```
'CASH RATE 16-APR-2012 30-APR-2012'    'Mon Apr 30 13:00:00 EDT 2012'    [        0.9999]
'CASH RATE 16-APR-2012 16-MAY-2012'    'Wed May 16 13:00:00 EDT 2012'    [        0.9998]
'CASH RATE 16-APR-2012 18-JUN-2012'    'Mon Jun 18 13:00:00 EDT 2012'    [        0.9994]
'CASH RATE 16-APR-2012 16-JUL-2012'    'Mon Jul 16 13:00:00 EDT 2012'    [        0.9988]
'CASH RATE 18-MAY-2012 12-AUG-2012'    'Sun Aug 12 13:00:00 EDT 2012'    [        0.9987]
'CASH RATE 20-JUN-2012 20-SEP-2012'    'Thu Sep 20 13:00:00 EDT 2012'    [        0.9981]
'CASH RATE 18-JUL-2012 18-OCT-2012'    'Thu Oct 18 13:00:00 EDT 2012'    [        0.9975]
'CASH RATE 15-AUG-2012 15-NOV-2012'    'Thu Nov 15 12:00:00 EST 2012'    [        0.9973]
'CASH RATE 19-SEP-2012 19-DEC-2012'    'Wed Dec 19 12:00:00 EST 2012'    [        0.9968]
'CASH RATE 17-OCT-2012 17-JAN-2013'    'Thu Jan 17 12:00:00 EST 2013'    [        0.9962]
'CASH RATE 19-DEC-2012 19-MAR-2013'    'Tue Mar 19 13:00:00 EDT 2013'    [        0.9955]
'SWAP RATE 18-APR-2012 19-APR-2016'    'Tue Apr 19 13:00:00 EDT 2016'    [        0.9645]
'SWAP RATE 18-APR-2012 18-APR-2017'    'Tue Apr 18 13:00:00 EDT 2017'    [        0.9445]
'SWAP RATE 18-APR-2012 18-APR-2018'    'Wed Apr 18 13:00:00 EDT 2018'    [        0.9199]
'SWAP RATE 18-APR-2012 18-APR-2019'    'Thu Apr 18 13:00:00 EDT 2019'    [        0.8925]
'SWAP RATE 18-APR-2012 21-APR-2020'    'Tue Apr 21 13:00:00 EDT 2020'    [        0.8639]
'SWAP RATE 18-APR-2012 19-APR-2021'    'Mon Apr 19 13:00:00 EDT 2021'    [        0.8356]
'SWAP RATE 18-APR-2012 19-APR-2022'    'Tue Apr 19 13:00:00 EDT 2022'    [        0.8069]
'SWAP RATE 18-APR-2012 18-APR-2023'    'Tue Apr 18 13:00:00 EDT 2023'    [        0.7784]
'SWAP RATE 18-APR-2012 18-APR-2024'    'Thu Apr 18 13:00:00 EDT 2024'    [        0.7506]
'SWAP RATE 18-APR-2012 19-APR-2027'    'Mon Apr 19 13:00:00 EDT 2027'    [        0.6733]
'SWAP RATE 18-APR-2012 20-APR-2032'    'Tue Apr 20 13:00:00 EDT 2032'    [        0.5682]
'SWAP RATE 18-APR-2012 20-APR-2037'    'Mon Apr 20 13:00:00 EDT 2037'    [        0.4828]
'SWAP RATE 18-APR-2012 21-APR-2042'    'Mon Apr 21 13:00:00 EDT 2042'    [        0.4112]
'SWAP RATE 18-APR-2012 18-APR-2052'    'Thu Apr 18 13:00:00 EDT 2052'    [        0.3087]
'SWAP RATE 18-APR-2012 18-APR-2062'    'Tue Apr 18 13:00:00 EDT 2062'    [        0.2414]
```

See Also

`numerix` | `parseResults` | `numerixCrossAsset`

Related Examples

- “Working with Simple Numerix Trades” on page 10-2
- “Working with Advanced Numerix Trades” on page 10-4
- “Use Numerix to Price Cash Deposits” on page 10-8

External Websites

- https://www.numerix.com/product/crossasset

Numerix CROSSASSET Interface Workflow Example Using Matrix, Data, and Call Objects

This example shows how to use the Numerix CROSSASSET API to create and price a vanilla European option.

Construct a `numerixCrossAsset` object for Java® or C++.

For the Java SDK API:

```
c = numerixCrossAsset
```

```
 c =

numerixCrossAsset with properties:

Application: [1x1 com.numerix.pro.Application]
ApplicationWarning: [1x1 com.numerix.pro.ApplicationWarning]
```

For the C++ SDK API on Windows®:

```
c = numerixCrossAsset(true)
```

```
 c =

numerixCrossAsset with properties:

Application: [1×1 fininst.internal.NumerixCAIL]
ApplicationWarning: []
```

Create and register data as a Matrix with the Numerix Cross Asset Integration Layer Application using the `applicationMatrix` method.

```
rowData = [41992, 42020, 42449, 42905, 43115];
colData = [390, 395, 400, 405];
volData = [0.35778, 0.35132, 0.34394, 0.33582;...
           0.33405, 0.32819, 0.32669, 0.31904;...
           0.31576, 0.31235, 0.30371, 0.30261;...
           0.29391, 0.29366, 0.28962, 0.28932;...
           0.28787, NaN,     0.28347, NaN    ];
applicationMatrix(c,'BYSTRIKEVOLDATA',rowData,colData,volData);
```

Create and register the yield curve data with the Application object. Use a table for optimal display purposes. Dates must be relative to '01/01/1900' and the Numerix Cross Asset Integration Layer API supports date number representation only. MATLAB datetime's get converted automatically, otherwise date numbers must be input and based relative to '01/01/1900'.

```
dates = datetime({'18-Feb-2014';'20-May-2014';'18-Jun-2014';'16-Jul-2014';
                  '20-Aug-2014';'17-Sep-2014';'15-Oct-2014';'19-Nov-2014';
                  '17-Dec-2014';'18-Mar-2015';'17-Jun-2015';'16-Sep-2015';
                  '16-Dec-2015';'16-Mar-2016';'15-Jun-2016';'21-Sep-2016';
                  '21-Dec-2016';'15-Mar-2017';'20-Feb-2018';'20-Feb-2019';
                  '20-Feb-2020';'22-Feb-2021';'22-Feb-2022';'21-Feb-2023';
                  '20-Feb-2024';'20-Feb-2025';'20-Feb-2026';'20-Feb-2029';
                  '21-Feb-2034';'22-Feb-2039';'22-Feb-2044';'20-Feb-2054';
                  '20-Feb-2064'},'locale','en_US');
```

Define the corresponding discount factors.

```
discountFactors = [1;0.99942;0.999231;0.999037;0.998797;0.998616;0.998385;...
                   0.998122;0.997941;0.997159;0.996157;0.994825;0.993065;...
                   0.99078;0.987889;0.984092;0.979913;0.975459;0.952707;...
                   0.922223;0.888128;0.852291;0.816462;0.781228;0.746677;...
                   0.712892;0.680462;0.592285;0.474003;0.383493;0.312617;...
                   0.213809;0.152345];
```

Supported Numerix Cross Asset Integration Layer API names are `DATE` and `DISCOUNTFACTOR` for the creation of the data.

```
curveData = table(dates,discountFactors,'VariableNames',{'DATE','DISCOUNTFACTOR'});
applicationData(c,'USD_3MLIBOR_CURVE',curveData);
```

Define the headers for registering the `RATESPEC` and `DIVSPEC` call objects.

```
headers = {'ID','LOCAL ID','TIMER','TIMER CPU','UPDATED'};
```

Data is required to create dividend curve. Create and register the `DIVSPEC` call object using name-value pairs in this example.

```
applicationCall(c,headers,'ID','DIVSPEC','OBJECT','MARKET DATA','TYPE','DIVIDEND',...
                  'COMMENT','Comments here','SKIP',false,'NOWDATE',41688,...
                  'CURRENCY','USD','RATE/DIVIDEND',0,'BASIS','ACT/360');
```

Create the `EQUITYVOLSPEC` call object. `BYSTRIKEVOLDATA` denotes the volatility matrix object created previously, using an array of names and an array of values in this example.

```
applicationCall(c,headers,{'ID','OBJECT','TYPE','COMMENT','SKIP','NOWDATE','CURRENCY','VOLATILITYBASIS',...
                           'DATA','INTERPMETHOD','INTERPVARIABLE','EXTRAPOLATION'},...
                         {'EQVOLSPEC','MARKET DATA','EQ VOL','Comments here',...
                          false,41688,'USD','ACT/360','BYSTRIKEVOLDATA',...
                          'LINEAR','VOL','FLAT EXTRAPOLATION'});
```

Create the `RATESPEC` call object. `USD_3MLIBOR_CURVE` denotes yield curve data object created previously using name-value pairs.

```
applicationCall(c,headers,'ID','RATESPEC','OBJECT','MARKET DATA','TYPE','YIELD','COMMENT','Comments here',...
                  'SKIP',false,'INTERPMETHOD','LogLinear','INTERPVARIABLE','DF',...
                  'CURRENCY','USD','DATA','USD_3MLIBOR_CURVE','BASIS','ACT/360');
```

Create the `EuropeanOptionEQ` instrument. Create the `STOCKSPEC` call object using the `applicationCall` method.

```
applicationCall(c,headers,'ID','STOCKSPEC','OBJECT','INSTRUMENT','TYPE','EQ EUROPEAN',...
                  'COMMENT','Comments here','SKIP',false,'FLAVOR','PUT',...
                  'CURRENCY','USD','ENDDATE',43976,'SETTLEMENTDATE',43976,...
                  'STRIKE',112,'SIGMA1',0.2,'NOTIONAL',100);
```

Price the portfolio by creating and registering call object to run pricing analytics. Create the `OPTIONSPEC_CLOSEFORM` call object headers for registering the `OPTIONSPEC_CLOSEFORM` call object.

```
headers = {'ATM','DELTA','DELTA TRADER','FORWARD DELTA','FORWARD DELTA TRADER', ...
           'FUTURES DELTA','FUTURES DELTA TRADER','GAMMA','GAMMA TRADER', ...
           'ID','LOCAL ID','NOTIONAL','PRICE','PV','RHO','RHO TRADER', ...
           'SIGMA1','STRIKE','THETA','TIMER','TIMER CPU','UPDATED','VANNA', ...
           'VANNA TRADER','VEGA','VEGA TRADER','VOLGA','VOLGA TRADER'};

applicationCall(c,headers,'ID','OPTIONSPEC_CLOSEFORM','OBJECT','ANALYTIC',...
                  'TYPE','EUROPEAN OPTION','COMMENT','Comments here',...
                  'SKIP',false,'NOWDATE',41688,'OPTION','STOCKSPEC',...
                  'DIVIDENDCURVE','DIVSPEC','DOMESTICYIELDCURVE','RATESPEC',...
                  'SPOTPRICE',112,'SPOTDATE',41688,'MODEL','BLACK');
```

Create an output structure in MATLAB from the Application object using the `getdata` method.

```
appData = getdata(c);
```

Display the results.

```
[appData.OPTIONSPEC_CLOSEFORM.OUTPUT_HEADERS
appData.OPTIONSPEC_CLOSEFORM.OUTPUT_VALUES]
```

```
ans =

  28×2 cell array

    'PRICE'                  [            1467.24]
    'PV'                     [            1467.24]
    'DELTA'                  [             -30.54]
    'FORWARD DELTA'          [             -30.54]
    'FUTURES DELTA'          [             -26.83]
    'GAMMA'                  [               0.62]
```

```
'VEGA'                  [             9827.91]
'VOLGA'                 [              205.45]
'VANNA'                 [               -1.44]
'DELTA TRADER'          [              -34.20]
'FORWARD DELTA TRADER'  [              -34.20]
'FUTURES DELTA TRADER'  [              -30.05]
'GAMMA TRADER'          [                0.78]
'VEGA TRADER'           [               98.28]
'VOLGA TRADER'          [                0.02]
'VANNA TRADER'          [               -0.02]
'SIGMA1'                [                0.20]
'STRIKE'                [              112.00]
'NOTIONAL'              [              100.00]
'RHO'                   [           -30638.08]
'THETA'                 [               -0.15]
'RHO TRADER'            [               -3.06]
'ATM'                   [              127.48]
'UPDATED'               '12 @ 01:37:24 PM'
'ID'                    'OPTIONSPEC_CLOSEFORM'
'TIMER'                 [                0.17]
'TIMER CPU'             [                0.06]
'LOCAL ID'              'OPTIONSPEC_CLOSEFORM'
```

Close the `numerixCrossAsset` object.

```
close(c)
```

See Also

`numerixCrossAsset` | `applicationData` | `applicationMatrix` | `getdata` | `applicationCall` | `close`

External Websites

- https://www.numerix.com/product/crossasset

Functions

Calibrate Pricing Model

Calibrate option pricing model in the Live Editor

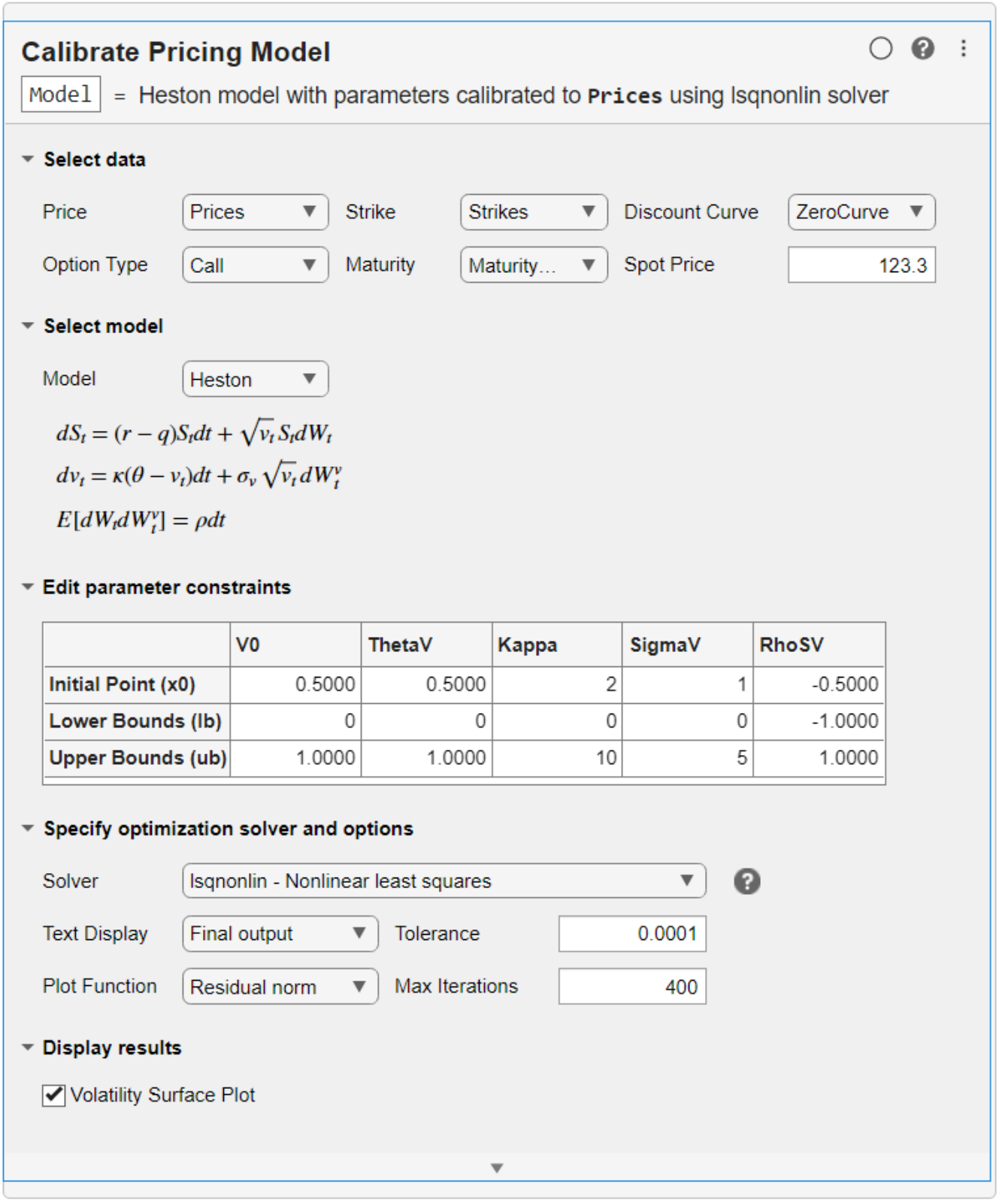
Description

The **Calibrate Pricing Model** task lets you interactively calibrate an equity, FX, or commodity option pricing model using market data. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Select data.
- Select a model.
- Edit parameter constraints.
- Specify an optimization solver and options.
- Display the results in a volatility surface plot.

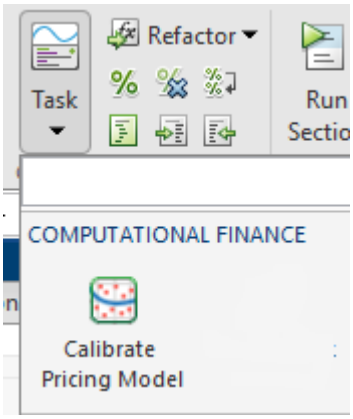
For general information about Live Editor tasks, see “Add Interactive Tasks to a Live Script”.


Calibrate Pricing Model
Model = Heston model with parameters calibrated to Prices using lsqnonlin solver
Select data
Price
Prices
Strike
Strikes
Discount Curve
ZeroCurve
Option Type
Call
Maturity
Maturity...
Spot Price
123.3
Select model
Model
Heston
$dS_t = (r-q)S_t dt + \sqrt{v_t} S_t dW_t$
$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$
$E[dW_t dW_t^v] = \rho dt$
Edit parameter constraints
V0 ThetaV Kappa SigmaV RhoSV
Initial Point (x0) 0.5000 0.5000 2 1 -0.5000
Lower Bounds (lb) 0 0 0 0 -1.0000
Upper Bounds (ub) 1.0000 1.0000 10 5 1.0000
Specify optimization solver and options
Solver
lsqnonlin - Nonlinear least squares
Text Display
Final output
Tolerance
0.0001
Plot Function
Residual norm
Max Iterations
400
Display results
Volatility Surface Plot

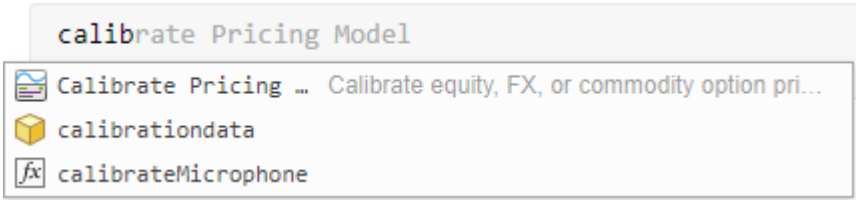
Open the Task

To add the **Calibrate Pricing Model** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Calibrate Pricing Model**.



- In a code block in the script, type a relevant keyword, such as `calibrate`. Select **Calibrate Pricing Model** from the suggested command completions.



Parameters

`Select data` — Data for model
matrix for **Price**, **Strike**, **Maturity** | object for **Discount Curve** | string for **Option Type** | positive numeric for **Spot Price**

Select data enables you to select the data that you want to use to calibrate your pricing model:

- **Price** — `M`-by-`N` numeric matrix for prices
- **Strike** — `M`-by-`N` numeric matrix or `M`-by-`1` column vector for strikes and `Strike` must be a nonnegative value.
- **Discount Curve** — Rate curve (`ratecurve`) for the zero curve
- **Option Type** — Put or call option type
- **Maturity** — Option maturity dates
- **Spot Price** — Current underlying asset price

`Select model` — Specify model type
object for **Heston**, **Bates**, or **Merton**

Select model enables you to select the model type that you want to use to calibrate pricing:

- **Heston** — The Heston model is an extension of the Black-Scholes model, where the volatility (square root of variance) is no longer assumed to be constant, and the variance now follows a stochastic (CIR) process. This option allows modeling the implied volatility smiles observed in the market.

  The stochastic differential equation is

  $$dS_t = (r - q)S_t dt + \sqrt{v_t} S_t dW_t$$
  $$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$
  $$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$

  where

  - r is the continuous risk-free rate.
  - q is the continuous dividend yield.
  - S_t is the asset price at time t.
  - v_t is the asset price variance at time t
  - v_0 is the initial variance of the asset price at $t = 0$ for ($v_0 > 0$).
  - θ is the long-term variance level for ($\theta > 0$).
  - κ is the mean reversion speed for the variance for ($\kappa > 0$).
  - σ_v is the volatility of the variance for ($\sigma_v > 0$).
  - p is the correlation between the Weiner processes W_t and W^v_t for ($-1 \le p \le 1$).
- **Bates** — The Bates model extends the Heston model by including stochastic volatility and (similar to Merton) jump diffusion parameters in the modeling of sudden asset price movements.

  The stochastic differential equation is

  $$dS_t = (r - q - \lambda_p \mu_J)S_t dt + \sqrt{v_t} S_t dW_t + J S_t dP_t$$
  $$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t$$
  $$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$
  $$\mathrm{prob}(dP_t = 1) = \lambda_p dt$$

  where

  - r is the continuous risk-free rate.
  - q is the continuous dividend yield.
  - S_t is the asset price at time t.
  - v_t is the asset price variance at time t.
  - J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

    $$\frac{1}{(1 + J)\delta\sqrt{2\pi}} \exp\left\{ \frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2} \right\}$$

where

- v_0 is the initial variance of the asset price at $t = 0$ ($v_0 > 0$).
- θ is the long-term variance level for ($\theta > 0$).
- κ is the mean reversion speed for ($\kappa > 0$).
- σ_v is the volatility of variance for ($\sigma_v > 0$).
- p is the correlation between the Weiner processes W_t and W_t^v for ($-1 \le p \le 1$).
- μ_J is the mean of J for ($\mu_J > -1$).
- λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \ge 0$).
- δ is the standard deviation of $\ln(1+J)$ for ($\delta \ge 0$).

- **Merton** — The Merton jump diffusion model extends the Black-Scholes model by using the Poisson process to include jump diffusion parameters in the modeling of sudden asset price movements (both up and down).

  The stochastic differential equation is

  $$dS_t = (r - q - \lambda_p \mu_j)S_t dt + \sigma S_t dW_t + JS_t dP_t$$
  $$\text{prob}(dP_t = 1) = \lambda_p dt$$

  where

  - r is the continuous risk-free rate.
  - q is the continuous dividend yield.
  - W_t is the Weiner process.
  - J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

    $$\frac{1}{(1 + J)\delta\sqrt{2\pi}} \exp\left\{ \frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]^2\right.}{2\delta^2} \right\}$$

    where

    - μ_J is the mean of J for ($\mu_J > -1$).
    - δ is the standard deviation of $\ln(1+J)$ for ($\delta \ge 0$).
    - λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \ge 0$).
    - σ is the volatility of the asset price for ($\sigma > 0$).

`Edit parameter constraints` — Parameter constraints for model and solver table

The parameter constraints displayed in table for editing depend on the model type that you specify from **Select model**.

- **Heston**
  - **V0** — Initial variance of the underlying asset
  - **ThetaV** — Long-term variance of underlying asset
  - **Kappa** — Mean revision speed for the variance of underlying asset
  - **SigmaV** — Volatility of the variance of underlying asset
  - **RhoSV** — Correlation between Weiner processes for underlying asset and its variance
- **Bates**
  - **V0** — Initial variance of the underlying asset
  - **ThetaV** — Long-term variance of underlying asset
  - **Kappa** — Mean revision speed for the variance of underlying asset
  - **SigmaV** — Volatility of the variance of underlying asset
  - **RhoSV** — Correlation between Weiner processes for underlying asset and its variance
  - **MeanJ** — Mean of the random percentage jump size
  - **JumpVol** — Standard deviation of `log`(1+*J*)
  - **JumpFreq** — Annual frequency of Poisson jump process
- **Merton**
  - **Volatility** — Volatility value for the underlying asset
  - **MeanJ** — Mean of the random percentage jump size
  - **JumpVol** — Standard deviation of `log`(1+*J*)
  - **JumpFreq** — Annual frequency of Poisson jump process

In addition, you can edit the table for **Lower Bounds (lb)** and **Upper Bounds (ub)**.

`Specify optimization solver and options` — Specify solver type and options
drop-down control for **Solver**, **Text Display**, and **Plot Function** | text box for **Tolerance** | text box for **Max Iterations**

Specify optimization solver and options enables you to specify the solver and options for optimization and visualizing results:

- **Solver**
  - **lsqnonlin - Non linear least squares** — For information, see `lsqnonlin`.
  - **simulannelbnd - Simulated annealing** — For information, see `simulannealbnd`.
- **Text Display**
  - **Final output**
  - **Each iteration**
  - **No display**

  For information, see “Iterative Display”.
- **Plot Function**

- **Best value**
- **Current value**
- **No plot**

For information, see “Plot Functions”.

- **Tolerance** — Termination tolerance of the objective function `lsqnonlin` or `simulannealbnd`. `Tolerance` must be a positive value.
- **Max Iterations** — Maximum number of iterations allowed. **Max Iterations** must be a positive numeric value.

`Display results` — Display optimization results
check box to toggle display of **Volatility Surface Plot**

Select the **Volatility Surface Plot** check box to display the current optimization results.

Version History

Introduced in R2022a

Support for Maximum Iterations
Behavior changed in R2022b

The **Specify optimization solver and options** section of the Live Editor task supports an option for **Max Iterations**.

See Also

Functions
`Heston` | `Bates` | `Merton` | `ratecurve` | `lsqnonlin` | `simulannealbnd`

Topics
“Calibrate Option Pricing Model Using Heston Model” on page 3-143

asianbycrr

Price Asian option from Cox-Ross-Rubinstein binomial tree

Syntax

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

Description

`Price = asianbycrr(CRRTree,OptSpec,Strike,Settle,ExerciseDates)` prices Asian options using a Cox-Ross-Rubinstein binomial tree.

Note Alternatively, you can use the `Asian` object to price Asian options. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`Price = asianbycrr( ___ ,AmericanOpt,AvgType,AvgPrice,AvgDate)` adds optional arguments for `AmericanOpt`, `AvgType`, `AvgPrice`, and `AvgDate`.

Examples

Price a Floating-Strike Asian Option Using a CRR Binomial Tree

This example shows how to price a floating-strike Asian option using a CRR binomial tree using 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;

OptSpec = 'put';
Strike = NaN;
Settle = datetime(2003,1,1);
ExerciseDates = datetime(2004,1,1);

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

```
Price = 1.2177
```

Input Arguments

`CRRTree` — Stock tree structure

structure

Stock tree structure, specified by using `crrtree`.

Data Types: `struct`

OptSpec — Definition of option
character vector with value 'call' or 'put' | cell array of character vectors with values 'call' or 'put'

Definition of option, specified as 'call' or 'put' using a character vector or a cell array of character vectors.

Data Types: char | cell

Strike — Option strike price value
matrix of nonnegative integers

Option strike price value, specified with a nonnegative integer using a NINST-by-1 matrix of strike price values.

To compute the value of a floating-strike Asian option, Strike must be specified as NaN. Floating-strike Asian options are also known as average strike options.

Data Types: double

Settle — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date for the Asian option, specified as a NINST-by-1 matrix of settlement or trade dates using a datetime array, string array, or date character vectors.

To support existing code, asianbycrr also accepts serial date numbers as inputs, but they are not recommended. For more information, see Version History on page 11-12.

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

Data Types: char | string | datetime

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use aNINST-by-1 matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one ExerciseDates on the option expiry date.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. 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 a NINST-by-1 vector, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

  To support existing code, asianbycrr also accepts serial date numbers as inputs, but they are not recommended. For more information, see Version History on page 11-12.

Data Types: char | string | datetime

AmericanOpt — Option type
0 European (default) | integer with values 0 or 1

(Optional) Option type, specified as `NINST`-by-`1` positive integer flags with values:

- `0` — European
- `1` — American

Data Types: `double`

`AvgType` — Average types
`arithmetic` (default) | character vector with values of `arithmetic` or `geometric`

Average types, specified as `arithmetic` for arithmetic average, or `geometric` for geometric average.

Data Types: `char`

`AvgPrice` — Average price of underlying asset at `Settle`
scalar numeric

Average price of underlying asset at `Settle`, specified as a scalar numeric.

Note Use this argument when `AvgDate` < `Settle`.

Data Types: `double`

`AvgDate` — Date averaging period begins
datetime scalar | string scalar | date character vector

Date averaging period begins, specified as a scalar datetime, string, or date character vector.

To support existing code, `asianbycrr` also accepts serial date numbers as inputs, but they are not recommended. For more information, see Version History on page 11-12.

Data Types: `char` | `string` | `datetime`

Output Arguments

`Price` — Expected prices for Asian options at time `0`
vector

Expected prices for Asian options at time 0, returned as a `NINST`-by-`1` vector. Pricing of Asian options is done using Hull-White (1993). Therefore, for these options there are no unique prices on the tree nodes except for the root node.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `asianbycrr` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`crrtree` | `instasian` | `Asian`

Topics

“Pricing Asian Options” on page 3-110
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asianbyeqp

Price Asian option from Equal Probabilities binomial tree

Syntax

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

Description

`Price = asianbyeqp(EQPTree,OptSpec,Strike,Settle,ExerciseDates)` prices Asian options using an Equal Probabilities binomial tree.

Note Alternatively, you can use the `Asian` object to price Asian options. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`Price = asianbyeqp( ___ ,AmericanOpt,AvgType,AvgPrice,AvgDate)` adds optional arguments for `AmericanOpt`, `AvgType`, `AvgPrice`, and `AvgDate`.

Examples

Price a Floating-Strike Asian Option Using an EQP Equity Tree

This example shows how to price a floating-strike Asian option using an EQP equity tree by loading 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;

OptSpec = 'put';
Strike = NaN;
Settle = datetime(2003,1,1);
ExerciseDates = datetime(2004,1,1);

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

```
Price = 1.2724
```

Input Arguments

EQPTree — Stock tree structure

structure

Stock tree structure, specified by using `eqptree`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a character vector or a cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price value
matrix of nonnegative integers

Option strike price value, specified with a nonnegative integer using a `NINST`-by-`1` matrix of strike price values.

To compute the value of a floating-strike Asian option, `Strike` must be specified as `NaN`. Floating-strike Asian options are also known as average strike options.

Data Types: `double`

`Settle` — Settlement date or trade date
datetime array | scalar array | date character vector

Settlement date or trade date for the Asian option, specified as a `NINST`-by-`1` matrix of settlement or trade dates using a datetime array, string array, or date character vectors.

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

To support existing code, `asianbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a `NINST`-by-`1` matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. 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 a `NINST`-by-`1` vector, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `asianbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
`0` European (default) | integer with values `0` or `1`

(Optional) Option type, specified as `NINST`-by-`1` positive integer flags with values:

- `0` — European
- `1` — American

Data Types: `double`

`AvgType` — Average types
`arithmetic` (default) | character vector with values of `arithmetic` or `geometric`

Average types, specified as `arithmetic` for arithmetic average, or `geometric` for geometric average.

Data Types: `char`

`AvgPrice` — Average price of underlying asset at `Settle`
scalar numeric

Average price of underlying asset at `Settle`, specified as a scalar numeric.

Note Use this argument when `AvgDate` < `Settle`.

Data Types: `double`

`AvgDate` — Date averaging period begins
datetime scalar | string scalar | date character vector

Date averaging period begins, specified as a scalar datetime, string, or date character vector.

To support existing code, `asianbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`Price` — Expected prices for Asian options at time `0`
vector

Expected prices for Asian options at time 0, returned as a `NINST`-by-`1` vector. Pricing of Asian options is done using Hull-White (1993). Therefore, for these options there are no unique prices on the tree nodes except for the root node.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `asianbyeqp` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`eqptree` | `instasian` | `Asian`

Topics

“Pricing Asian Options” on page 3-110
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asianbyitt

Price Asian options using implied trinomial tree (ITT)

Syntax

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

Description

`Price = asianbyitt(ITTTree,OptSpec,Strike,Settle,ExerciseDates)` prices Asian options using an implied trinomial tree (ITT).

`Price = asianbyitt( ___ ,AmericanOpt,AvgType,AvgPrice,AvgDate)` adds optional arguments for `AmericanOpt`, `AvgType`, `AvgPrice`, and `AvgDate`.

Examples

Price a Floating-Strike Asian Option Using an ITT Equity Tree

This example shows how to price a floating-strike Asian option using an ITT equity tree by loading the file deriv.mat, which provides ITTTree. The ITTTree structure contains the stock specification and time information needed to price the option.

```
load deriv.mat;

OptSpec = 'put';
Strike = NaN;
Settle = datetime(2006,1,1);
ExerciseDates = datetime(2007,1,1);

Price = asianbyitt(ITTTree, OptSpec, Strike, Settle, ExerciseDates)
```

```
Price = 1.0778
```

Input Arguments

`ITTTree` — Stock tree structure

structure

Stock tree structure, specified by using `itttree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a character vector or a cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price value
matrix of nonnegative integers

Option strike price value, specified with a nonnegative integer using a `NINST`-by-`1` matrix of strike price values.

To compute the value of a floating-strike Asian option, `Strike` must be specified as `NaN`. Floating-strike Asian options are also known as average strike options.

Data Types: `double`

`Settle` — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date for the Asian option, specified as a `NINST`-by-`1` matrix of settlement or trade dates using a datetime array, string array, or date character vectors.

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

To support existing code, `asianbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vector:

- For a European option, use a`NINST`-by-`1` matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. 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 a `NINST`-by-`1` vector, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `asianbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
`0` European (default) | integer with values `0` or `1`

(Optional) Option type, specified as `NINST`-by-`1` positive integer flags with values:

- `0` — European
- `1` — American

Data Types: `double`

`AvgType` — Average types
`arithmetic` (default) | character vector with values of `arithmetic` or `geometric`

Average types, specified as `arithmetic` for arithmetic average, or `geometric` for geometric average.

Data Types: `char`

`AvgPrice` — Average price of underlying asset at `Settle`
scalar numeric

Average price of underlying asset at `Settle`, specified as a scalar numeric.

Note Use this argument when `AvgDate` < `Settle`.

Data Types: `double`

`AvgDate` — Date averaging period begins
datetime scalar | string scalar | date character vector

Date averaging period begins, specified as a scalar datetime, string, or date character vector.

To support existing code, `asianbyitt` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`Price` — Expected prices for Asian options at time `0`
vector

Expected prices for Asian options at time 0, returned as a `NINST`-by-`1` vector. Pricing of Asian options is done using Hull-White (1993). Therefore, for these options there are no unique prices on the tree nodes except for the root node.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2007a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `asianbyitt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`itttree` | `instasian`

Topics

“Pricing Asian Options” on page 3-110
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19

External Websites

How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asianbyls

Price European or American Asian options using Monte Carlo simulations

Syntax

```
Price = asianbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)
Price = asianbyls( ___ ,Name,Value)

[Price,Paths,Times,Z] = asianbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates)
[Price,Paths,Times,Z] = asianbyls( ___ ,Name,Value)
```

Description

`Price = asianbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)` returns fixed- and floating-strike Asian option prices using the Longstaff-Schwartz model. `asianbyls` computes prices of European and American Asian options.

For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium.

To compute the value of a floating-strike Asian option, `Strike` should be specified as `NaN`. Fixed-strike Asian options are also known as average price options and floating-strike Asian options are also known as average strike options.

Note Alternatively, you can use the `Asian` object to price Asian options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = asianbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

`[Price,Paths,Times,Z] = asianbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)` returns fixed- and floating-strike Asian option `Price`, `Paths`, `Times`, and `Z` values using the Longstaff-Schwartz model. `asianbyls` computes prices of European and American Asian options.

`[Price,Paths,Times,Z] = asianbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of an Asian Option Using the Longstaff-Schwartz Model

Define the `RateSpec`.

```
Rates = 0.05;
StartDate = datetime(2013,1,1);
```

```
EndDate = datetime(2014,1,1);
RateSpec = intenvset('ValuationDate', StartDate, 'StartDates', StartDate, ...
'EndDates', EndDate,'Compounding', -1, 'Rates', Rates)

RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0500
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec` for the asset.

```
AssetPrice = 100;
Sigma = 0.2;
StockSpec = stockspec(Sigma, AssetPrice)

StockSpec = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.2000
          AssetPrice: 100
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the Asian `'call'` option.

```
Settle = datetime(2013,1,1);
ExerciseDates = datetime(2014,1,1);
Strike = 110;
OptSpec = 'call';
```

Compute the price for the European arithmetic average price for the Asian option using the Longstaff-Schwartz model.

```
NumTrials = 10000;
NumPeriods = 100;
AvgType = 'arithmetic';
Antithetic= true;
Price= asianbyls(RateSpec, StockSpec, OptSpec, Strike, Settle, ExerciseDates, ...
'NumTrials', NumTrials, 'NumPeriods', NumPeriods,'Antithetic', Antithetic, 'AvgType', AvgType)

Price = 1.9876
```

Input Arguments

RateSpec — Interest-rate term structure
structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset
structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with a value of `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a character vector

Data Types: `char`

`Strike` — Option strike price value
nonnegative scalar integer

Option strike price value, specified with a nonnegative scalar integer. To compute the value of a floating-strike Asian option, `Strike` should be specified as `NaN`. Floating-strike Asian options are also known as average strike options.

Data Types: `double`

`Settle` — Settlement or trade date
datetime scalar | string scalar | date character vector

Settlement or trade date for the Asian option, specified as a scalar datetime, string, or date character vector. By default, `asianbyls` calculates the price of Asian options based on averages that start on the settlement date.

To support existing code, `asianbyls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

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

To support existing code, `asianbyls` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = asianbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'NumTrials',NumTrials,'NumPeriods',NumPeriods,'Antithetic',Antithetic,'AvgType','arithmetic')`

`AmericanOpt` — Option type

`0` European (default) | scalar with value `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

Note For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `single` | `double`

`AvgType` — Average types

`arithmetic` (default) | character vector with values of `arithmetic` or `geometric`

Average types, specified as the comma-separated pair consisting of `'AvgType'` and `arithmetic` for arithmetic average, or `geometric` for geometric average.

Data Types: `char`

`AvgPrice` — Average price of underlying asset at `Settle`

scalar numeric

Average price of underlying asset at `Settle`, specified as the comma-separated pair consisting of `'AvgPrice'` and a scalar numeric value. The `AvgPrice` is assumed to be calculated in the time window starting at `AvgDate` and ending on `Settle`. In other words, the average is backward looking.

Note Use this argument when `AvgDate` < `Settle`.

Data Types: `double`

`AvgDate` — Date averaging period begins

datetime scalar | string scalar | date character vector

Date averaging period begins, specified as the comma-separated pair consisting of `'AvgDate'` and a scalar datetime, string, or date character vector.

To support existing code, `asianbyls` also accepts serial date numbers as inputs, but they are not recommended.

`NumTrials` — Simulation trials
`1000` (default) | numeric

Simulation trials, specified as the comma-separated pair consisting of `'NumTrials'` and a scalar number of independent sample paths.

Data Types: `double`

`NumPeriods` — Simulation periods per trial
`100` (default) | numeric

Simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` scalar numeric value. `NumPeriods` is considered only when pricing European Asian options. For American Asian options, `NumPeriods` is equal to the number of exercise days during the life of the option.

Data Types: `double`

`Z` — Dependent random variates
nonnegative integer

Dependent random variates used to generate the Brownian motion vector (that is, Wiener processes) that drive the simulation, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-2-by-`NumTrials` 3-D time series array.

Data Types: `single` | `double`

`Antithetic` — Indicator for antithetic sampling
`false` (default) | logical flag with value of `true` or `false`

Logical flag to indicate antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a value of `true` or `false`.

Data Types: `logical`

Output Arguments

`Price` — Expected price of Asian option
scalar

Expected price of the Asian option, returned as a 1-by-1 scalar.

`Paths` — Simulated paths of correlated state variables
vector

Simulated paths of correlated state variables, returned as a (`NumPeriods` + 1)-by-1-by-`NumTrials` 3-D time series array. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths
vector

Observation times associated with the simulated paths, returned as a (`NumPeriods` + 1)-by-1 column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates
vector

Dependent random variates, returned, if `Z` is specified as an optional input argument, the same value is returned. Otherwise, `Z` contains the random variates generated internally.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `asianbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`asiansensbyls` | `asianbycrr` | `intenvset` | `stockspec` | `asianbykv` | `asianbylevy` | `Asian`

Topics
“Pricing Asian Options” on page 3-110
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135

“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites
How to Price Asian Options Efficiently Using MATLAB (4 min 38 sec)

asianbystt

Price Asian options using standard trinomial tree

Syntax

```
Price = asianbystt(STTTree,OptSpec,Strike,Settle,ExerciseDates)
Price = asianbystt( ___ ,AmericanOpt,AvgType,AvgPrice,AvgDate)
```

Description

`Price = asianbystt(STTTree,OptSpec,Strike,Settle,ExerciseDates)` prices Asian options using a standard trinomial (STT) tree.

Note Alternatively, you can use the `Asian` object to price Asian options. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`Price = asianbystt( ___ ,AmericanOpt,AvgType,AvgPrice,AvgDate)` prices Asian options using a standard trinomial (STT) tree with optional arguments for `AmericanOpt`, `AvgType`, `AvgPrice`, and `AvgDate`.

Examples

Price an Asian Option Using the Standard Trinomial Tree Model

Create a `RateSpec`.

```
StartDates = datetime(2009,1,1);
EndDates = datetime(2013,1,1);
Rates = 0.035;
Basis = 1;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8694
            Rates: 0.0350
         EndTimes: 4
       StartTimes: 0
         EndDates: 735235
       StartDates: 733774
    ValuationDate: 733774
            Basis: 1
     EndMonthRule: 1
```

Create a StockSpec.

```
AssetPrice = 85;
Sigma = 0.15;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1500
         AssetPrice: 85
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Create an STTTree.

```
NumPeriods = 4;
TimeSpec = stttimespec(StartDates, EndDates, 4);
STTTree = stttree(StockSpec, RateSpec, TimeSpec)
```

```
STTTree = struct with fields:
       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 1 2 3 4]
         dObs: [733774 734139 734504 734869 735235]
        STree: {1x5 cell}
        Probs: {[3x1 double]  [3x3 double]  [3x5 double]  [3x7 double]}
```

Define the Asian option and compute the price.

```
Settle = datetime(2009,1,1);
ExerciseDates = [datetime(2012,1,1) ; datetime(2013,1,1)];
OptSpec = 'call';
Strike = 100;

Price = asianbystt(STTTree, OptSpec, Strike, Settle, ExerciseDates)
```

```
Price = 2×1

    1.6905
    2.6203
```

Input Arguments

STTTree — Stock tree structure for standard trinomial tree

structure

Stock tree structure for a standard trinomial tree, specified by using stttree.

Data Types: struct

OptSpec — Definition of option

character vector with value 'call' or 'put'

Definition of option, specified as `'call'` or `'put'` using a character vector.

Data Types: `char`

`Strike` — Option strike price value
matrix of nonnegative integers

Option strike price value, specified with a nonnegative integer using a `NINST`-by-`1` matrix of strike price values. To compute the value of a floating-strike Asian option, `Strike` should be specified as `NaN`. Floating-strike Asian options are also known as average strike options.

Data Types: `double`

`Settle` — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date for the Asian option, specified as a `NINST`-by-`1` vector of settlement or trade dates using date character vectors, or a datetime array.

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

To support existing code, `asianbystt` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or character vectors:

- For a European option, use a`NINST`-by-`1` matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. 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 a `NINST`-by-`1` vector of dates, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `asianbystt` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
`0` European (default) | scalar with values `[0,1]`

Option type, specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

Data Types: `single` | `double`

`AvgType` — Average types
`arithmetic` (default) | character vector with values of `arithmetic` or `geometric`

Average types, specified as `arithmetic` for arithmetic average, or `geometric` for geometric average.

Data Types: `char`

`AvgPrice` — Average price of underlying asset at `Settle`
scalar numeric

Average price of underlying asset at `Settle`, specified as a scalar numeric.

Note Use this argument when `AvgDate < Settle`.

Data Types: `double`

`AvgDate` — Date averaging period begins
datetime scalar | string scalar | date character vector

Date averaging period begins, specified as a scalar datetime, string, or date character vector.

To support existing code, `asianbystt` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`Price` — Expected prices for Asian options at time `0`
matrix

Expected prices for Asian options at time 0, returned as a `NINST`-by-`1` matrix. Pricing of Asian options is done using Hull-White (1993). Consequently, for these options there are no unique prices on the tree nodes with the exception of the root node.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2015b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `asianbystt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`stttimespec` | `stttree` | `sttprice` | `sttsens` | `instasian` | `Asian`

Topics

“Pricing Asian Options” on page 3-110
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19

External Websites

How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asiansensbyls

Calculate price and sensitivities for European or American Asian options using Monte Carlo simulations

Syntax

```
PriceSens = asiansensbyls(RateSpec,StockSpec,OptSpec,StrikeSettle,
ExerciseDates)
PriceSens = asiansensbyls( ___ ,Name,Value)

[PriceSens,Path,Times,Z] = asiansensbyls(RateSpec,StockSpec,OptSpec,Strike
Settle,ExerciseDates)
[PriceSens,Path,Times,Z] = asiansensbyls( ___ ,Name,Value)
```

Description

`PriceSens = asiansensbyls(RateSpec,StockSpec,OptSpec,StrikeSettle, ExerciseDates)` returns Asian option prices or sensitivities for fixed- and floating-strike Asian options using the Longstaff-Schwartz model. `asiansensbyls` supports European and American Asian options.

For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium.

To compute the value of a floating-strike Asian option, `Strike` should be specified as `NaN`. Fixed-strike Asian options are also known as average price options and floating-strike Asian options are also known as average strike options.

Note Alternatively, you can use the `Asian` object to calculate prices or sensitivities for Asian options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = asiansensbyls( ___ ,Name,Value)` returns Asian option prices or sensitivities for fixed- and floating-strike Asian options using optional name-value pair arguments and the Longstaff-Schwartz model.

`[PriceSens,Path,Times,Z] = asiansensbyls(RateSpec,StockSpec,OptSpec,Strike Settle,ExerciseDates)` returns Asian option prices or sensitivities (`PriceSens`, `Path`, `Times`, and `Z`) for fixed- and floating-strike Asian options using the Longstaff-Schwartz model.

`[PriceSens,Path,Times,Z] = asiansensbyls( ___ ,Name,Value)` returns Asian option prices or sensitivities (`PriceSens`, `Path`, `Times`, and `Z`) for fixed- and floating-strike Asian options using optional name-value pair arguments and the Longstaff-Schwartz model.

Examples

Compute the Price and Sensitivities of an Asian Option Using the Longstaff-Schwartz Model

Define the `RateSpec`.

```
Rates = 0.05;
StartDate = datetime(2013,1,1);
EndDate = datetime(2014,1,1);
RateSpec = intenvset('ValuationDate', StartDate, 'StartDates', StartDate, ...
'EndDates', EndDate,'Compounding', -1, 'Rates', Rates)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0500
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec` for the asset.

```
AssetPrice = 100;
Sigma = 0.2;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2000
         AssetPrice: 100
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the Asian `'call'` option.

```
Settle = datetime(2013,1,1);
ExerciseDates = datetime(2014,1,1);
Strike = 110;
OptSpec = 'call';
```

Compute the price for the European arithmetic average price and sensitivities for the Asian option using the Longstaff-Schwartz model.

```
NumTrials = 10000;
NumPeriods = 100;
AvgType = 'arithmetic';
Antithetic= true;
OutSpec = {'Price', 'Delta', 'Gamma'};
PriceSens = asiansensbyls(RateSpec, StockSpec, OptSpec, Strike, Settle, ExerciseDates, ...
'NumTrials', NumTrials, 'NumPeriods', NumPeriods,'Antithetic', Antithetic, 'AvgType', ...
AvgType,'OutSpec',OutSpec)
```

```
PriceSens = 1.9876
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a character vector.

Data Types: `char`

`Strike` — Option strike price value

nonnegative scalar integer

Option strike price value, specified with a nonnegative scalar integer. To compute the value of a floating-strike Asian option, `Strike` should be specified as `NaN`. Floating-strike Asian options are also known as average strike options.

Data Types: `double`

`Settle` — Settlement date or trade date

datetime scalar | string scalar | date character vector

Settlement date or trade date for the Asian option, specified as a scalar datetime, string, or date character vector.

To support existing code, `asiansensbyls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a 1-by-1 vector of dates. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a 1-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date

is listed, or if `ExerciseDates` is a `1`-by-`1` vector of dates, the option can be exercised between `Settle` and the single listed `ExerciseDates`.

To support existing code, `asiansensbyls` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = asiansensbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'NumTrials',NumTrials,'NumPeriods', NumPeriods,'Antithetic',Antithetic,'AvgType',AvgType,'OutSpec',{'All'})`

`AmericanOpt` — Option type

`0` European (default) | scalar with values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

Note For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `single` | `double`

`AvgType` — Average types

`arithmetic` (default) | character vector with values of `arithmetic` or `geometric`

Average types, specified as the comma-separated pair consisting of `'AvgType'` and `arithmetic` for arithmetic average, or `geometric` for geometric average.

Data Types: `char`

`AvgPrice` — Average price of underlying asset at `Settle`

scalar numeric

Average price of underlying asset at `Settle`, specified as the comma-separated pair consisting of `'AvgPrice'` and a scalar numeric value.

Note Use this argument when `AvgDate` < `Settle`.

Data Types: `double`

`AvgDate` — Date averaging period begins
datetime scalar | string scalar | date character vector

Date averaging period begins, specified as the comma-separated pair consisting of `'AvgDate'` and a scalar datetime, string, or data character vector.

To support existing code, `asiansensbyls` also accepts serial date numbers as inputs, but they are not recommended.

`NumTrials` — Simulation trials
`1000` (default) | numeric

Simulation trials, specified as the comma-separated pair consisting of `'NumTrials'` and a scalar number of independent sample paths.

Data Types: `double`

`NumPeriods` — Simulation periods per trial
`100` (default) | numeric

Simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar numeric value. `NumPeriods` is considered only when pricing European Asian options. For American Asian options, `NumPeriod` is equal to the number of exercise days during the life of the option.

Data Types: `double`

`Z` — Dependent random variates
nonnegative integer

Dependent random variates used to generate the Brownian motion vector (that is, Wiener processes) that drive the simulation, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-2-by-`NumTrials` 3-D time series array.

Data Types: `single` | `double`

`Antithetic` — Indicates antithetic sampling
`false` (default) | logical flag with value of `true` or `false`

Indicates antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a value of `true` or `false`.

Data Types: `logical`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected price or sensitivities of Asian option

scalar

Expected price or sensitivities (defined by `OutSpec`) of the Asian option, returned as a `1`-by-`1` array.

`Path` — Simulated paths of correlated state variables

vector

Simulated paths of correlated state variables, returned as a (`NumPeriods` + `1`)-by-`2`-by-`NumTrials` 3-D time series array. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths

vector

Observation times associated with simulated paths, returned as a (`NumPeriods` + `1`)-by-`1` column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates

vector

Dependent random variates, returned, if `Z` is specified as an optional input argument, the same value is returned. Otherwise, `Z` contains the random variates generated internally.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `asiansensbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`asianbyls` | `asianbycrr` | `stockspec` | `intenvset` | `asianbykv` | `asianbylevy` | `Asian`

Topics

“Pricing Asian Options” on page 3-110
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19

External Websites

How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asianbykv

Prices European geometric Asian options using Kemna-Vorst model

Syntax

```
Price = asianbykv(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)
```

Description

`Price = asianbykv(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)` returns prices of European geometric Asian options using the Kemna-Vorst model.

Note Alternatively, you can use the `Asian` object to price Asian options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Compute the Price of an Asian Option Using the Kemna-Vorst Model

Define the `RateSpec`.

```
StartDates = datetime(2013,1,1);
EndDates = datetime(2014,1,1);
Rates = 0.035;
Basis = 1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates, ...
'EndDates', EndDates,'Rates', Rates,  'Compounding', -1, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9656
            Rates: 0.0350
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec` for the asset.

```
AssetPrice = 100;
Sigma = 0.15;
DivType = 'continuous';
DivAmounts = 0.03;
StockSpec = stockspec(Sigma, AssetPrice, DivType, DivAmounts)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1500
         AssetPrice: 100
       DividendType: {'continuous'}
    DividendAmounts: 0.0300
    ExDividendDates: []
```

Define the Asian `'call'` and `'put'` options.

```
Strike = 102;
OptSpec = {'put'; 'call'};
Settle = datetime(2013,1,1);
Maturity = datetime(2013,4,1);
```

Compute the European geometric Average Price for the Asian option using the Kemna-Vorst model.

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

```
Price = 2×1

    2.8881
    0.9210
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

The annualized continuously compounded interest-rate term structure specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors

Definition of option, specified as `'call'` or `'put'` using a `NINST`-by-`1` cell array of character vectors.

Data Types: `cell` | `char`

`Strike` — Option strike price values

nonnegative integer | vector of nonnegative integers

Option strike price values, specified with nonnegative integers using a `NINST`-by-`1` vector.

Data Types: `single` | `double`

`Settle` — Settlement dates or trade dates
datetime array | string array | date character vector

Settlement dates or trade dates for the Asian option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `asianbykv` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — European option exercise dates
datetime array | string array | date character vector

European option exercise dates, specified as `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. For a European option, there is only one `ExerciseDates` on the option expiry date.

To support existing code, `asianbykv` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`Price` — Expected prices of an Asian option
vector

Expected prices of the Asian option, returned as an `NINST`-by-`1` vector.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `asianbykv` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`asiansensbykv` | `asianbycrr` | `intenvset` | `stockspec` | `asianbyls` | `asianbylevy` | `Asian`

Topics

“Pricing Asian Options” on page 3-110
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asiansensbykv

Calculate prices or sensitivities of European geometric Asian options using Kemna-Vorst model

Syntax

```
PriceSens = asiansensbykv(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates)
PriceSens = asiansensbykv( ___ ,Name,Value)
```

Description

`PriceSens = asiansensbykv(RateSpec,StockSpec,OptSpec,Strike,Settle, ExerciseDates)` returns prices or sensitivities of European geometric Asian options using Kemna-Vorst model.

Note Alternatively, you can use the `Asian` object to calculate prices or sensitivities for Asian options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = asiansensbykv( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price and Sensitivities of an Asian Option Using the Kemna-Vorst Model

Define the `RateSpec`.

```
StartDates = datetime(2013,1,1);
EndDates = datetime(2014,1,1);
Rates = 0.035;
Basis = 1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates, ...
'EndDates', EndDates,'Rates', Rates,  'Compounding', -1, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9656
            Rates: 0.0350
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec` for the asset.

```
AssetPrice = 100;
Sigma = 0.15;
DivType = 'continuous';
DivAmounts = 0.03;
StockSpec = stockspec(Sigma, AssetPrice, DivType, DivAmounts)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1500
         AssetPrice: 100
       DividendType: {'continuous'}
    DividendAmounts: 0.0300
    ExDividendDates: []
```

Define the Asian `'call'` and `'put'` options.

```
Strike = 102;
OptSpec = {'put'; 'call'};
Settle = datetime(2013,1,1);
ExerciseDates = datetime(2014,1,1);
```

Compute the European geometric Average Price and sensitivities for the Asian option using the Kemna-Vorst model.

```
OutSpec = {'Price', 'Delta', 'Gamma'};
PriceSens = asiansensbykv(RateSpec, StockSpec, OptSpec, Strike,...
Settle, ExerciseDates,'OutSpec', OutSpec)
```

```
PriceSens = 2×1

    4.3871
    2.5163
```

Input Arguments

RateSpec — Interest-rate term structure

structure

The annualized continuously compounded interest-rate term structure specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

Data Types: `struct`

OptSpec — Definition of option
character vector with values 'call' or 'put' | cell array of character vectors

Definition of option, specified as 'call' or 'put' using a NINST-by-1 cell array of character vectors.

Data Types: cell | char

Strike — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values, specified with nonnegative integers using a NINST- by-1 vector.

Data Types: single | double

Settle — Settlement dates or trade dates
datetime array | string array | date character vector

Settlement dates or trade dates for the Asian option, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, asiansensbykv also accepts serial date numbers as inputs, but they are not recommended.

ExerciseDates — European option exercise dates
datetime array | string array | date character vector

European option exercise dates, specified as NINST-by-1 vector using a datetime array, string array, or date character vectors. For a European option, there is only one ExerciseDates on the option expiry date.

To support existing code, asiansensbykv also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: PriceSens = asiansensbykv(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'OutSpec',{'All'})

OutSpec — Define outputs
{'Price'} (default) | character vector with values: 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta' and 'All'. | cell array of character vectors with values: 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta' and 'All'.

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and a NOUT- by-1 or 1-by-NOUT cell array of character vectors with possible values of 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'.

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:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities of the Asian option

vector

Expected prices or sensitivities (defined by `OutSpec`) of the Asian option, returned as an `1`-by-`1` vector. If the `OutSpec` is not specified only price is returned.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `asiansensbykv` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`asianbykv` | `asianbycrr` | `stockspec` | `intenvset` | `asianbyls` | `asianbylevy` | `Asian`

Topics

“Pricing Asian Options” on page 3-110
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Asian Option” on page 3-34

“Supported Equity Derivative Functions” on page 3-19

External Websites
How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asianbylevy

Price of European arithmetic Asian options using Levy model

Syntax

```
Price = asianbylevy(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)
```

Description

`Price = asianbylevy(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)` returns European arithmetic average pricing for Asian options using the Levy model.

Note Alternatively, you can use the `Asian` object to price Asian options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Compute the Price of an Asian Option Using the Levy Model

Define the `RateSpec`.

```
Rates = 0.07;
StartDates = datetime(2013,1,1);
EndDates = datetime(2014,1,1);
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates, 'EndDates', ...
EndDates, 'Rates', Rates, 'Compounding', -1)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9324
            Rates: 0.0700
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec` for the asset.

```
AssetPrice = 6.8;
Sigma = 0.14;
DivType = 'continuous';
DivAmounts = 0.09;
StockSpec = stockspec(Sigma, AssetPrice, DivType, DivAmounts)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1400
         AssetPrice: 6.8000
       DividendType: {'continuous'}
    DividendAmounts: 0.0900
    ExDividendDates: []
```

Define two options for `'call'` and `'put'`.

```
Settle = datetime(2013,1,1);
Maturity = datetime(2013,7,1);
Strike = 6.9;
OptSpec = {'call'; 'put'};
```

Compute the European arithmetic average price for the Asian option using the Levy model.

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

```
Price = 2×1

    0.0944
    0.2237
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

Data Types: `struct`

OptSpec — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option, specified as `'call'` or `'put'` using a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

Strike — Option strike price values

nonnegative integer | vector of nonnegative integers

Option strike price values, specified with nonnegative integers as a `NINST`-by-`1` vector.

Data Types: `single` | `double`

`Settle` — Settlement dates or trade dates
datetime array | string array | date character vector

Settlement dates or trade dates for the Asian option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `asianbylevy` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. For a European option, there is only one `ExerciseDates` on the option expiry date.

To support existing code, `asianbylevy` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`Price` — Expected prices of Asian option
vector

Expected prices of the Asian option, returned as a `NINST`-by-`1` vector.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2013b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `asianbylevy` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`asiansensbylevy` | `asianbycrr` | `intenvset` | `stockspec` | `asianbyls` | `asianbykv` | `Asian`

Topics

“Pricing Asian Options” on page 3-110
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asiansensbylevy

Calculate prices or sensitivities of European arithmetic Asian options using Levy model

Syntax

```
PriceSens = asiansensbylevy(RateSpec,StockSpec,OptSpec,StrikeSettle,
ExerciseDates)
PriceSens = asiansensbylevy( ___ ,Name,Value)
```

Description

`PriceSens = asiansensbylevy(RateSpec,StockSpec,OptSpec,StrikeSettle, ExerciseDates)` returns European average pricing or sensitivities for arithmetic Asian options using the Levy model.

Note Alternatively, you can use the `Asian` object to calculate prices or sensitivities for Asian options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = asiansensbylevy( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price and Sensitivities of an Asian Option Using the Levy Model

Define the `RateSpec`.

```
Rates = 0.07;
StartDates = datetime(2013,1,1);
EndDates = datetime(2014,1,1);
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates, 'EndDates', ...
EndDates, 'Rates', Rates, 'Compounding', -1)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9324
            Rates: 0.0700
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec` for the asset.

```
AssetPrice = 6.8;
Sigma = 0.14;
DivType = 'continuous';
DivAmounts = 0.09;
StockSpec = stockspec(Sigma, AssetPrice, DivType, DivAmounts)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1400
         AssetPrice: 6.8000
       DividendType: {'continuous'}
    DividendAmounts: 0.0900
    ExDividendDates: []
```

Define two options for a `'call'` and `'put'`.

```
Settle = datetime(2013,1,1);
ExerciseDates = datetime(2014,1,1);
Strike = 6.9;
OptSpec = {'call'; 'put'};
```

Compute the European arithmetic average price and sensitivities for the Asian option using the Levy model.

```
OutSpec = {'Price', 'Delta', 'Gamma'};
PriceSens = asiansensbylevy(RateSpec, StockSpec, OptSpec, Strike,...
Settle, ExerciseDates,'OutSpec', OutSpec)
```

```
PriceSens = 2×1

    0.1358
    0.2921
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option, specified as `'call'` or `'put'` using a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values, specified with nonnegative integers using a `NINST`-by-`1` vector.

Data Types: `single` | `double`

`Settle` — Settlement dates or trade dates
datetime array | string array | date character vector

Settlement dates or trade dates for the Asian option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `asiansensbylevy` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. For a European option, there is only one `ExerciseDates` on the option expiry date.

To support existing code, `asiansensbylevy` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = asiansensbylevy(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'OutSpec',{'All'})`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values: `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`. | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities of Asian option

vector

Expected prices or sensitivities (defined by `OutSpec`) of the Asian option, returned as an `1`-by-`1` vector. If the `OutSpec` is not specified only the price is returned.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `asiansensbylevy` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`asianbykv` | `asianbycrr` | `stockspec` | `intenvset` | `asianbyls` | `asianbykv` | `Asian`

Topics

“Pricing Asian Options” on page 3-110
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Asian Option” on page 3-34

“Supported Equity Derivative Functions” on page 3-19

External Websites
How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asianbyhhm

Price European discrete arithmetic fixed Asian options using Haug, Haug, Margrabe model

Syntax

```
Price = asianbyhhm(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)
Price = asianbyhhm( ___ ,Name,Value)
```

Description

`Price = asianbyhhm(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)` prices European discrete arithmetic fixed Asian options using the Haug, Haug, Margrabe model.

`Price = asianbyhhm( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price an Asian Option with Averaging Period Starting Before the Settle Date

Define the Asian option parameters.

```
AssetPrice = 100;
Strike = 95;
Rates = 0.1;
Sigma = 0.15;
Settle = datetime(2013,4,1);
Maturity = datetime(2013,10,1);
```

Create a `RateSpec` using the `intenvset` function.

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

Create a `StockSpec` for the underlying asset using the `stockspec` function.

```
DividendType = 'Continuous';
DividendAmounts = 0.05;

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

Calculate the price of the Asian option using the Haug, Haug, Margrabe approximation. Assume that the averaging period has started before the `Settle` date.

```
OptSpec = 'Call';
ExerciseDates = datetime(2013,10,1);
NumFixings = 12;
AvgDate = datetime(2013,1,1);
AvgPrice = 100;

Price = asianbyhhm(RateSpec, StockSpec, OptSpec, Strike, Settle, ExerciseDates, ...
'NumFixings', NumFixings, 'AvgDate', AvgDate, 'AvgPrice', AvgPrice)
```

```
Price = 5.8216
```

Price an Asian Option with Averaging Period Starting After the Settle Date

Define the Asian option parameters.

```
AssetPrice = 100;
Strike = 95;
Rates = 0.1;
Sigma = 0.15;
Settle = datetime(2013,4,1);
Maturity = datetime(2013,10,1);
```

Create a `RateSpec` using the `intenvset` function.

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

Create a `StockSpec` for the underlying asset using the `stockspec` function.

```
DividendType = 'Continuous';
DividendAmounts = 0.05;

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

Calculate the price of the Asian option using the Haug, Haug, Margrabe approximation. Assume that the averaging period starts after the `Settle` date.

```
OptSpec = 'Call';
ExerciseDates = datetime(2013,10,1);
NumFixings = 15;
AvgDate = datetime(2013,1,1);

Price = asianbyhhm(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates, ...
'NumFixings',NumFixings,'AvgDate',AvgDate)
```

```
Price = 1.3785e-07
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

stockspec can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in StockSpec, dividends are assumed to be 0.

Data Types: struct

OptSpec — Definition of option
character vector with value 'call' or 'put' | cell array of character vectors with values 'call' or 'put' | string array with values "call" or "put"

Definition of option, specified as 'call' or 'put' using a character vector, cell array of character vectors, or string array.

Data Types: char | cell | string

Strike — Option strike price value
nonnegative integer | vector of nonnegative integers

Option strike price value, specified with a nonnegative integer using a NINST-by-1 vector of strike price values.

Data Types: double

Settle — Settlement dates or trade dates
datetime array | string array | date character vector

Settlement date or trade date for the Asian option, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, asianbyhhm also accepts serial date numbers as inputs, but they are not recommended.

ExerciseDates — European option exercise dates
datetime array | string array | date character vector

European option exercise dates, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one ExerciseDates on the option expiry date.

To support existing code, asianbyhhm also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: Price = asianbyhhm(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'NumFixings',15)

`AvgPrice` — Average price of underlying asset at the `Settle` date
vector

Average price of underlying asset at the `Settle` date, specified as the comma-separated pair consisting of `'AvgPrice'` and a `NINST`-by-`1` vector.

Note Use the `AvgPrice` argument when `AvgDate` < `Settle`.

Data Types: `double`

`AvgDate` — Date averaging period begins
datetime array | string array | date character vector

Date averaging period begins, specified as the comma-separated pair consisting of `'AvgDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `asianbyhhm` also accepts serial date numbers as inputs, but they are not recommended.

`NumFixings` — Total number of fixings or averaging points
`10` (default) | vector

Total number of fixings or averaging points, specified as the comma-separated pair consisting of `'NumFixings'` and a `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected prices for fixed Asian options
vector

Expected prices for fixed Asian options, returned as a `NINST`-by-`1` vector.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2018a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `asianbyhhm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Haug, E. G. *The Complete Guide to Option Pricing Formulas.* McGraw-Hill Education, 2007.

See Also

`asiansensbyhhm` | `asianbytw` | `asianbykv` | `asianbyls` | `stockspec` | `intenvset` | `asianbycrr` | `asianbylevy`

Topics

“Pricing Asian Options” on page 3-110
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asiansensbyhhm

Calculate price and sensitivities of European discrete arithmetic fixed Asian options using Haug, Haug, Margrabe model

Syntax

```
PriceSens = asiansensbyhhm(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates)
PriceSens = asiansensbyhhm( ___ ,Name,Value)
```

Description

`PriceSens = asiansensbyhhm(RateSpec,StockSpec,OptSpec,Strike,Settle, ExerciseDates)` calculates prices and sensitivities for European discrete arithmetic fixed Asian options using the Haug, Haug, Margrabe model.

`PriceSens = asiansensbyhhm( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute Price and Sensitivities for Asian Option with Averaging Period Starting Before the Settle Date

Define the Asian option parameters.

```
AssetPrice = 100;
Strike = 95;
Rates = 0.1;
Sigma = 0.15;
Settle = datetime(2013,4,1);
Maturity = datetime(2013,10,1);
```

Create a `RateSpec` using the `intenvset` function.

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

Create a `StockSpec` for the underlying asset using the `stockspec` function.

```
DividendType = 'Continuous';
DividendAmounts = 0.05;

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

Calculate the price and sensitivities of the Asian option using the Haug, Haug, Margrabe approximation. Assume that the averaging period has started before the `Settle` date.

```
OptSpec = 'Call';
ExerciseDates = datetime(2013,10,1);
NumFixings = 12;
AvgDate = datetime(2013,1,1);
```

```
AvgPrice = 100;
OutSpec = {'Price','Delta','Gamma'};

[Price,Delta,Gamma] = asiansensbyhhm(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates, ...
'NumFixings',NumFixings,'AvgDate',AvgDate,'AvgPrice',AvgPrice,'OutSpec',OutSpec)

Price = 5.8216

Delta = 0.5907

Gamma = 0.0143
```

Compute Price and Sensitivities for Asian Option with Averaging Period Starting After the Settle Date

Define the Asian option parameters.

```
AssetPrice = 100;
Strike = 95;
Rates = 0.1;
Sigma = 0.15;
Settle = 'Apr-1-2013';
Maturity = 'Oct-1-2013';
```

Create a `RateSpec` using the `intenvset` function.

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

Create a `StockSpec` for the underlying asset using the `stockspec` function.

```
DividendType = 'Continuous';
DividendAmounts = 0.05;

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

Calculate the price and sensitivities of the Asian option using the Haug, Haug, Margrabe approximation. Assume that the averaging period started after the `Settle` date.

```
OptSpec = 'Call';
ExerciseDates = 'Oct-1-2013';
NumFixings = 15;
AvgDate = 'Jan-1-2013';
OutSpec = {'Price','Delta','Gamma'};

[Price,Delta,Gamma] = asiansensbyhhm(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates, ...
'NumFixings',NumFixings,'AvgDate',AvgDate,'OutSpec',OutSpec)

Price = 1.3785e-07

Delta = 1.1438e-07

Gamma = 9.0830e-08
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of option, specified as `'call'` or `'put'` using a character vector, cell array of character vectors, or string array.

Data Types: `char` | `cell` | `string`

`Strike` — Option strike price value

nonnegative integer | vector of nonnegative integers

Option strike price value, specified with a nonnegative integer using a `NINST`-by-`1` vector of strike price values.

Data Types: `double`

`Settle` — Settlement dates or trade dates

datetime array | string array | date character vector

Settlement date or trade date for the Asian option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `asiansensbyhhm` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — European option exercise dates

datetime array | string array | date character vector

European option exercise dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one `ExerciseDates` on the option expiry date.

To support existing code, asiansensbyhhm also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: PriceSens = asiansensbyhhm(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'OutSpec',{'All'},'NumFixings',15)

OutSpec — Define outputs
{'Price'} (default) | character vector with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All' | cell array of character vectors with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All' | string array with values "Price", "Delta", "Gamma", "Vega", "Lambda", "Rho", "Theta", and "All"

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and a NOUT- by-1 or 1-by-NOUT cell array of character vectors or string array with possible values of 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'.

OutSpec = {'All'} specifies that the output is Delta, Gamma, Vega, Lambda, Rho, Theta, and Price, in that order. This is the same as specifying OutSpec to include each sensitivity:

Example: OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}

Data Types: char | cell | string

AvgPrice — Average price of underlying asset at the Settle date
vector

Average price of underlying asset at the Settle date, specified as the comma-separated pair consisting of 'AvgPrice' and a NINST-by-1 vector.

Note Use the AvgPrice argument when AvgDate < Settle.

Data Types: double

AvgDate — Date averaging period begins
character vector | datetime | string array

Date averaging period begins, specified as the comma-separated pair consisting of 'AvgDate' and a NINST-by-1 vector using character vectors, datetimes, or string arrays.

To support existing code, asiansensbyhhm also accepts serial date numbers as inputs, but they are not recommended.

NumFixings — Total number of fixings or averaging points
10 (default) | vector

Total number of fixings or averaging points, specified as the comma-separated pair consisting of `'NumFixings'` and a `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`PriceSens` — Expected prices or sensitivities for fixed Asian options

vector

Expected prices or sensitivities for fixed Asian options, returned as a `NINST`-by-`1` vector. `asianbyhhm` calculates prices of European arithmetic fixed (average price) Asian options with discretely monitoring.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `asiansensbyhhm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Haug, E. G. *The Complete Guide to Option Pricing Formulas.* McGraw-Hill Education, 2007.

See Also

`asianbyhhm` | `asianbytw` | `asianbykv` | `asianbyls` | `stockspec` | `intenvset` | `asianbycrr` | `asianbylevy`

Topics

“Pricing Asian Options” on page 3-110
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19

External Websites

How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asianbytw

Price European arithmetic fixed Asian options using Turnbull-Wakeman model

Syntax

```
Price = asianbytw(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)
Price = asianbytw( ___ ,Name,Value)
```

Description

`Price = asianbytw(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)` prices European arithmetic fixed Asian options using the Turnbull-Wakeman model.

Note Alternatively, you can use the `Asian` object to price Asian options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = asianbytw( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price an Asian Option with Averaging Period Before the Settle Date

Define the Asian option parameters.

```
AssetPrice = 100;
Strike = 95;
Rates = 0.1;
Sigma = 0.15;
Settle = datetime(2013,4,1);
Maturity = datetime(2013,10,1);
```

Create a `RateSpec` using the `intenvset` function.

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

Create a `StockSpec` for the underlying asset using the `stockspec` function.

```
DividendType = 'Continuous';
DividendAmounts = 0.05;

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

Calculate the price of the Asian option using the Turnbull-Wakeman approximation. Assume that the averaging period has started before the `Settle` date.

```
OptSpec = 'Call';
ExerciseDates = datetime(2013,10,1);
```

```
AvgDate = datetime(2013,1,1);
AvgPrice = 100;

Price = asianbytw(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates, ...
'AvgDate',AvgDate,'AvgPrice',AvgPrice)

Price = 5.6731
```

Price an Asian Option with Averaging Period After the Settle Date

Define the Asian option parameters.

```
AssetPrice = 100;
Strike = 95;
Rates = 0.1;
Sigma = 0.15;
Settle = datetime(2013,4,1);
Maturity = datetime(2013,10,1);
```

Create a `RateSpec` using the `intenvset` function.

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

Create a `StockSpec` for the underlying asset using the `stockspec` function.

```
DividendType = 'Continuous';
DividendAmounts = 0.05;

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

Calculate the price of the Asian option using the Turnbull-Wakeman approximation. Assume that the averaging period starts after the `Settle` date.

```
OptSpec = 'Call';
ExerciseDates = datetime(2013,10,1);
AvgDate = datetime(2013,1,1);

Price = asianbytw(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates, ...
'AvgDate',AvgDate)

Price = 1.0774e-08
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of option, specified as `'call'` or `'put'` using a character vector, cell array of character vectors, or string array.

Data Types: `char` | `cell` | `string`

`Strike` — Option strike price value
nonnegative integer | vector of nonnegative integer

Option strike price value, specified with a nonnegative integer using a `NINST`-by-`1` vector of strike price values.

Data Types: `double`

`Settle` — Settlement dates or trade dates
datetime array | string array | date character vector

Settlement date or trade date for the Asian option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `asianbytw` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — European option exercise dates
datetime array | string array | date character vector

European option exercise dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one `ExerciseDates` on the option expiry date.

To support existing code, `asianbytw` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = asianbytw(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'AvgPrice',1500)`

`AvgPrice` — Average price of underlying asset at the `Settle` date
vector

Average price of underlying asset at the `Settle` date, specified as the comma-separated pair consisting of `'AvgPrice'` and a `NINST`-by-`1` vector.

Note Use the `AvgPrice` argument when `AvgDate` < `Settle`.

Data Types: `double`

`AvgDate` — Date averaging period begins
datetime array | string array | date character vector

Date averaging period begins, specified as the comma-separated pair consisting of `'AvgDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `asianbytw` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`Price` — Expected prices for fixed Asian options
vector

Expected prices for Asian options, returned as a `NINST`-by-`1` vector.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2018a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `asianbytw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Turnbull, S. M. and L. M. Wakeman. "A Quick Algorithm for Pricing European Average Options."*Journal of Financial and Quantitative Analysis* Vol. 26(3).1991, pp. 377-389.

See Also

`asiansensbytw` | `asianbyhhm` | `asianbykv` | `asianbyls` | `stockspec` | `intenvset` | `asianbycrr` | `asianbylevy` | `Asian`

Topics

“Pricing Asian Options” on page 3-110
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

asiansensbytw

Calculate price and sensitivities of European fixed arithmetic Asian options using Turnbull-Wakeman model

Syntax

```
PriceSens = asiansensbytw(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates)
PriceSens = asiansensbytw( ___ ,Name,Value)
```

Description

`PriceSens = asiansensbytw(RateSpec,StockSpec,OptSpec,Strike,Settle, ExerciseDates)` calculates prices and sensitivities for European fixed arithmetic Asian options using the Turnbull-Wakeman model.

Note Alternatively, you can use the `Asian` object to calculate prices or sensitivities for Asian options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = asiansensbytw( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute Price and Sensitivities for Asian Option with Averaging Period Before the Settle Date

Define the Asian option parameters.

```
AssetPrice = 100;
Strike = 95;
Rates = 0.1;
Sigma = 0.15;
Settle = datetime(2013,4,1);
Maturity = datetime(2013,10,1);
```

Create a `RateSpec` using the `intenvset` function.

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

Create a `StockSpec` for the underlying asset using the `stockspec` function.

```
DividendType = 'Continuous';
DividendAmounts = 0.05;

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

Calculate the price and sensitivities of the Asian option using the Turnbull-Wakeman approximation. Assume that the averaging period has started before the `Settle` date.

```
OptSpec = 'Call';
ExerciseDates = datetime(2013,10,1);
AvgDate = datetime(2013,1,1);
AvgPrice = 100;
OutSpec = {'Price','Delta','Gamma'};

[Price,Delta,Gamma] = asiansensbytw(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates, ...
'AvgDate',AvgDate,'AvgPrice',AvgPrice,'OutSpec',OutSpec)

Price = 5.6731

Delta = 0.5995

Gamma = 0.0135
```

Compute Price and Sensitivities for Asian Option with Averaging Period After the Settle Date

Define the Asian option parameters.

```
AssetPrice = 100;
Strike = 95;
Rates = 0.1;
Sigma = 0.15;
Settle = 'Apr-1-2013';
Maturity = 'Oct-1-2013';
```

Create a `RateSpec` using the `intenvset` function.

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

Create a `StockSpec` for the underlying asset using the `stockspec` function.

```
DividendType = 'Continuous';
DividendAmounts = 0.05;

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

Calculate the price and sensitivities of the Asian option using the Turnbull-Wakeman approximation. Assume that the averaging period starts after the `Settle` date.

```
OptSpec = 'Call';
ExerciseDates = 'Oct-1-2013';
AvgDate = 'Jan-1-2013';
OutSpec = {'Price','Delta','Gamma'};

[Price,Delta,Gamma] = asiansensbytw(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates, ...
'AvgDate',AvgDate,'OutSpec',OutSpec)

Price = 1.0774e-08

Delta = 1.0380e-08
```

```
Gamma = 9.6246e-09
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

Data Types: `struct`

OptSpec — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of option, specified as `'call'` or `'put'` using a character vector, cell array of character vectors, or string array.

Data Types: `char` | `cell` | `string`

Strike — Option strike price value

nonnegative integer | vector of nonnegative integers

Option strike price value, specified with a nonnegative integer using a `NINST`-by-`1` vector of strike price values.

Data Types: `double`

Settle — Settlement dates or trade dates

datetime array | string array | date character vector

Settlement date or trade date for the Asian option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `asiansensbytw` also accepts serial date numbers as inputs, but they are not recommended.

ExerciseDates — European option exercise dates

datetime array | string array | date character vector

European option exercise dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one `ExerciseDates` on the option expiry date.

To support existing code, `asiansensbytw` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = asiansensbytw(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'OutSpec',{'All'})`

`OutSpec` — Define outputs

`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | string array with values `"Price"`, `"Delta"`, `"Gamma"`, `"Vega"`, `"Lambda"`, `"Rho"`, `"Theta"`, and `"All"`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors or string array with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`OutSpec = {'All'}` specifies that the output is `Delta`, `Gamma`, `Vega`, `Lambda`, `Rho`, `Theta`, and `Price`, in that order. This is the same as specifying `OutSpec` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell` | `string`

`AvgPrice` — Average price of underlying asset at the `Settle` date

vector

Average price of underlying asset at the `Settle` date, specified as the comma-separated pair consisting of `'AvgPrice'` and a `NINST`-by-`1` vector.

Note Use the `AvgPrice` argument when `AvgDate < Settle`.

Data Types: `double`

`AvgDate` — Date averaging period begins

datetime array | string array | date character vector

Date averaging period begins, specified as the comma-separated pair consisting of `'AvgDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `asiansensbytw` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`PriceSens` — Expected prices or sensitivities for fixed Asian options

vector

Expected prices or sensitivities for fixed Asian options, returned as a `NINST`-by-`1` vector. `asiansensbytw` calculates prices of European arithmetic fixed (average price) Asian options.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `asiansensbytw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Turnbull, S. M. and L. M. Wakeman. "A Quick Algorithm for Pricing European Average Options."*Journal of Financial and Quantitative Analysis* Vol. 26(3).1991, pp. 377-389.

See Also

`asianbyhhm` | `asianbytw` | `asianbykv` | `asianbyls` | `stockspec` | `intenvset` | `asianbycrr` | `asianbylevy` | `Asian`

Topics

“Pricing Asian Options” on page 3-110

“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Asian Option” on page 3-34
“Supported Equity Derivative Functions” on page 3-19

External Websites
How to Price Asian Options Efficiently Using MATLAB (4 min 37 sec)

assetbybls

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

Syntax

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

Description

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

Note Alternatively, you can use the `Binary` object to price digital options. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

Examples

Compute Asset-Or-Nothing Digital Option Prices Using the Black-Scholes Option Pricing Model

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%. First, create the `RateSpec`.

```
Settle = datetime(2008,11,3);
Maturity = datetime(2009,1,30);
Rates = 0.045;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9893
            Rates: 0.0450
         EndTimes: 0.2391
       StartTimes: 0
         EndDates: 733803
       StartDates: 733715
    ValuationDate: 733715
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec`.

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

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2200
         AssetPrice: 97.5000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the put options.

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

Calculate the price.

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

```
Paon = 2×1

   33.7666
   26.9662
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `assetbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `assetbybls` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as an `NINST`-by-`1` vector.

Data Types: `char` | `cell`

`Strike` — Pay-off strike value
vector

Pay-off strike value, specified as an `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected prices for asset-or-nothing option
vector

Expected prices for asset-or-nothing option, returned as a `NINST`-by-`1` vector.

Version History

Introduced in R2009a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `assetbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`assetsensbybls` | `cashbybls` | `gapbybls` | `supersharebybls` | `Binary`

Topics

“Pricing Using the Black-Scholes Model” on page 3-82
“Digital Option” on page 3-26
“Supported Equity Derivative Functions” on page 3-19

assetsensbybls

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

Syntax

```
PriceSens = assetsensbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike)
PriceSens = assetsensbybls( ___ ,Name,Value)
```

Description

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

Note Alternatively, you can use the `Binary` object to calculate price or sensitivities for digital options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = assetsensbybls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute Asset-Or-Nothing Digital Option Prices and Sensitivities Using the Black-Scholes Option Pricing Model

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%. First, create the `RateSpec`.

```
Settle = datetime(2008,11,3);
Maturity = datetime(2009,1,30);
Rates = 0.045;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9893
            Rates: 0.0450
         EndTimes: 0.2391
       StartTimes: 0
         EndDates: 733803
       StartDates: 733715
    ValuationDate: 733715
```

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

Define the `StockSpec`.

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

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2200
         AssetPrice: 97.5000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

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 = 2×1

   -3.0833
   -2.8337


Price = 2×1

   33.7666
   26.9662


Gamma = 2×1

    0.0941
    0.1439
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset
structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors

To support existing code, `assetsensbybls` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `assetsensbybls` also accepts serial date numbers as inputs, but they are not recommended.

OptSpec — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as an `NINST`-by-`1` vector.

Data Types: `char` | `cell`

Strike — Pay-off strike value
vector

Pay-off strike value, specified as an `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities for asset-or-nothing option
vector

Expected prices or sensitivities (defined using `OutSpec`) for asset-or-nothing option, returned as a `NINST`-by-`1` vector.

Version History

Introduced in R2009a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `assetsensbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`assetbybls` | `cashbybls` | `gapbybls` | `supersharebybls` | `Binary`

Topics
“Pricing Using the Black-Scholes Model” on page 3-82
“Digital Option” on page 3-26
“Supported Equity Derivative Functions” on page 3-19

barrierbycrr

Price barrier option from Cox-Ross-Rubinstein binomial tree

Syntax

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

Description

`[Price,PriceTree] = barrierbycrr(CRRTree,OptSpec,Strike,Settle,AmericanOpt, ExerciseDates,BarrierSpec,Barrier)` calculates prices for barrier options using a Cox-Ross-Rubinstein binomial tree.

Note Alternatively, you can use the `Barrier` object to price Barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = barrierbycrr( ___ ,Rebate,Options)` adds optional arguments for `Rebate` and `Options`.

Examples

Price a Barrier Option Using a CRR Binomial Tree

This example shows how to price a barrier option using a CRR binomial tree by loading 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;

OptSpec = 'Call';
Strike = 105;
Settle = datetime(2003,1,1);
ExerciseDates = datetime(2006,1,1);
AmericanOpt = 1;
BarrierSpec = 'UI';
Barrier = 102;

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

Price = 12.1272
```

Input Arguments

`CRRTree` — Stock tree structure

structure

Stock tree structure, specified by using `crrtree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of an option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vector values.

Data Types: `char` | `cell`

`Strike` — Option strike price value

integer

Option strike price value for a European or an American Option, specified as `NINST`-by-`1` matrix of integers. Each row is the schedule for one option.

Data Types: `double`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the barrier option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every barrier is set to the `ValuationDate` of the stock tree. The barrier argument `Settle` is ignored.

To support existing code, `barrierbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vector:

- For a European option, use a `1`-by-`1` matrix of dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date which is the maturity of the instrument.
- For an American option, use a `1`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-`1`, the option can be exercised between `ValuationDate` of the stock tree and the single listed date in `ExerciseDates`.

To support existing code, `barrierbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type

Option type values `0` or `1`

Option type, specified as `NINST`-by-`1` matrix of integer flags with values:

- `0` — European
- `1` — American

Data Types: `double`

`BarrierSpec` — Barrier option type
character vector with values: `'UI'`, `'UO'`, `'DI'`, `'DO'` | cell array of character vectors with values: `'UI'`, `'UO'`, `'DI'`, `'DO'`

Barrier option type, specified as a character vector or an `NINST`-by-`1` cell array of character vectors with the following values:

- `'UI'` — Up Knock-in

  This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option.
- `'UO'` — Up Knock-out

  This option gives the option holder the right, but not the obligation, to buy/sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the barrier level during the life of the option. This option terminates when the price of the underlying asset passes above the barrier level. Usually, with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.
- `'DI'` — Down Knock-in

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option. Note, `barrierbyfd` does not support American knock-in barrier options.
- `'DO'` — Down Knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: `char` | `cell`

`Barrier` — Barrier level
numeric

Barrier level, specified as a `NINST`-by-`1` matrix of numeric values.

Data Types: `double`

`Rebate` — Rebate value
`0` (default) | numeric

(Optional) Rebate value, specified as a `NINST`-by-`1` matrix of numeric values. For Knock-in options, the `Rebate` is paid at expiry. For Knock-out options, the `Rebate` is paid when the `Barrier` is reached.

Data Types: `double`

`Options` — Derivatives pricing options
structure

(Optional) Derivatives pricing options, specified as structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices for barrier options at time 0
matrix

Expected prices for barrier options at time 0, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure with vector of barrier option prices at each node
tree structure

Structure with a vector of barrier option prices at each node, returned as a tree structure.

`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.

More About

Barrier Option

A Barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by `Barrier`, during the life of the option. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `barrierbycrr` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] 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` | `Barrier`

Topics
“Computing Prices Using CRR” on page 3-65
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Calibrate Option Pricing Model Using Heston Model” on page 3-143
“Barrier Option” on page 3-20
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

barrierbyeqp

Price barrier option from Equal Probabilities binomial tree

Syntax

```
[Price,PriceTree] = barrierbyeqp(EQPTree,OptSpec,Strike,Settle,AmericanOpt,
ExerciseDates,BarrierSpec,Barrier)
[Price,PriceTree] = barrierbyeqp( ___ ,Rebate,Options)
```

Description

`[Price,PriceTree] = barrierbyeqp(EQPTree,OptSpec,Strike,Settle,AmericanOpt, ExerciseDates,BarrierSpec,Barrier)` calculates prices for barrier options using an Equal Probabilities binomial tree.

Note Alternatively, you can use the `Barrier` object to price Barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = barrierbyeqp( ___ ,Rebate,Options)` adds optional arguments for `Rebate` and `Options`.

Examples

Price a Barrier Option Using an EQP Equity Tree

This example shows how to price a barrier option using an EQP equity tree by loading 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;

OptSpec = 'Call';
Strike = 105;
Settle = datetime(2003,1,1);
ExerciseDates = datetime(2006,1,1);
AmericanOpt = 1;
BarrierSpec = 'UI';
Barrier = 102;

Price = barrierbyeqp(EQPTree, OptSpec, Strike, Settle, ...
ExerciseDates, AmericanOpt, BarrierSpec, Barrier)

Price = 12.2632
```

Input Arguments

`EQPTree` — Stock tree structure

structure

Stock tree structure, specified by using `eqptree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of an option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vector values.

Data Types: `char` | `cell`

`Strike` — Option strike price value

numeric

Option strike price value for a European or an American Option, specified as `NINST`-by-`1` matrix of numeric values. Each row is the schedule for one option.

Data Types: `double`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the barrier option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every barrier is set to the `ValuationDate` of the stock tree. The barrier argument `Settle` is ignored.

To support existing code, `barrierbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vector:

- For a European option, use a `1`-by-`1` matrix of dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date which is the maturity of the instrument.
- For an American option, use a `1`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-`1`, the option can be exercised between `ValuationDate` of the stock tree and the single listed date in `ExerciseDates`.

To support existing code, `barrierbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type

option type with values `0` or `1`

Option type, specified as `NINST`-by-`1` matrix of flags with values:

- `0` — European
- `1` — American

Data Types: `double`

`BarrierSpec` — Barrier option type
character vector with values: `'UI'`, `'UO'`, `'DI'`, `'DO'` | cell array of character vectors with values: `'UI'`, `'UO'`, `'DI'`, `'DO'`

Barrier option type, specified as a character vector or an `NINST`-by-`1` cell array of character vectors with the following values:

- `'UI'` — Up Knock-in

  This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option.
- `'UO'` — Up Knock-out

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the barrier level during the life of the option. This option terminates when the price of the underlying asset passes above the barrier level. Usually, with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.
- `'DI'` — Down Knock-in

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option. Note, `barrierbyfd` does not support American knock-in barrier options.
- `'DO'` — Down Knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: `char` | `cell`

`Barrier` — Barrier level
numeric

Barrier level, specified as a NINST-by-1 matrix of numeric values.

Data Types: double

Rebate — Rebate value
0 (default) | numeric

(Optional) Rebate value, specified as a NINST-by-1 matrix of numeric values. For Knock-in options, the Rebate is paid at expiry. For Knock-out options, the Rebate is paid when the Barrier is reached.

Data Types: double

Options — Derivatives pricing options
structure

(Optional) Derivatives pricing options, specified as a structure that is created with derivset.

Data Types: struct

Output Arguments

Price — Expected prices for barrier options at time 0
matrix

Expected prices for barrier options at time 0, returned as a NINST-by-1 matrix.

PriceTree — Structure with vector of barrier option prices at each node
tree structure

Structure with a vector of barrier option prices at each node, returned as a tree structure.

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.

More About

Barrier Option

A Barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by Barrier, during the life of the option. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `barrierbyeqp` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] 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` | `Barrier`

Topics

“Computing Prices Using CRR” on page 3-65
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Calibrate Option Pricing Model Using Heston Model” on page 3-143
“Barrier Option” on page 3-20
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

barrierbyfd

Calculate barrier option prices using finite difference method

Syntax

```
[Price,PriceGrid,AssetPrices,Times] = barrierbyfd(RateSpec,StockSpec,OptSpec,
Strike,Settle,ExerciseDates,BarrierSpec,Barrier)
[Price,PriceGrid,AssetPrices,Times] = barrierbyfd( ___ ,Name,Value)
```

Description

`[Price,PriceGrid,AssetPrices,Times] = barrierbyfd(RateSpec,StockSpec,OptSpec, Strike,Settle,ExerciseDates,BarrierSpec,Barrier)` calculates European and American barrier option prices on a single underlying asset using the finite difference method. `barrierbyfd` assumes that the barrier is continuously monitored.

Note Alternatively, you can use the `Barrier` object to price Barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceGrid,AssetPrices,Times] = barrierbyfd( ___ ,Name,Value)` adds optional name-value pair arguments. `barrierbyfd` assumes that the barrier is continuously monitored.

Examples

Price a Barrier Down and Out Call Option Using Finite Difference Method

Create a `RateSpec`.

```
AssetPrice = 50;
Strike = 45;
Rate = 0.035;
Volatility = 0.30;
Settle = datetime(2015,1,1);
Maturity = datetime(2016,1,1);
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9656
            Rates: 0.0350
         EndTimes: 1
       StartTimes: 0
         EndDates: 736330
```

```
       StartDates: 735965
    ValuationDate: 735965
            Basis: 1
     EndMonthRule: 1
```

Create a `StockSpec`.

```
StockSpec = stockspec(Volatility, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3000
         AssetPrice: 50
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Calculate the price of a European Down and Out call option using Finite Difference.

```
Barrier = 40;
BarrierSpec = 'DO';
OptSpec = 'Call';
Price = barrierbyfd(RateSpec, StockSpec, OptSpec, Strike, Settle, Maturity,...
BarrierSpec, Barrier)
```

```
Price = 8.5020
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

OptSpec — Definition of option

character vector with values `'call'` or `'put'` | string array with values `'call'` or `'put'`

Definition of an option as `'call'` or `'put'`, specified as a character vector or string array with values `"call"` or `"put"`.

Data Types: `char` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

`Settle` — Settlement or trade date
datetime scalar | string scalar | date character vector

Settlement or trade date for the barrier option, specified as a scalar datetime, string, or date character vector.

To support existing code, `barrierbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, there is only one `ExerciseDates` on the option expiry date which is the maturity of the instrument.
- For an American option, use a 1-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, the option can be exercised between `Settle` and the single listed date in `ExerciseDates`.

To support existing code, `barrierbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`BarrierSpec` — Barrier option type
character vector with values: `'UI'`, `'UO'`, `'DI'`, `'DO'`

Barrier option type, specified as a character vector with the following values:

- `'UI'` — Up Knock-in

  This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option. Note, `barrierbyfd` does not support American knock-in barrier options.
- `'UO'` — Up Knock-out

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the barrier level during the life of the option. This option terminates when the price of the underlying asset passes above the barrier level. Usually, with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.
- `'DI'` — Down Knock-in

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the

underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option. Note, `barrierbyfd` does not support American knock-in barrier options.

- `'DO'` — Down Knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: `char`

`Barrier` — Barrier level

numeric

Barrier level, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = barrierbyfd(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec,Barrier,Rebate,1000)`

`Rebate` — Rebate value

`0` (default) | numeric

Rebate value, specified as the comma-separated pair consisting of `'Rebate'` and a scalar numeric. For Knock-in options, the `Rebate` is paid at expiry. For Knock-out options, the `Rebate` is paid when the `Barrier` is reached.

Data Types: `double`

`AssetGridSize` — Size of asset grid used for a finite difference grid

`400` (default) | positive numeric

Size of the asset grid used for finite difference grid, specified as the comma-separated pair consisting of `'AssetGridSize'` and a scalar positive numeric.

Data Types: `double`

`TimeGridSize` — Size of time grid used for finite difference grid
`100` (default) | positive numeric

Size of the time grid used for the finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a scalar positive numeric.

Data Types: `double`

`AmericanOpt` — Option type
`0` European (default) | integer with values `0` or `1`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar flag with one of the following values:

- `0` — European
- `1` — American

Data Types: `logical`

Output Arguments

`Price` — Expected prices for barrier options
matrix

Expected prices for barrier options, returned as a `NINST`-by-`1` matrix.

`PriceGrid` — Grid containing prices calculated by finite difference method
grid

Grid containing prices calculated by the finite difference method, returned as a grid that is two-dimensional with size `PriceGridSize*length(Times)`. The number of columns does not have to be equal to the `TimeGridSize`, because ex-dividend dates in the `StockSpec` are added to the time grid. The price for `t = 0` is contained in `PriceGrid(:, end)`.

`AssetPrices` — Prices of asset defined by `StockSpec`
vector

Prices of the asset defined by the `StockSpec` corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Times` — Times corresponding to second dimension of `PriceGrid`
vector

Times corresponding to the second dimension of the `PriceGrid`, returned as a vector.

More About

Barrier Option

A Barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying

asset crosses the predetermined trigger value (barrier level), indicated by `Barrier`, during the life of the option. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced in R2016b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `barrierbyfd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. *Options, Futures, and Other Derivatives.* Fourth Edition. Prentice Hall. 2000, pp. 646–649.

[2] Aitsahlia, F., L. Imhof, and T.L. Lai. “Pricing and hedging of American knock-in options.” *The Journal of Derivatives.* Vol. 11.3, 2004, pp. 44–50.

[3] Rubinstein M. and E. Reiner. “Breaking down the barriers.” *Risk*. Vol. 4(8), 1991, pp. 28–35.

See Also

`barriersensbyfd` | `barrierbybls` | `barriersensbybls` | `barrierbyls` | `barriersensbyls` | `Barrier`

Topics

“Pricing European Call Options Using Different Equity Models” on page 3-88
“Calibrate Option Pricing Model Using Heston Model” on page 3-143
“Barrier Option” on page 3-20
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

barriersensbyfd

Calculate barrier option prices or sensitivities using finite difference method

Syntax

```
[PriceSens,PriceGrid,AssetPrices,Times] = barriersensbyfd(RateSpec,StockSpec,
OptSpec,Strike,Settle,ExerciseDates,BarrierSpec,Barrier)
[PriceSens,PriceGrid,AssetPrices,Times] = barriersensbyfd( ___ ,Name,Value)
```

Description

`[PriceSens,PriceGrid,AssetPrices,Times] = barriersensbyfd(RateSpec,StockSpec, OptSpec,Strike,Settle,ExerciseDates,BarrierSpec,Barrier)` calculates European and American barrier option prices or sensitivities of a single underlying asset using the finite difference method. `barrierbyfd` assumes that the barrier is continuously monitored.

Note Alternatively, you can use the `Barrier` object to calculate price or sensitivities for Barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,PriceGrid,AssetPrices,Times] = barriersensbyfd( ___ ,Name,Value)` adds optional name-value pair arguments. `barriersesbyfd` assumes that the barrier is continuously monitored.

Examples

Calculate Price and Sensitivities for a Barrier Down and Out Call Option Using Finite Difference Method

Create a `RateSpec`.

```
AssetPrice = 50;
Strike = 45;
Rate = 0.035;
Volatility = 0.30;
Settle = datetime(2015,1,1);
Maturity = datetime(2016,1,1);
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9656
            Rates: 0.0350
         EndTimes: 1
```

```
       StartTimes: 0
         EndDates: 736330
       StartDates: 735965
    ValuationDate: 735965
            Basis: 1
     EndMonthRule: 1
```

Create a `StockSpec`.

```
StockSpec = stockspec(Volatility,AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3000
         AssetPrice: 50
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Calculate the `Price`, `Delta`, and `Theta` of a European Down and Out call option using the finite difference method.

```
Barrier = 40;
BarrierSpec = 'DO';
OptSpec = 'Call';
OutSpec = {'price';'delta';'theta'};
[Price, Delta, Theta] = barriersensbyfd(RateSpec,StockSpec,OptSpec,Strike,Settle,...
Maturity, BarrierSpec,Barrier,'Outspec',OutSpec)
```

```
Price = 8.5020

Delta = 0.8569

Theta = -1.8502
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | string array with values `'call'` or `'put'`

Definition of an option as `'call'` or `'put'`, specified as a character vector or string array with values `"call"` or `"put"`.

Data Types: `char` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

`Settle` — Settlement or trade date
datetime scalar | string scalar | date character vector

Settlement or trade date for the barrier option, specified as a scalar datetime, string, or date character vector.

To support existing code, `barriersensbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, there is only one `ExerciseDates` on the option expiry date which is the maturity of the instrument.
- For an American option, use a `1`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, the option can be exercised between `Settle` and the single listed date in `ExerciseDates`.

To support existing code, `barriersensbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`BarrierSpec` — Barrier option type
character vector with values: `'UI'`, `'UO'`, `'DI'`, `'DO'`

Barrier option type, specified as a character vector with the following values:

- `'UI'` — Up Knock-in

  This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option. Note, `barrierbyfd` does not support American knock-in barrier options.
- `'UO'` — Up Knock-out

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the

barrier level during the life of the option. This option terminates when the price of the underlying asset passes above the barrier level. Usually, with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.

- `'DI'` — Down Knock-in

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option. Note, `barrierbyfd` does not support American knock-in barrier options.

- `'DO'` — Down Knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying asset at the strike price, as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: `char`

`Barrier` — Barrier level

numeric

Barrier level, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
```
PriceSens =
barriersensbyfd(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec
,Barrier,Rebate,1000,AmericanOpt,1)
```

`Rebate` — Rebate value

`0` (default) | numeric

Rebate value, specified as the comma-separated pair consisting of `'Rebate'` and a scalar numeric. For Knock-in options, the `Rebate` is paid at expiry. For `Knock-out` options, the `Rebate` is paid when the `Barrier` is reached.

Data Types: double

OutSpec — Define outputs
{'Price'} (default) | character vector with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All' | cell array of character vectors with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and a NOUT- by-1 or a 1-by-NOUT cell array of character vectors with possible values of 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'.

OutSpec = {'All'} specifies that the output is Delta, Gamma, Vega, Lambda, Rho, Theta, and Price, in that order. This is the same as specifying OutSpec to include each sensitivity.

Example: OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}

Data Types: char | cell

AssetGridSize — Size of asset grid used for finite difference grid
400 (default) | positive numeric

Size of the asset grid used for a finite difference grid, specified as the comma-separated pair consisting of 'AssetGridSize' and a scalar positive numeric.

Data Types: double

TimeGridSize — Size of time grid used for finite difference grid
100 (default) | positive numeric

Size of the time grid used for a finite difference grid, specified as the comma-separated pair consisting of 'TimeGridSize' and a scalar positive numeric.

Data Types: double

AmericanOpt — Option type
0 European (default) | integer with values 0 or 1

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and a scalar flag with one of the following values:

- 0 — European
- 1 — American

Data Types: logical

Output Arguments

PriceSens — Expected prices or sensitivities values for barrier options
matrix

Expected prices or sensitivities (defined using OutSpec) for barrier options, returned as a NINST-by-1 matrix.

PriceGrid — Grid containing prices calculated by finite difference method
grid

Grid containing prices calculated by the finite difference method, returned as a two-dimensional grid with size `PriceGridSize*length(Times)`. The number of columns does not have to be equal to the `TimeGridSize`, because ex-dividend dates in the `StockSpec` are added to the time grid. The price for `t = 0` is contained in `PriceGrid(:, end)`.

`AssetPrices` — Prices of the asset defined by `StockSpec`
vector

Prices of the asset defined by the `StockSpec` corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Times` — Times corresponding to second dimension of `PriceGrid`
vector

Times corresponding to the second dimension of the `PriceGrid`, returned as a vector.

More About

Barrier Option

A Barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by `Barrier`, during the life of the option. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced in R2016b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `barriersensbyfd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. *Options, Futures and Other Derivatives.* Fourth Edition. Prentice Hall, 2000, pp. 646–649.

[2] Aitsahlia, F., L. Imhof, and T.L. Lai. “Pricing and hedging of American knock-in options.” *The Journal of Derivatives.* Vol. 11.3 , 2004, pp. 44–50.

[3] Rubinstein M. and E. Reiner. “Breaking down the barriers.” *Risk*. Vol. 4(8), 1991, pp. 28–35.

See Also

`barrierbyfd` | `barrierbybls` | `barriersensbybls` | `barrierbyls` | `barriersensbyls` | `Barrier`

Topics

“Pricing European Call Options Using Different Equity Models” on page 3-88
“Calibrate Option Pricing Model Using Heston Model” on page 3-143
“Barrier Option” on page 3-20
“Supported Equity Derivative Functions” on page 3-19

dblbarrierbyfd

Calculate double barrier option price using finite difference method

Syntax

```
[Price,PriceGrid,AssetPrices,Times] = dblbarrierbyfd(RateSpec,StockSpec,
OptSpec,Strike,Settle,ExerciseDates,BarrierSpec,Barrier)
[Price,PriceGrid,AssetPrices,Times] = dblbarrierbyfd( ___ ,Name,Value)
```

Description

`[Price,PriceGrid,AssetPrices,Times] = dblbarrierbyfd(RateSpec,StockSpec, OptSpec,Strike,Settle,ExerciseDates,BarrierSpec,Barrier)` calculates a European or American call or put double barrier option price on a single underlying asset using the finite difference method. `dblbarrierbyfd` assumes that the barrier is continuously monitored.

Note Alternatively, you can use the `DoubleBarrier` object to price double barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceGrid,AssetPrices,Times] = dblbarrierbyfd( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Price an American Double Knock-Out Call Option with Rebate

Compute the price of an American double barrier option for a double knock-out (down and out-up and out) call option with a rebate using the following data:

```
Rate = 0.05;
Settle = datetime(2018,6,1);
Maturity = datetime(2018,12,1);
Basis = 1;
```

Define a `RateSpec`.

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates', Maturity,'Rates'
```

Define a `StockSpec`.

```
AssetPrice = 100;
Volatility = 0.25;
StockSpec = stockspec(Volatility, AssetPrice);
```

Define the double barrier option.

```
LBarrier = 80;
UBarrier = 130;
Barrier = [UBarrier LBarrier];
BarrierSpec = 'DKO';
OptSpec = 'Call';
Strike = 110;
Rebate = 1;
```

Compute the price of an American option using finite differences.

```
Price = dblbarrierbyfd(RateSpec, StockSpec, OptSpec, Strike, Settle, Maturity, BarrierSpec, Barr
```

```
Price = 4.0002
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset, specified by the `StockSpec` obtained from `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

OptSpec — Definition of option

character vector with values `'call'` or `'put'` | string scalar with values `"call"` or `"put"`

Definition of an option, specified as a character vector with a value of `'call'` or `'put'`, or a string scalar with values `"call"` or `"put"`.

Data Types: `char` | `string`

Strike — Option strike price value

scalar numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

Settle — Settlement or trade date

datetime scalar | string scalar | date character vector

Settlement or trade date for the barrier option, specified as a scalar datetime, string, or date character vector.

To support existing code, `dblbarrierbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors.

- For a European option, the option expiry date has only one `ExerciseDates` value.
- For an American option, use a 1-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates. If only one non-`NaN` date is listed, the option can be exercised between `Settle` and the single listed date in `ExerciseDates`.

To support existing code, `dblbarrierbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`BarrierSpec` — Double barrier option type
character vector with value of `'DKI'` or `'DKO'` | scalar string with value of `"DKI"` or `"DKO"`

Double barrier option type, specified as a character vector or string with one of the following values:

- `'DKI'` — Double Knock-in

  The `'DKI'` option becomes effective when the price of the underlying asset reaches one of the barriers. It gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, if the underlying asset goes above or below the barrier levels during the life of the option.
- `'DKO'` — Double Knock-out

  The `'DKO'` option gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, as long as the underlying asset remains between the barrier levels during the life of the option. This option terminates when the price of the underlying asset passes one of the barriers.

Option	Barrier Type	Payoff If Any Barrier Crossed	Payoff If Barriers Not Crossed
Call/Put	Double Knock-in	Standard Call/Put	Worthless
Call/Put	Double Knock-out	Worthless	Standard Call/Put

Data Types: `char` | `string`

`Barrier` — Barrier level
vector

Barrier level, specified as a 1-by-2 vector of numeric values, where the first column is the upper barrier (1)(UB) and the second column is the lower barrier (2)(LB). Barrier(1) must be greater than Barrier(2).

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = dblbarrierbyfd(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec, Barrier,'Rebate',[100,100])`

`Rebate` — Rebate value

`[0 0]` for Double Knock-out or `0` for Double Knock-in (default) | vector | scalar numeric

Rebate value, specified as the comma-separated pair consisting of `'Rebate'` and one of the following:

- For a Double Knock-out option, use a 1-by-2 vector of rebate values where the first column is the payout if the upper barrier(1)(UB) is hit and the second column is payout if the lower barrier(2) (LB) is hit. The rebate is paid when the barrier is reached.
- For a Double Knock-in option, use a scalar rebate value. The rebate is paid at expiry.

Data Types: `double`

`AssetGridSize` — Size of asset grid used for a finite difference grid

`400` (default) | positive scalar numeric

Size of the asset grid used for the finite difference grid, specified as the comma-separated pair consisting of `'AssetGridSize'` and a positive scalar numeric.

Data Types: `double`

`TimeGridSize` — Size of time grid used for finite difference grid

`100` (default) | positive scalar numeric

Size of the time grid used for the finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive scalar numeric.

Note The actual time grid may have a larger size because exercise and ex-dividend dates might be added to the grid from `StockSpec`.

Data Types: `double`

`AmericanOpt` — Option type

`0` European (default) | integer with values `0` or `1`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar flag with one of the following values:

- `0` — European
- `1` — American

Data Types: `logical`

Output Arguments

`Price` — Expected prices for double barrier options

matrix

Expected prices for double barrier options, returned as a 1-by-1 matrix.

`PriceGrid` — Grid containing prices

grid

Grid containing prices calculated by the finite difference method, returned as a two-dimensional grid with the size `AssetGridSize*TimeGridSize`. The number of columns does not have to be equal to the `TimeGridSize`, because exercise and ex-dividend dates in `StockSpec` are added to the time grid. `PriceGrid(:, end)` contains the price for `t = 0`.

`AssetPrices` — Prices of asset defined by `StockSpec`

vector

Prices of the asset defined by the `StockSpec` corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Times` — Times corresponding to second dimension of `PriceGrid`

vector

Times corresponding to the second dimension of `PriceGrid`, returned as a vector.

More About

Double Barrier Option

A double barrier option is similar to the standard single barrier option except that it has two barrier levels: a lower barrier (LB) and an upper barrier (UB).

The payoff for a double barrier option depends on whether the underlying asset remains between the barrier levels during the life of the option. Double barrier options are less expensive than single barrier options as they have a higher knock-out probability. Because of this, double barrier options allow investors to reduce option premiums and match an investor’s belief about the future movement of the underlying price process.

Version History

Introduced in R2019a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `dblbarrierbyfd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Boyle, P., and Y. Tian. “An Explicit Finite Difference Approach to the Pricing of Barrier Options.” *Applied Mathematical Finance.* Vol. 5, Number 1, 1998, pp. 17–43.

[2] Hull, J. *Options, Futures, and Other Derivatives.* Fourth Edition. Upper Saddle River, NJ: Prentice Hall, 2000, pp. 646–649.

[3] Rubinstein, M., and E. Reiner. “Breaking Down the Barriers.” *Risk.* Vol. 4, Number 8, 1991, pp. 28–35.

[4] Zvan, R., P. A. Forsyth and K. R. Vetzal. “PDE Methods for Pricing Barrier Options.” *Journal of Economic Dynamics and Control.* Vol. 24, Number 11-12, 2000, pp. 1563–1590.

See Also

`dblbarrierbybls` | `dblbarriersensbybls` | `dblbarriersensbyfd` | `DoubleBarrier`

Topics

“Double Barrier Option” on page 3-21
“Supported Equity Derivative Functions” on page 3-19

dblbarriersensbyfd

Calculate double barrier option price and sensitivities using finite difference method

Syntax

```
[PriceSens,PriceGrid,AssetPrices,Times] = dblbarriersensbyfd(RateSpec,
StockSpec,OptSpec,Strike,Settle,ExerciseDates,BarrierSpec,Barrier)
[PriceSens,PriceGrid,AssetPrices,Times] = dblbarriersensbyfd( ___ ,Name,Value)
```

Description

`[PriceSens,PriceGrid,AssetPrices,Times] = dblbarriersensbyfd(RateSpec, StockSpec,OptSpec,Strike,Settle,ExerciseDates,BarrierSpec,Barrier)` calculates a European or American call or put double barrier option price and sensitivities of a single underlying asset using the finite difference method. `dblbarrierbyfd` assumes that the barrier is continuously monitored.

Note Alternatively, you can use the `DoubleBarrier` object to calculate price or sensitivities for double barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,PriceGrid,AssetPrices,Times] = dblbarriersensbyfd( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Calculate Price and Sensitivities for an American Double Knock-Out Call Option with Rebate

Compute the price and sensitivities for an American double barrier option for a double knock-out (down and out-up and out) call option with a rebate using the following data:

```
Rate = 0.05;
Settle = datetime(2018,6,1);
Maturity = datetime(2018,12,1);
Basis = 1;
```

Define a `RateSpec`.

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates', Maturity,'Rates'
```

Define a `StockSpec`.

```
AssetPrice = 100;
Volatility = 0.25;
StockSpec = stockspec(Volatility, AssetPrice);
```

Define the double barrier option.

```
LBarrier = 80;
UBarrier = 130;
Barrier = [UBarrier LBarrier];
BarrierSpec = 'DKO';
OptSpec = 'Call';
Strike = 110;
Rebate = 1;
OutSpec = {'price'; 'vega'; 'theta'};
```

Compute the price and sensitivities for an American option using finite differences.

```
[Price, Vega, Theta] = dblbarriersensbyfd(RateSpec, StockSpec, OptSpec, Strike, Settle, Maturity
```

```
Price = 4.0002
```

```
Vega = -1.9180e+03
```

```
Theta = -6.6509
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset, specified by the `StockSpec` obtained from `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

OptSpec — Definition of option

character vector with values `'call'` or `'put'` | string scalar with values `"call"` or `"put"`

Definition of an option, specified as a character vector with a value of `'call'` or `'put'`, or a string scalar with values `"call"` or `"put"`.

Data Types: `char` | `string`

Strike — Option strike price value

scalar numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

Settle — Settlement or trade date

datetime scalar | string scalar | date character vector

Settlement or trade date for the barrier option, specified as a scalar datetime, string, or date character vector.

To support existing code, `dblbarriersensbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors.

- For a European option, the option expiry date has only one `ExerciseDates` value.
- For an American option, use a 1-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates. If only one non-`NaN` date is listed, the option can be exercised between `Settle` and the single listed date in `ExerciseDates`.

To support existing code, `dblbarriersensbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`BarrierSpec` — Double barrier option type

character vector with value of `'DKI'` or `'DKO'` | scalar string with value of `"DKI"` or `"DKO"`

Double barrier option type, specified as a character vector or string with one of the following values:

- `'DKI'` — Double Knock-in

  The `'DKI'` option becomes effective when the price of the underlying asset reaches one of the barriers. It gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, if the underlying asset goes above or below the barrier levels during the life of the option.
- `'DKO'` — Double Knock-out

  The `'DKO'` option gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, as long as the underlying asset remains between the barrier levels during the life of the option. This option terminates when the price of the underlying asset passes one of the barriers.

Option	Barrier Type	Payoff If Any Barrier Crossed	Payoff If Barriers Not Crossed
Call/Put	Double Knock-in	Standard Call/Put	Worthless
Call/Put	Double Knock-out	Worthless	Standard Call/Put

Data Types: `char` | `string`

`Barrier` — Barrier level

vector

Barrier level, specified as a 1-by-2 vector of numeric values, where the first column is the upper barrier (1)(UB) and the second column is the lower barrier (2)(LB). Barrier(1) must be greater than Barrier(2).

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = dblbarriersensbyfd(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec,Barrier,'OutSpec', {'delta','gamma','vega','lambda','rho','theta','price'},'AmericanOpt',1)`

`OutSpec` — Define outputs

`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | string array with values `"Price"`, `"Delta"`, `"Gamma"`, `"Vega"`, `"Lambda"`, `"Rho"`, `"Theta"`, and `"All"`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors or a string array with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell` | `string`

`Rebate` — Rebate value

`[0 0]` for Double Knock-out or `0` for Double Knock-in (default) | vector | scalar numeric

Rebate value, specified as the comma-separated pair consisting of `'Rebate'` and one of the following:

- For a Double Knock-out option, use a `1`-by-`2` vector of rebate values where the first column is the payout if the upper barrier(1)(UB) is hit and the second column is payout if the lower barrier(2)(LB) is hit. The rebate is paid when the barrier is reached.
- For a Double Knock-in option, use a scalar rebate value. The rebate is paid at expiry.

Data Types: `double`

`AssetGridSize` — Size of asset grid used for finite difference grid

`400` (default) | positive scalar numeric

Size of the asset grid used for the finite difference grid, specified as the comma-separated pair consisting of `'AssetGridSize'` and a positive scalar numeric.

Data Types: `double`

`TimeGridSize` — Size of time grid used for finite difference grid

`100` (default) | positive scalar numeric

Size of the time grid used for the finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive scalar numeric.

Note The actual time grid may have a larger size because exercise and ex-dividend dates might be added to the grid from `StockSpec`.

Data Types: `double`

`AmericanOpt` — Option type
`0` European (default) | integer with values `0` or `1`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar flag with one of the following values:

- `0` — European
- `1` — American

Data Types: `logical`

Output Arguments

`PriceSens` — Expected prices or sensitivities values for double barrier options
matrix

Expected prices or sensitivities (defined using `OutSpec`) for double barrier options, returned as a `1`-by-`NOUT` matrix.

`PriceGrid` — Grid containing prices
grid

Grid containing prices calculated by the finite difference method, returned as a two-dimensional grid with the size `AssetGridSize*TimeGridSize`. The number of columns does not have to be equal to the `TimeGridSize`, because exercise and ex-dividend dates in `StockSpec` are added to the time grid. `PriceGrid(:, end)` contains the price for `t = 0`.

`AssetPrices` — Prices of the asset defined by `StockSpec`
vector

Prices of the asset defined by the `StockSpec` corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Times` — Times corresponding to second dimension of `PriceGrid`
vector

Times corresponding to the second dimension of the `PriceGrid`, returned as a vector.

More About

Double Barrier Option

A double barrier option is similar to the standard single barrier option except that it has two barrier levels: a lower barrier (`LB`) and an upper barrier (`UB`).

The payoff for a double barrier option depends on whether the underlying asset remains between the barrier levels during the life of the option. Double barrier options are less expensive than single barrier options as they have a higher knock-out probability. Because of this, double barrier options

allow investors to reduce option premiums and match an investor's belief about the future movement of the underlying price process.

Version History

Introduced in R2019a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `dblbarriersensbyfd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Boyle, P., and Y. Tian. "An Explicit Finite Difference Approach to the Pricing of Barrier Options." *Applied Mathematical Finance*. Vol. 5, Number 1, 1998, pp. 17–43.

[2] Hull, J. *Options, Futures, and Other Derivatives.* Fourth Edition. Upper Saddle River, NJ: Prentice Hall, 2000, pp. 646–649.

[3] Rubinstein, M., and E. Reiner. "Breaking Down the Barriers." *Risk*. Vol. 4, Number 8, 1991, pp. 28–35.

[4] Zvan, R., P. A. Forsyth and K. R. Vetzal. "PDE Methods for Pricing Barrier Options." *Journal of Economic Dynamics and Control*. Vol. 24, Number 11-12, 2000, pp. 1563–1590.

See Also

`dblbarrierbybls` | `dblbarriersensbybls` | `dblbarrierbyfd` | `DoubleBarrier`

Topics

"Double Barrier Option" on page 3-21
"Supported Equity Derivative Functions" on page 3-19

barrierbyls

Price European or American barrier options using Monte Carlo simulations

Syntax

```
[Price,Paths,Times,Z] = barrierbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates,BarrierSpec,Barrier)
[Price,Paths,Times,Z] = barrierbyls( ___ ,Name,Value)
```

Description

`[Price,Paths,Times,Z] = barrierbyls(RateSpec,StockSpec,OptSpec,Strike,Settle, ExerciseDates,BarrierSpec,Barrier)` calculates barrier option prices on a single underlying asset using the Longstaff-Schwartz model. `barrierbyls` computes prices of European and American barrier options.

For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium.

Note Alternatively, you can use the `Barrier` object to price Barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,Paths,Times,Z] = barrierbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price an American Barrier Down In Put Option

Compute the price of an American down in put option using the following data:

```
Rates = 0.0325;
Settle = datetime(2016,1,1);
Maturity = datetime(2017,1,1);
Compounding = -1;
Basis = 1;
```

Define a `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9680
            Rates: 0.0325
```

```
          EndTimes: 1
        StartTimes: 0
          EndDates: 736696
        StartDates: 736330
     ValuationDate: 736330
             Basis: 1
      EndMonthRule: 1
```

Define a `StockSpec`.

```
AssetPrice = 40;
Volatility = 0.20;
StockSpec = stockspec(Volatility,AssetPrice)
```

```
StockSpec = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.2000
          AssetPrice: 40
        DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Calculate the price of an American barrier down in put option.

```
Strike = 45;
OptSpec = 'put';
Barrier = 35;
BarrierSpec = 'DI';
AmericanOpt = 1;

Price = barrierbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec,...
Barrier,'NumTrials',2000,'AmericanOpt',AmericanOpt)
```

```
Price = 4.7306
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option as `'call'` or `'put'`, specified as a character vector or string array with a value of `"call"` or `"put"`.

Data Types: `char` | `string`

`Strike` — Option strike price value
scalar numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

`Settle` — Settlement or trade date
datetime scalar | string scalar | date character vector

Settlement or trade date for the barrier option, specified as a scalar datetime, string, or date character vector.

To support existing code, `barrierbyls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, there is only one `ExerciseDates` on the option expiry date which is the maturity of the instrument.
- For an American option, use a `1`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, the option can be exercised between `Settle` and the single listed date in `ExerciseDates`.

To support existing code, `barrierbyls` also accepts serial date numbers as inputs, but they are not recommended.

`BarrierSpec` — Barrier option type
character vector with values: `'UI'`, `'UO'`, `'DI'`, `'DO'`

Barrier option type, specified as a character vector with the following values:

- `'UI'` — Up Knock-in

  This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option.
- `'UO'` — Up Knock-out

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the

barrier level during the life of the option. This option terminates when the price of the underlying asset passes above the barrier level. Usually with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.

- `'DI'` — Down Knock-in

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option.

- `'DO'` — Down Knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually, the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: `char`

`Barrier` — Barrier level

scalar numeric

Barrier level, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = barrierbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec,Barrier,Rebate,1000)`

`AmericanOpt` — Option type

`0` (European) (default) | values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

Note For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `double`

`Rebate` — Rebate value
`0` (default) | scalar numeric

Rebate value, specified as the comma-separated pair consisting of `'Rebate'` and a scalar numeric. For Knock-in options, the `Rebate` is paid at expiry. For Knock-out options, the `Rebate` is paid when the `Barrier` is reached.

Data Types: `double`

`NumTrials` — Number of independent sample paths
`1000` (default) | nonnegative integer

Number of independent sample paths (simulation trials), specified as the comma-separated pair consisting of `'NumTrials'` and a scalar nonnegative integer.

Data Types: `double`

`NumPeriods` — Number of simulation periods per trial
`100` (default) | nonnegative integer

Number of simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar nonnegative integer.

Data Types: `double`

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-`1`-by-`NumTrials` 3-D time series array. The `Z` value generates the Brownian motion vector (that is, Wiener processes) that drives the simulation.

Data Types: `double`

`Antithetic` — Indicator for antithetic sampling
`false` (default) | scalar logical flag with value of `true` or `false`

Indicator for antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a value of `true` or `false`.

Data Types: `logical`

Output Arguments

Price — Expected prices for barrier options

matrix

Expected prices for barrier options, returned as a NINST-by-1 matrix.

Paths — Simulated paths of correlated state variables

vector

Simulated paths of correlated state variables, returned as a NumPeriods + 1-by-1-by-NumTrials 3-D time series array of simulated paths of correlated state variables. Each row of Paths is the transpose of the state vector $X(t)$ at time t for a given trial.

Times — Observation times associated with simulated paths

vector

Observation times associated with simulated paths, returned as a NumPeriods + 1-by-1 column vector of observation times associated with the simulated paths. Each element of Times is associated with the corresponding row of Paths.

Z — Time series array of dependent random variates

vector

Time series array of dependent random variates, returned as a NumPeriods-by-1-by-NumTrials 3-D array when Z is specified as an input argument. If the Z input argument is not specified, then the Z output argument contains the random variates generated internally.

More About

Barrier Option

A Barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by Barrier, during the life of the option. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced in R2016b

Serial date numbers not recommended

Not recommended starting in R2022b

Although barrierbyls supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. *Options, Futures and Other Derivatives.* Fourth Edition. Prentice Hall, 2000, pp. 646–649.

[2] Aitsahlia, F., L. Imhof, and T.L. Lai. “Pricing and hedging of American knock-in options.” *The Journal of Derivatives.* Vol. 11.3 , 2004, pp. 44–50.

[3] Rubinstein M. and E. Reiner. “Breaking down the barriers.” *Risk*. Vol. 4(8), 1991, pp. 28–35.

See Also

`barrierbyfd` | `barriersensbyfd` | `barrierbybls` | `barriersensbybls` | `barriersensbyls` | `Barrier`

Topics

“Pricing European Call Options Using Different Equity Models” on page 3-88
“Calibrate Option Pricing Model Using Heston Model” on page 3-143
“Barrier Option” on page 3-20
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

barriersensbyls

Calculate price and sensitivities for European or American barrier options using Monte Carlo simulations

Syntax

```
[PriceSens,Paths,Times,Z] = barriersensbyls(RateSpec,StockSpec,OptSpec,
Strike,Settle,ExerciseDates,BarrierSpec,Barrier)
[PriceSens,Paths,Times,Z] = barriersensbyls( ___ ,Name,Value)
```

Description

`[PriceSens,Paths,Times,Z] = barriersensbyls(RateSpec,StockSpec,OptSpec, Strike,Settle,ExerciseDates,BarrierSpec,Barrier)` calculates barrier option prices or sensitivities on a single underlying asset using the Longstaff-Schwartz model. `barriersensbyls` computes prices of European and American barrier options.

For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium.

Note Alternatively, you can use the `Barrier` object to calculate price or sensitivities for Barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,Paths,Times,Z] = barriersensbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Delta and Gamma of an American Barrier Down In Put Option

Compute the price of an American down in put option using the following data:

```
Rates = 0.0325;
Settle = datetime(2016,1,1);
Maturity = datetime(2017,1,1);
Compounding = -1;
Basis = 1;
```

Define a `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9680
```

```
           Rates: 0.0325
         EndTimes: 1
       StartTimes: 0
         EndDates: 736696
       StartDates: 736330
    ValuationDate: 736330
           Basis: 1
     EndMonthRule: 1
```

Define a `StockSpec`.

```
AssetPrice = 40;
Volatility = 0.20;
StockSpec = stockspec(Volatility,AssetPrice)
```

```
StockSpec = struct with fields:
            FinObj: 'StockSpec'
             Sigma: 0.2000
         AssetPrice: 40
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Calculate the delta and gamma of an American barrier down in put option.

```
Strike = 45;
OptSpec = 'put';
Barrier = 35;
BarrierSpec = 'DI';
AmericanOpt = 1;

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

[Delta,Gamma] = barriersensbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,...
Maturity,BarrierSpec,Barrier,'NumTrials',2000,'AmericanOpt',AmericanOpt,'OutSpec',OutSpec)
```

```
Delta = -0.6346

Gamma = -0.3091
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option as `'call'` or `'put'`, specified as a character vector or string array with values `"call"` or `"put"`.

Data Types: `char` | `string`

`Strike` — Option strike price value
scalar numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

`Settle` — Settlement or trade date
datetime scalar | string scalar | date character vector

Settlement or trade date for the barrier option, specified as a scalar datetime, string, or date character vector.

To support existing code, `barriersensbyls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, there is only one `ExerciseDates` on the option expiry date which is the maturity of the instrument.
- For an American option, use a 1-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, the option can be exercised between `Settle` and the single listed date in `ExerciseDates`.

To support existing code, `barriersensbyls` also accepts serial date numbers as inputs, but they are not recommended.

`BarrierSpec` — Barrier option type
character vector with values: `'UI'`, `'UO'`, `'DI'`, `'DO'`

Barrier option type, specified as a character vector with the following values:

- `'UI'` — Up Knock-in

  This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option.

- `'UO'` — Up Knock-out

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the barrier level during the life of the option. This option terminates when the price of the underlying asset passes above the barrier level. Usually with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.

- `'DI'` — Down Knock-in

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option.

- `'DO'` — Down Knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually, the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: `char`

`Barrier` — Barrier level

scalar numeric

Barrier level, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = barriersensbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec,Barrier,Rebate,1000)`

`AmericanOpt` — Option type

`0` (European) (default) | values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar flag with one of the following values:

- `0` — European
- `1` — American

Note For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `double`

`Rebate` — Rebate value
`0` (default) | numeric

Rebate value, specified as the comma-separated pair consisting of `'Rebate'` and a scalar numeric. For Knock-in options, the `Rebate` is paid at expiry. For Knock-out options, the `Rebate` is paid when the `Barrier` is reached.

Data Types: `double`

`NumTrials` — Number of independent sample paths
`1000` (default) | nonnegative integer

Number of independent sample paths (simulation trials), specified as the comma-separated pair consisting of `'NumTrials'` and a scalar nonnegative integer.

Data Types: `double`

`NumPeriods` — Number of simulation periods per trial
`100` (default) | nonnegative integer

Number of simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar nonnegative integer.

Data Types: `double`

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-`1`-by-`NumTrials` 3-D time series array. The `Z` value generates the Brownian motion vector (that is, Wiener processes) that drives the simulation.

Data Types: `double`

`Antithetic` — Indicator for antithetic sampling
`false` (default) | logical flag with value of `true` or `false`

Indicator for antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a scalar value of `true` or `false`.

Data Types: `logical`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

`MonitoringFreq` — Number of days between monitoring barriers
`0` (default) | integer

Number of days between monitoring barriers, specified as a scalar integer. The default is `0` which indicates that the barrier is continuously monitored.

Data Types: `double`

Output Arguments

`PriceSens` — Expected prices or sensitivities for barrier options
matrix

Expected prices or sensitivities (defined using `OutSpec`) for barrier options, returned as a `NINST`-by-`1` matrix.

`Paths` — Simulated paths of correlated state variables
vector

Simulated paths of correlated state variables, returned as a `NumPeriods + 1`-by-`1`-by-`NumTrials` 3-D time series array of simulated paths of correlated state variables. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths
vector

Observation times associated with simulated paths, returned as a `NumPeriods + 1`-by-`1` column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, returned as a `NumPeriods`-by-`1`-by-`NumTrials` 3-D array when `Z` is specified as an input argument. If the `Z` input argument is not specified, then the `Z` output argument contains the random variates generated internally.

More About

Barrier Option

A Barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by `Barrier`, during the life of the option. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced in R2016b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `barriersensbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. *Options, Futures and Other Derivatives* Fourth Edition. Prentice Hall, 2000, pp. 646-649.

[2] Aitsahlia, F., L. Imhof and T.L. Lai. “Pricing and hedging of American knock-in options.” *The Journal of Derivatives.* Vol. 11.3, 2004, pp. 44–50.

[3] Broadie, M., P. Glasserman and S. Kou. "A continuity correction for discrete barrier options." *Mathematical Finance.* Vol. 7.4 , 1997, pp. 3250–349.

[4] Moon, K.S. "Efficient Monte Carlo algorithm for pricing barrier options." *Communications of the Korean Mathematical Society.* Vol 23.2, 2008 pp. 85–294.

[5] Papatheodorou, B. *“Enhanced Monte Carlo methods for pricing and hedging exotic options.*" University of Oxford thesis, 2005.

[6] Rubinstein M. and E. Reiner. “Breaking down the barriers.” *Risk*. Vol. 4(8), 1991, pp. 28–35.

See Also

`barrierbyfd` | `barriersensbyfd` | `barrierbybls` | `barrierbybls` | `barriersensbyls` | `Barrier`

Topics

“Pricing European Call Options Using Different Equity Models” on page 3-88
“Calibrate Option Pricing Model Using Heston Model” on page 3-143
“Barrier Option” on page 3-20
“Supported Equity Derivative Functions” on page 3-19

barrierbybls

Price European barrier options using Black-Scholes option pricing model

Syntax

```
Price = barrierbybls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,
BarrierSpec,Barrier)
Price = barrierbybls( ___ ,Name,Value)
```

Description

`Price = barrierbybls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates, BarrierSpec,Barrier)` calculates European barrier option prices using the Black-Scholes option pricing model.

Note Alternatively, you can use the `Barrier` object to price Barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = barrierbybls( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price an European Barrier Down Out Call Option

Compute the price of an European barrier down out call option using the following data:

```
Rates = 0.035;
Settle = datetime(2015,1,1);
Maturity = datetime(2016,1,1);
Compounding = -1;
Basis = 1;
```

Define a `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9656
            Rates: 0.0350
         EndTimes: 1
       StartTimes: 0
         EndDates: 736330
       StartDates: 735965
    ValuationDate: 735965
            Basis: 1
```

```
     EndMonthRule: 1
```

Define a StockSpec.

```
AssetPrice = 50;
Volatility = 0.30;
StockSpec = stockspec(Volatility, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3000
         AssetPrice: 50
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Calculate the price of an European barrier down out call option using the Black-Scholes option pricing model.

```
Strike = 50;
OptSpec = 'call';
Barrier = 45;
BarrierSpec = 'DO';

Price = barrierbybls(RateSpec, StockSpec, OptSpec, Strike, Settle,...
Maturity,  BarrierSpec, Barrier)
```

```
Price = 4.4285
```

Price European Barrier Down Out and Down In Call Options

Compute the price of European down out and down in call options using the following data:

```
Rates = 0.035;
Settle = datetime(2015,1,1);
Maturity = datetime(2016,1,1);
Compounding = -1;
Basis = 1;
```

Define a RateSpec.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9656
            Rates: 0.0350
         EndTimes: 1
       StartTimes: 0
         EndDates: 736330
       StartDates: 735965
    ValuationDate: 735965
```

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

Define a `StockSpec`.

```
AssetPrice = 50;
Volatility = 0.30;
StockSpec = stockspec(Volatility, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3000
         AssetPrice: 50
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Calculate the price of European barrier down out and down in call options using the Black-Scholes Option Pricing model.

```
Strike = 50;
OptSpec = 'Call';
Barrier = 45;
BarrierSpec = {'DO';'DI'};

Price = barrierbybls(RateSpec, StockSpec, OptSpec, Strike, Settle, Maturity,  BarrierSpec, Barrie
```

```
Price = 2×1

    4.4285
    2.3301
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option as `'call'` or `'put'`, specified as an `NINST`-by-`1` cell array of character vectors or string array with values `'call'` or `'put'` or `"call"` or `"put"`.

Data Types: `char` | `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as an `NINST`-by-`1` matrix of numeric values, where each row is the schedule for one option.

Data Types: `double`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the barrier option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `barrierbybls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one `ExerciseDates` on the option expiry date which is the maturity of the instrument.

To support existing code, `barrierbybls` also accepts serial date numbers as inputs, but they are not recommended.

`BarrierSpec` — Barrier option type
character vector with values: `'UI'`, `'UO'`, `'DI'`, `'DO'`

Barrier option type, specified as an `NINST`-by-`1` cell array of character vectors with the following values:

- `'UI'` — Up Knock-in

  This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option.
- `'UO'` — Up Knock-out

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the barrier level during the life of the option. This option terminates when the price of the underlying

asset passes above the barrier level. Usually with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.

- `'DI'` — Down Knock-in

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option.

- `'DO'` — Down Knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually, the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: `char` | `cell`

`Barrier` — Barrier level

numeric

Barrier level, specified as `NINST`-by-`1` matrix of numeric values.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = barrierbybls(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec,Barrier,Rebate,1000)`

`Rebate` — Rebate value

`0` (default) | numeric

Rebate value, specified as the comma-separated pair consisting of `'Rebate'` and `NINST`-by-`1` matrix of numeric values. For Knock-in options, the `Rebate` is paid at expiry. For Knock-out options, the `Rebate` is paid when the `Barrier` is reached.

Data Types: `double`

Output Arguments

`Price` — Expected prices for barrier options

matrix

Expected prices for barrier options at time 0, returned as a `NINST`-by-`1` matrix.

More About

Barrier Option

A Barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by `Barrier`, during the life of the option. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced in R2016b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `barrierbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. *Options, Futures and Other Derivatives* Fourth Edition. Prentice Hall, 2000, pp. 646–649.

[2] Aitsahlia, F., L. Imhof, and T.L. Lai. “Pricing and hedging of American knock-in options.” *The Journal of Derivatives.* Vol. 11.3, 2004, pp. 44–50.

[3] Rubinstein M. and E. Reiner. “Breaking down the barriers.” *Risk*. Vol. 4(8), 1991, pp. 28–35.

See Also

`barrierbyfd` | `barriersensbyfd` | `barrierbyls` | `barriersensbybls` | `barriersensbyls` | `Barrier`

Topics

“Pricing European Call Options Using Different Equity Models” on page 3-88
“Calibrate Option Pricing Model Using Heston Model” on page 3-143
“Barrier Option” on page 3-20
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

barriersensbybls

Calculate price or sensitivities for European barrier options using Black-Scholes option pricing model

Syntax

```
PriceSens = barriersensbybls(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates,BarrierSpec,Barrier)
PriceSens = barriersensbybls( ___ ,Name,Value)
```

Description

`PriceSens = barriersensbybls(RateSpec,StockSpec,OptSpec,Strike,Settle, ExerciseDates,BarrierSpec,Barrier)` calculates European barrier option prices or sensitivities using the Black-Scholes option pricing model.

Note Alternatively, you can use the `Barrier` object to calculate price or sensitivities for Barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = barriersensbybls( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate Price and Sensitivities for European Barrier Down Out and Down In Call Options

Compute price of European barrier down out and down in call options using the following data:

```
Rates = 0.035;
Settle = '01-Jan-2015';
Maturity = '01-April-2015';
Compounding = -1;
Basis = 1;
```

Define a `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9913
            Rates: 0.0350
         EndTimes: 0.2500
       StartTimes: 0
         EndDates: 736055
       StartDates: 735965
    ValuationDate: 735965
            Basis: 1
```

```
     EndMonthRule: 1
```

Define a `StockSpec`.

```
AssetPrice = 19;
Volatility = 0.40;
DivType = 'Continuous';
DivAmount = 0.035;
StockSpec = stockspec(Volatility, AssetPrice, DivType, DivAmount)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.4000
         AssetPrice: 19
       DividendType: {'continuous'}
    DividendAmounts: 0.0350
    ExDividendDates: []
```

Calculate the `price`, `delta`, and `gamma` for European barrier down out and down in call options using the Black-Scholes option pricing model.

```
OptSpec = 'Call';
Strike = 20;
Barrier = 18;
BarrierSpec = {'DO';'DI'};
OutSpec = {'price', 'delta', 'gamma'};

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

```
Price = 2×1

    0.6287
    0.4655


Delta = 2×1

    0.6376
   -0.2036


Gamma = 2×1

    0.0255
    0.0773
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset
structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option as `'call'` or `'put'`, specified as an `NINST`-by-`1` cell array of character vectors or string arrays with values `"call"` or `"put"`.

Data Types: `char` | `string` | `cell`

`Strike` — Option strike price value
numeric

Option strike price value, specified as an `NINST`-by-`1` matrix of numeric values.

Data Types: `double`

`Settle` — Settlement or trade date
datetime array | string array | date character vector | serial date number

Settlement or trade date for the barrier option, specified as an `NINST`-by-`1` vector using a datetime array, string array, date character vectors, or serial date numbers.

Data Types: `double` | `char` | `string` | `datetime`

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector | serial date number

Option exercise dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, date character vectors, or serial date numbers.

Note For a European option, there is only one `ExerciseDates` on the option expiry date which is the maturity of the instrument.

Data Types: `double` | `char` | `string` | `datetime`

`BarrierSpec` — Barrier option type
character vector with values: `'UI'`, `'UO'`, `'DI'`, `'DO'`

Barrier option type, specified as an `NINST`-by-`1` cell array of character vectors with the following values:

- `'UI'` — Up Knock-in

This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option.

- `'UO'` — Up Knock-out

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the barrier level during the life of the option. This option terminates when the price of the underlying asset passes above the barrier level. Usually with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.

- `'DI'` — Down Knock-in

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option.

- `'DO'` — Down Knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually, the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: `char` | `cell`

`Barrier` — Barrier level

numeric

Barrier level, specified as an `NINST`-by-`1` matrix of numeric values.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = barriersensbybls(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec,Barrier,'Rebate',1000,'OutSpec','Delta')`

Rebate — Rebate values
0 (default) | numeric

Rebate values, specified as the comma-separated pair consisting of 'Rebate' and NINST-by-1 matrix of numeric values. For Knock-in options, the Rebate is paid at expiry. For Knock-out options, the Rebate is paid when the Barrier is reached.

Data Types: double

OutSpec — Define outputs
{'Price'} (default) | character vector with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All' | cell array of character vectors with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and a NOUT- by-1 or a 1-by-NOUT cell array of character vectors with possible values of 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'.

OutSpec = {'All'} specifies that the output is Delta, Gamma, Vega, Lambda, Rho, Theta, and Price, in that order. This is the same as specifying OutSpec to include each sensitivity.

Example: OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}

Data Types: char | cell

Output Arguments

PriceSens — Expected prices or sensitivities for barrier options
matrix

Expected prices at time 0 or sensitivities (defined using OutSpec) for barrier options, returned as a NINST-by-1 matrix.

More About

Barrier Option

A Barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by Barrier, during the life of the option. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced in R2016b

References

[1] Hull, J. *Options, Futures and Other Derivatives* Fourth Edition. Prentice Hall, 2000, pp. 646–649.

[2] Aitsahlia, F., L. Imhof, and T.L. Lai. “Pricing and hedging of American knock-in options.” *The Journal of Derivatives.* Vol. 11.3, 2004, pp. 44–50.

[3] Rubinstein M. and E. Reiner. “Breaking down the barriers.” *Risk*. Vol. 4(8), 1991, pp. 28–35.

See Also

`barrierbyfd` | `barriersensbyfd` | `barrierbyls` | `barrierbybls` | `barriersensbyls` | `Barrier`

Topics

“Pricing European Call Options Using Different Equity Models” on page 3-88
“Calibrate Option Pricing Model Using Heston Model” on page 3-143
“Barrier Option” on page 3-20
“Supported Equity Derivative Functions” on page 3-19

dblbarrierbybls

Price European double barrier options using Black-Scholes option pricing model

Syntax

```
Price = dblbarrierbybls(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates,BarrierSpec,Barrier)
Price = dblbarrierbybls( ___ ,Name,Value)
```

Description

`Price = dblbarrierbybls(RateSpec,StockSpec,OptSpec,Strike,Settle, ExerciseDates,BarrierSpec,Barrier)` calculates European double barrier option prices using the Black-Scholes option pricing model and the Ikeda and Kunitomo approximation.

Note Alternatively, you can use the `DoubleBarrier` object to price double barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = dblbarrierbybls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Price a European Double Knock-Out Call Option

Compute the price of a European for a double knock-out (down and out-up and out) call option using the following data:

```
Rate = 0.05;
Settle = datetime(2018,6,1);
Maturity = datetime(2018,12,1);
Basis = 1;
```

Define a `RateSpec`.

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates', Maturity,'Rates'
```

Define a `StockSpec`.

```
AssetPrice = 100;
Volatility = 0.25;
StockSpec = stockspec(Volatility, AssetPrice);
```

Define the double barrier option.

```
LBarrier = 80;
UBarrier = 130;
Barrier = [UBarrier LBarrier];
```

```
BarrierSpec = 'DKO';
OptSpec = 'Call';
Strike = 110;
```

Compute price of option using flat boundaries.

```
PriceFlat = dblbarrierbybls(RateSpec, StockSpec, OptSpec, Strike, Settle, Maturity, BarrierSpec,
```

```
PriceFlat = 1.1073
```

Compute price of option using two curved boundaries.

```
Curvature = [0.05 -0.05];
PriceCurved = dblbarrierbybls(RateSpec, StockSpec, OptSpec, Strike, Settle, Maturity, BarrierSpec
```

```
PriceCurved = 1.4548
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset, specified by the `StockSpec` obtained from `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

OptSpec — Definition of option

cell array of character vectors with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors or string array with values `'call'` or `'put'` or `"call"` or `"put"`.

Data Types: `char` | `cell` | `string`

Strike — Option strike price value

matrix

Option strike price value, specified as an `NINST`-by-`1` matrix of numeric values, where each row is the schedule for one option.

Data Types: `double`

Settle — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the double barrier option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `dblbarrierbybls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note For a European option, the option expiry date has only one `ExerciseDates` value, which is the maturity of the instrument.

To support existing code, `dblbarrierbybls` also accepts serial date numbers as inputs, but they are not recommended.

`BarrierSpec` — Double barrier option type

cell array of character vectors with values of `'DKI'` or `'DKO'` | string array with values of `"DKI"` or `"DKO"`

Double barrier option type, specified as an `NINST`-by-`1` cell array of character vectors or string array with the following values:

- `'DKI'` — Double Knock-In

  The `'DKI'` option becomes effective when the price of the underlying asset reaches one of the barriers. It gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, if the underlying asset goes above or below the barrier levels during the life of the option.
- `'DKO'` — Double Knock-Out

  The `'DKO'` option gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, as long as the underlying asset remains between the barrier levels during the life of the option. This option terminates when the price of the underlying asset passes one of the barriers.

Option	Barrier Type	Payoff If Any Barrier Crossed	Payoff If Barriers Not Crossed
Call/Put	Double Knock-in	Standard Call/Put	Worthless
Call/Put	Double Knock-out	Worthless	Standard Call/Put

Data Types: `char` | `cell` | `string`

`Barrier` — Double barrier value

numeric

Double barrier value, specified as `NINST`-by-`1` matrix of numeric values, where each element is a `1`-by-`2` vector where the first column is Barrier(1)(UB) and the second column is Barrier(2)(LB). Barrier(1) must be greater than Barrier(2).

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = dblbarrierbybls(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec,Barrier,'Curvature',[1,5])`

`Curvature` — Curvature levels of the upper and lower barriers

`[]` (default) | matrix

Curvature levels of the upper and lower barriers, specified as the comma-separated pair consisting of `'Curvature'` and an `NINST`-by-`1` matrix, where each element is a `1`-by-`2` vector. The first column is the upper barrier curvature (*d1*) and the second column is the lower barrier curvature (*d2*).

- *d1* = *d2* = `0` corresponds to two flat boundaries.
- *d1* < 0 < *d2* corresponds to an exponentially growing lower boundary and an exponentially decaying upper boundary.
- *d1* > 0 > *d2* corresponds to a convex downward lower boundary and a convex upward upper boundary.

Data Types: `double`

Output Arguments

`Price` — Expected prices for double barrier options

matrix

Expected prices for double barrier options at time 0, returned as a `NINST`-by-`1` matrix.

More About

Double Barrier Option

A double barrier option is similar to the standard single barrier option except that it has two barrier levels: a lower barrier (`LB`) and an upper barrier (`UB`).

The payoff for a double barrier option depends on whether the underlying asset remains between the barrier levels during the life of the option. Double barrier options are less expensive than single barrier options as they have a higher knock-out probability. Because of this, double barrier options allow investors to reduce option premiums and match an investor’s belief about the future movement of the underlying price process.

Ikeda and Kunitomo Approximation

The analytical formulas of Ikeda and Kunitomo approach pricing as constrained by curved boundaries.

This approach has the advantage of covering barriers that are flat, have exponential growth or decay, or are concave. The Ikeda and Kunitomo model for pricing double barrier options focuses on calculating double knock-out barriers.

Version History

Introduced in R2019a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `dblbarrierbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. *Options, Futures, and Other Derivatives.* Fourth Edition. Upper Saddle River, NJ: Prentice Hall, 2000.

[2] Kunitomo, N., and M. Ikeda. “Pricing Options with Curved Boundaries.” *Mathematical Finance*. Vol. 2, Number 4, 1992.

[3] Rubinstein, M., and E. Reiner. “Breaking Down the Barriers.” *Risk*. Vol. 4, Number 8, 1991, pp. 28–35.

See Also

`dblbarriersensbybls` | `dblbarrierbyfd` | `dblbarriersensbyfd` | `DoubleBarrier`

Topics

“Double Barrier Option” on page 3-21
“Supported Equity Derivative Functions” on page 3-19

dblbarriersensbybls

Calculate prices and sensitivities for European double barrier options using Black-Scholes option pricing model

Syntax

```
PriceSens = dblbarriersensbybls(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates,BarrierSpec,Barrier)
PriceSens = dblbarriersensbybls( ___ ,Name,Value)
```

Description

`PriceSens = dblbarriersensbybls(RateSpec,StockSpec,OptSpec,Strike,Settle, ExerciseDates,BarrierSpec,Barrier)` calculates European double barrier option prices and sensitivities using the Black-Scholes option pricing model and the Ikeda and Kunitomo approximation.

Note Alternatively, you can use the `DoubleBarrier` object to calculate price or sensitivities for double barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = dblbarriersensbybls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Calculate Price and Sensitivities for a European Double Knock-Out Call Option

Compute the price and sensitivities for a European double knock-out (down and out-up and out) call option using the following data:

```
Rate = 0.05;
Settle = datetime(2018,6,1);
Maturity = datetime(2018,12,1);
Basis = 1;
```

Define a `RateSpec`.

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates', Maturity,'Rates'
```

Define a `StockSpec`.

```
AssetPrice = 100;
Volatility = 0.25;
StockSpec = stockspec(Volatility, AssetPrice);
```

Define the double barrier option and sensitivities.

```
LBarrier = 80;
UBarrier = 130;
```

```
Barrier = [UBarrier LBarrier];
BarrierSpec = 'DKO';
OptSpec = 'Call';
Strike = 110;
OutSpec = {'Price', 'Delta', 'Gamma'};
```

Compute the price of the option using flat boundaries.

```
[PriceFlat, Delta, Gamma] = dblbarriersensbybls(RateSpec, StockSpec, OptSpec, Strike, Settle, Ma
```

```
PriceFlat = 1.1073
```

```
Delta = 0.0411
```

```
Gamma = -0.0040
```

Compute the price of the option using two curved boundaries.

```
Curvature = [0.05 -0.05];
[PriceCurved, Delta, Gamma] = dblbarriersensbybls(RateSpec, StockSpec, OptSpec, Strike, Settle, M
```

```
PriceCurved = 1.4548
```

```
Delta = 0.0620
```

```
Gamma = -0.0045
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset, specified by the `StockSpec` obtained from `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

OptSpec — Definition of option

cell array of character vectors with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors or string array with values `'call'` or `'put'` or `"call"` or `"put"`.

Data Types: `char` | `cell` | `string`

Strike — Option strike price value
matrix

Option strike price value, specified as an NINST-by-1 matrix of numeric values, where each row is the schedule for one option.

Data Types: double

Settle — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the double barrier option, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, dblbarriersensbybls also accepts serial date numbers as inputs, but they are not recommended.

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

Note For a European option, the option expiry date has only one ExerciseDates value, which is the maturity of the instrument.

To support existing code, dblbarriersensbybls also accepts serial date numbers as inputs, but they are not recommended.

BarrierSpec — Double barrier option type
cell array of character vectors with values of 'DKI' or 'DKO' | string array with values of "DKI" or "DKO"

Double barrier option type, specified as an NINST-by-1 cell array of character vectors or string array with the following values:

- 'DKI' — Double Knock-in

  The 'DKI' option becomes effective when the price of the underlying asset reaches one of the barriers. It gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, if the underlying asset goes above or below the barrier levels during the life of the option.
- 'DKO' — Double Knock-out

  The 'DKO' option gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, as long as the underlying asset remains between the barrier levels during the life of the option. This option terminates when the price of the underlying asset passes one of the barriers.

Option	Barrier Type	Payoff If Any Barrier Crossed	Payoff If Barriers Not Crossed
Call/Put	Double Knock-in	Standard Call/Put	Worthless

Option	Barrier Type	Payoff If Any Barrier Crossed	Payoff If Barriers Not Crossed
Call/Put	Double Knock-out	Worthless	Standard Call/Put

Data Types: `char` | `cell` | `string`

`Barrier` — Double barrier value
matrix

Double barrier value, specified as `NINST`-by-`1` matrix of numeric values, where each element is a `1`-by-`2` vector where the first column is `Barrier`(1)(`UB`) and the second column is `Barrier`(2)(`LB`). `Barrier`(1) must be greater than `Barrier`(2).

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = dblbarriersensbybls(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec,Barrier,'Curvature',[1,5],'OutSpec','Delta')`

`Curvature` — Curvature levels of the upper and lower barriers
`[]` (default) | matrix

Curvature levels of the upper and lower barriers, specified as the comma-separated pair consisting of `'Curvature'` and an `NINST`-by-`1` matrix of numeric values, where each element is a `1`-by-`2` vector. The first column is the upper barrier curvature (*d1*) and the second column is the lower barrier curvature (*d2*).

- *d1* = *d2* = `0` corresponds to two flat boundaries.
- *d1* < 0 < *d2* corresponds to an exponentially growing lower boundary and an exponentially decaying upper boundary.
- *d1* > 0 > *d2* corresponds to a convex downward lower boundary and a convex upward upper boundary.

Data Types: `double`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | string array with values `"Price"`, `"Delta"`, `"Gamma"`, `"Vega"`, `"Lambda"`, `"Rho"`, `"Theta"`, and `"All"`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities for double barrier options

matrix

Expected prices at time 0 or sensitivities (defined using `OutSpec`) for double barrier options, returned as a `NINST`-by-`1` matrix.

More About

Double Barrier Option

A double barrier option is similar to the standard single barrier option except that it has two barrier levels: a lower barrier (`LB`) and an upper barrier (`UB`).

The payoff for a double barrier option depends on whether the underlying asset remains between the barrier levels during the life of the option. Double barrier options are less expensive than single barrier options as they have a higher knock-out probability. Because of this, double barrier options allow investors to reduce option premiums and match an investor's belief about the future movement of the underlying price process.

Ikeda and Kunitomo Approximation

The analytical formulas of Ikeda and Kunitomo approach pricing as constrained by curved boundaries.

This approach has the advantage of covering barriers that are flat, have exponential growth or decay, or are concave. The Ikeda and Kunitomo model for pricing double barrier options focuses on calculating double knock-out barriers.

Version History

Introduced in R2019a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `dblbarriersensbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. *Options, Futures, and Other Derivatives.* Fourth Edition. Upper Saddle River, NJ: Prentice Hall, 2000.

[2] Kunitomo, N., and M. Ikeda. “Pricing Options with Curved Boundaries.” *Mathematical Finance*. Vol. 2, Number 4, 1992.

[3] Rubinstein, M., and E. Reiner. “Breaking Down the Barriers.” *Risk*. Vol. 4, Number 8, 1991, pp. 28–35.

See Also

`dblbarrierbybls` | `dblbarrierbyfd` | `dblbarriersensbyfd` | `DoubleBarrier`

Topics

“Double Barrier Option” on page 3-21
“Supported Equity Derivative Functions” on page 3-19

barrierbyitt

Price barrier options using implied trinomial tree (ITT)

Syntax

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

Description

`[Price,PriceTree] = barrierbyitt(ITTTree,OptSpec,Strike,Settle,AmericanOpt, ExerciseDates,BarrierSpec,Barrier)` calculates prices for barrier options using implied trinomial tree (ITT).

`[Price,PriceTree] = barrierbyitt( ___ ,Rebate,Options)` adds optional arguments for `Rebate` and `Options`.

Examples

Price a Barrier Option Using an ITT Tree

This example shows how to price a barrier option using an ITT tree by loading the file deriv.mat, which provides ITTTree. The ITTTree structure contains the stock specification and time information needed to price the option.

```
load deriv.mat;

OptSpec = 'Call';
Strike =  85;
Settle = datetime(2006,1,1);
ExerciseDates = datetime(2008,12,31);
AmericanOpt = 1;
BarrierSpec = 'UI';
Barrier =  115;

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

```
Price = 2.4074
```

Input Arguments

`ITTTree` — Stock tree structure

structure

Stock tree structure, specified by using `itttree`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of an option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vector values.

Data Types: `char` | `cell`

`Strike` — Option strike price value
numeric

Option strike price value for a European or an American Option, specified as `NINST`-by-`1` matrix of numeric values. Each row is the schedule for one option.

Data Types: `double`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the barrier option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every barrier is set to the `ValuationDate` of the stock tree. The barrier argument `Settle` is ignored.

To support existing code, `barrierbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a `NINST`-by-`1` matrix of dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date which is the maturity of the instrument.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. For each instrument, the option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is an `NINST`-by-`1`, the option can be exercised between `ValuationDate` of the stock tree and the single listed date in `ExerciseDates`.

To support existing code, `barrierbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
option type values `0` or `1`

Option type, specified as `NINST`-by-`1` flags with values:

- `0` — European
- `1` — American

Data Types: `double`

BarrierSpec — Barrier option type
character vector with values: 'UI', 'UO', 'DI', 'DO' | cell array of character vectors with values: 'UI', 'UO', 'DI', 'DO'

Barrier option type, specified as a character vector or a cell array of character vectors with the following values:

- 'UI' — Up Knock-in

  This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option. Note, barrierbyfd does not support American knock-in barrier options.
- 'UO' — Up Knock-out

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the barrier level during the life of the option. This option terminates when the price of the underlying asset passes above the barrier level. Usually, with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.
- 'DI' — Down Knock-in

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option. Note, barrierbyfd does not support American knock-in barrier options.
- 'DO' — Down Knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: char | cell

Barrier — Barrier level
numeric

Barrier level, specified as an NINST-by-1 matrix of numeric values.

Data Types: double

`Rebate` — Rebate value
`0` (default) | numeric

(Optional) Rebate value, specified as a `NINST`-by-`1` matrix of numeric values. For Knock-in options, the `Rebate` is paid at expiry. For Knock-out options, the `Rebate` is paid when the `Barrier` is reached.

Data Types: `double`

`Options` — Derivatives pricing options
structure

(Optional) Derivatives pricing options, specified as structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices for barrier options at time 0
vector

Expected prices for barrier options at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Structure with vector of barrier option prices at each node
tree structure

Structure with a vector of barrier option prices at each node, returned as a tree structure.

`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.

More About

Barrier Option

A Barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by `Barrier`, during the life of the option. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced in R2007a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `barrierbyitt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] 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

`itttree` | `instbarrier`

Topics

“Computing Prices Using CRR” on page 3-65
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Barrier Option” on page 3-20
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

barrierbystt

Price barrier options using standard trinomial tree

Syntax

```
[Price,PriceTree] = barrierbystt(STTTree,OptSpec,Strike,Settle,ExerciseDates,
AmericanOpt,BarrierSpec,Barrier)
[Price,PriceTree] = barrierbystt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = barrierbystt(STTTree,OptSpec,Strike,Settle,ExerciseDates, AmericanOpt,BarrierSpec,Barrier)` prices barrier options using a standard trinomial (STT) tree.

Note Alternatively, you can use the `Barrier` object to price Barrier options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = barrierbystt( ___ ,Name,Value)` prices barrier options using a standard trinomial (STT) tree with an optional name-value pair argument for `Rebate` and `Options`.

Examples

Price a Barrier Option Using the Standard Trinomial Tree Model

Create a `RateSpec`.

```
StartDates = datetime(2009,1,1);
EndDates = datetime(2013,1,1);
Rates = 0.035;
Basis = 1;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8694
            Rates: 0.0350
         EndTimes: 4
       StartTimes: 0
         EndDates: 735235
       StartDates: 733774
    ValuationDate: 733774
            Basis: 1
     EndMonthRule: 1
```

Create a StockSpec.

```
AssetPrice = 85;
Sigma = 0.15;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1500
         AssetPrice: 85
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Create an STTTree.

```
NumPeriods = 4;
TimeSpec = stttimespec(StartDates, EndDates, 4);
STTTree = stttree(StockSpec, RateSpec, TimeSpec)
```

```
STTTree = struct with fields:
       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 1 2 3 4]
         dObs: [733774 734139 734504 734869 735235]
        STree: {1x5 cell}
        Probs: {[3x1 double]  [3x3 double]  [3x5 double]  [3x7 double]}
```

Define the barrier option and compute the price.

```
Settle = datetime(2009,1,1);
ExerciseDates = datetime(2012,1,1);
OptSpec =  'call';
Strike = 105;
AmericanOpt = 1;
BarrierSpec = 'UI';
Barrier = 115;

Price= barrierbystt(STTTree, OptSpec, Strike, Settle, ExerciseDates,...
AmericanOpt, BarrierSpec, Barrier)
```

```
Price = 3.7977
```

Input Arguments

STTTree — Stock tree structure for standard trinomial tree

structure

Stock tree structure for a standard trinomial tree, specified by using stttree.

Data Types: struct

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a character vector or a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — European or American option strike price value
nonnegative numeric

European or American option strike price value, specified with a nonnegative integer using a `NINST`-by-`1` matrix of nonnegative numeric values. Each row is the schedule for one option. To compute the value of a floating-strike barrier option, `Strike` should be specified as `NaN`. Floating-strike barrier options are also known as average strike options.

Data Types: `double`

`Settle` — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date for the barrier option, specified as a `NINST`-by-`1` vector of settlement or trade dates using a datetime array, string array, or date character vectors.

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

To support existing code, `barrierbystt` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a`NINST`-by-`1` matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. 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 a `NINST`-by-`1` cell array of character vectors, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `barrierbystt` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
scalar with values `[0,1]`

Option type, specified as an `NINST`-by-`1` matrix of flags with values:

- `0` — European
- `1` — American

Data Types: `double`

`BarrierSpec` — Barrier option type
character vector with values: `'UI'`, `'UO'`, `'DI'`, `'DO'` | cell array of character vectors with values: `'UI'`, `'UO'`, `'DI'`, `'DO'`

Barrier option type, specified as a character vector or an `NINST`-by-`1` cell array of character vectors with the following values:

- `'UI'` — Up Knock-in

  This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option. Note, `barrierbyfd` does not support American knock-in barrier options.
- `'UO'` — Up Knock-out

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the barrier level during the life of the option. This option terminates when the price of the underlying asset passes above the barrier level. Usually, with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.
- `'DI'` — Down Knock-in

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option. Note, `barrierbyfd` does not support American knock-in barrier options.
- `'DO'` — Down Knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call/put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: `char` | `cell`

`Barrier` — Barrier levels
numeric

Barrier levels, specified as an `NINST`-by-`1` matrix of numeric values.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = barrierbystt(STTTree,OptSpec,Strike,Settle,ExerciseDates,1,'UI',115,'Rebate',25)`

`Rebate` — Rebate values
`0` (default) | numeric

Rebate values, specified as the comma-separated pair consisting of `'Rebate'` and a `NINST`-by-`1` matrix of numeric values. For Knock-in options, the `Rebate` is paid at expiry. For Knock-out options, the `Rebate` is paid when the`Barrier` is reached.

Data Types: `double`

`Options` — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices for barrier options at time `0`
matrix

Expected prices for barrier options at time 0, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure with vector of barrier option prices at each node
tree structure

Structure with a vector of barrier option prices at each node, returned as a tree structure.

`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.

More About

Barrier Option

A Barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by `Barrier`, during the life of the option. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced in R2015b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `barrierbystt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] 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

`stttimespec` | `stttree` | `sttprice` | `sttsens` | `derivset` | `instbarrier` | `Barrier`

Topics

“Pricing European Call Options Using Different Equity Models” on page 3-88
“Calibrate Option Pricing Model Using Heston Model” on page 3-143
“Barrier Option” on page 3-20
“Supported Equity Derivative Functions” on page 3-19

basketbyju

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.

Examples

Price European Basket Options Using Nengjiu Ju Approximation Model

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 Nengjiu Ju approximation model.

```
Settle = datetime(2009,1,1);
Maturity  = datetime(2010,1,1);

% Define the 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 the 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)
```

```
PriceCorr30 = 2.1221
```

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)

PriceCorr60 = 2.2757
```

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)

PriceVanillaOption = 2×1

    1.0451
    1.4186
```

Find the total cost of buying two individual call options.

```
sum(PriceVanillaOption)

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%.

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

BasketStockSpec — BasketStock specification

structure

`BasketStock` specification, specified using `basketstockspec`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a character vector or a 2-by-1 cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price value
scalar numeric | vector

Option strike price value, specified as one of the following:

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

Data Types: `double`

`Settle` — Settlement or trade date
datetime scalar | string scalar | date character vector

Settlement or trade date for the basket option, specified as a scalar datetime, string, or date character vector.

To support existing code, `basketbyju` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime scalar | string scalar | date character vector

Maturity date for the basket option, specified as a scalar datetime, string, or date character vector.

To support existing code, `basketbyju` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`Price` — Expected price for basket option
numeric

Expected price for basket option, returned as a numeric.

More About

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.

For more information, see “Basket Option” on page 3-22.

Version History

Introduced in R2009b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `basketbyju` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`basketstockspec` | `basketsensbyju`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Basket Option” on page 3-22
“Supported Equity Derivative Functions” on page 3-19

basketbyls

Price European or American basket options using Monte Carlo simulations

Syntax

```
[Price,Paths,Times,Z] = basketbyls(RateSpec,BasketStockSpec,OptSpec,Strike,
Settle,ExerciseDates)
[Price,Paths,Times,Z] = basketbyls( ___ ,Name,Value)
```

Description

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

For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium.

`[Price,Paths,Times,Z] = basketbyls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Prices Basket Options Using the Longstaff-Schwartz Model

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 = datetime(2009,5,1);
Maturity  = datetime(2012,5,1);

% 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)

Price = 5.4687
```

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 = 5.5501
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

BasketStockSpec — BasketStock specification

structure

`BasketStock` specification, specified using `basketstockspec`.

Data Types: `struct`

OptSpec — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a character vector or a 2-by-1 cell array of character vectors.

Data Types: `char` | `cell`

Strike — Option strike price value

scalar numeric | vector

Option strike price value, specified as one of the following:

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

Data Types: `double`

Settle — Settlement or trade date

datetime scalar | string scalar | date character vector

Settlement or trade date for the basket option, specified as a scalar datetime, string, or date character vector.

To support existing code, `basketbyls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- 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`.

To support existing code, `basketbyls` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = basketbyls(RateSpec,BasketStockSpec,OptSpec, Strike,Settle,Maturity,'AmericanOpt',AmericanOpt,'NumTrials',NumTrial)`

`AmericanOpt` — Option type

`0` (European/Bermuda) (default) | values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European/Bermuda
- `1` — American

Note For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `double`

`NumPeriods` — Number of simulation periods per trial

`100` (default) | nonnegative integer

Number of simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar nonnegative integer.

Note `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.

Data Types: `double`

`NumTrials` — Number of independent sample paths (simulation trials)
`1000` (default) | nonnegative integer

Number of independent sample paths (simulation trials), specified as the comma-separated pair consisting of `'NumTrials'` and a scalar nonnegative integer.

Data Types: `double`

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-`NINST`-by-`NumTrials` 3-D time series array. The `Z` value generates the Brownian motion vector (that is, Wiener processes) that drives the simulation.

Data Types: `double`

`Antithetic` — Indicator for antithetic sampling
`false` (default) | scalar logical flag with value of `true` or `false`

Indicator for antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a value of `true` or `false`.

Data Types: `logical`

Output Arguments

`Price` — Expected prices for basket option
matrix

Expected prices for basket option, returned as a `NINST`-by-`1` matrix.

`Paths` — Simulated paths of correlated state variables
vector

Simulated paths of correlated state variables, returned as a `NumPeriods + 1`-by-`1`-by-`NumTrials` 3-D time series array of simulated paths of correlated state variables. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths
vector

Observation times associated with simulated paths, returned as a `NumPeriods + 1`-by-`1` column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

Z — Time series array of dependent random variates

vector

Time series array of dependent random variates, returned as a `NumPeriods`-by-`1`-by-`NumTrials` 3-D array when `Z` is specified as an input argument. If the `Z` input argument is not specified, then the `Z` output argument contains the random variates generated internally.

More About

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.

For more information, see “Basket Option” on page 3-22.

Version History

Introduced in R2009b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `basketbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] 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`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Basket Option” on page 3-22
“Supported Equity Derivative Functions” on page 3-19

basketsensbyju

Determine European basket options price or sensitivities using Nengjiu Ju approximation model

Syntax

```
PriceSens = basketsensbyju(RateSpec,BasketStockSpec,OptSpec,Strike,Settle,
Maturity)
PriceSens = basketsensbyju( ___ ,Name,Value)
```

Description

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

`PriceSens = basketsensbyju( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Calculate Prices and Sensitivities for Basket Options Using the Nengjiu Ju Approximation Model

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 = datetime(2009,5,1);
Maturity  = datetime(2010,5,1);

% 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)

Price = 5.1610

Delta = 0.0297
```

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 = 0.0906
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

BasketStockSpec — BasketStock specification

structure

`BasketStock` specification, specified using `basketstockspec`.

Data Types: `struct`

OptSpec — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a character vector or a 2-by-1 cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price value
scalar numeric | vector

Option strike price value, specified as one of the following:

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

Data Types: `double`

`Settle` — Settlement or trade date
datetime scalar | string scalar | date character vector

Settlement or trade date for the basket option, specified as a scalar datetime, string, or date character vector.

To support existing code, `basketsensbyju` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime scalar | string scalar | date character vector

Maturity date for the basket option, specified as a scalar datetime, string, or date character vector.

To support existing code, `basketsensbyju` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = basketsensbyju(RateSpec,BasketStockSpec,OptSpec, Strike,Settle,Maturity,'OutSpec','delta')`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

`UndIdx` — Index of the underlying instrument to compute the sensitivity
`[]` (default) | scalar numeric

Index of the underlying instrument to compute the sensitivity, specified as the comma-separated pair consisting of `'UndIdx'` and a scalar numeric.

Data Types: `double`

Output Arguments

`PriceSens` — Expected prices or sensitivities for basket option

matrix

Expected prices or sensitivities (defined using `OutSpec`) for basket option, returned as a `NINST`-by-`1` matrix.

More About

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.

For more information, see “Basket Option” on page 3-22.

Version History

Introduced in R2009b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `basketsensbyju` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`basketstockspec` | `basketbyju`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Basket Option” on page 3-22
“Supported Equity Derivative Functions” on page 3-19

basketsensbyls

Calculate price and sensitivities for European or American basket options using Monte Carlo simulations

Syntax

```
[PriceSens,Paths,Times,Z] = basketsensbyls(RateSpec,BasketStockSpec,OptSpec,
Strike,Settle,ExerciseDates)
[PriceSens,Paths,Times,Z] = basketsensbyls( ___ ,Name,Value)
```

Description

`[PriceSens,Paths,Times,Z] = basketsensbyls(RateSpec,BasketStockSpec,OptSpec, Strike,Settle,ExerciseDates)` calculates price and sensitivities for European or American basket options using the Longstaff-Schwartz model.

For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium.

`[PriceSens,Paths,Times,Z] = basketsensbyls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Calculate Price and Sensitivities for Basket Options Using the Longstaff-Schwartz Model

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 = datetime(2009,5,1);
Maturity  = datetime(2010,5,1);

% 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)

PriceSens = 0.9822

Delta = -0.0995
```

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 = 0.7814

Delta = -0.0961
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

BasketStockSpec — BasketStock specification

structure

`BasketStock` specification, specified using `basketstockspec`.

Data Types: `struct`

OptSpec — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a character vector or a 2-by-1 cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price value

scalar numeric | vector

Option strike price value, specified as one of the following:

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

Data Types: `double`

`Settle` — Settlement or trade date

datetime scalar | string scalar | date character vector

Settlement or trade date for the basket option, specified as a scalar datetime, string, or date character vector.

To support existing code, `basketsensbyls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- 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`.

To support existing code, `basketsensbyls` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = basketsensbyls(RateSpec,BasketStockSpec,OptSpec, Strike,Settle,Maturity,'AmericanOpt',AmericanOpt,'NumTrials',NumTrial,'OutSpec','delta')`

`AmericanOpt` — Option type

`0` (European/Bermuda) (default) | values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AnericanOpt'` and a `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European/Bermuda

- 1 — American

Note For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `double`

`NumPeriods` — Number of simulation periods per trial

`100` (default) | nonnegative integer

Number of simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar nonnegative integer.

Note `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.

Data Types: `double`

`NumTrials` — Number of independent sample paths (simulation trials)

`1000` (default) | nonnegative integer

Number of independent sample paths (simulation trials), specified as the comma-separated pair consisting of `'NumTrials'` and a scalar nonnegative integer.

Data Types: `double`

`Z` — Time series array of dependent random variates

vector

Time series array of dependent random variates, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-`NINST`-by-`NumTrials` 3-D time series array. The `Z` value generates the Brownian motion vector (that is, Wiener processes) that drives the simulation.

Data Types: `double`

`Antithetic` — Indicator for antithetic sampling

`false` (default) | scalar logical flag with value of `true` or `false`

Indicator for antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a value of `true` or `false`.

Data Types: `logical`

`OutSpec` — Define outputs

`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-1 or a 1-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

`UndIdx` — Index of the underlying instrument to compute the sensitivity
`[]` (default) | scalar numeric

Index of the underlying instrument to compute the sensitivity, specified as the comma-separated pair consisting of `'UndIdx'` and a scalar numeric.

Data Types: `double`

Output Arguments

`PriceSens` — Expected prices or sensitivities for basket option
matrix

Expected prices or sensitivities (defined using `OutSpec`) for basket option, returned as a `NINST`-by-`1` matrix.

`Paths` — Simulated paths of correlated state variables
vector

Simulated paths of correlated state variables, returned as a `NumPeriods + 1`-by-`1`-by-`NumTrials` 3-D time series array of simulated paths of correlated state variables. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths
vector

Observation times associated with simulated paths, returned as a `NumPeriods + 1`-by-`1` column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, returned as a `NumPeriods`-by-`1`-by-`NumTrials` 3-D array when `Z` is specified as an input argument. If the `Z` input argument is not specified, then the `Z` output argument contains the random variates generated internally.

More About

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.

For more information, see “Basket Option” on page 3-22.

Version History

Introduced in R2009b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `basketsensbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] 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`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Basket Option” on page 3-22
“Supported Equity Derivative Functions” on page 3-19

basketstockspec

Specify basket stock structure using Longstaff-Schwartz model

Syntax

```
BasketStockSpec = basketstockspec(Sigma,AssetPrice,Quantity,Correlation)
BasketStockSpec = basketstockspec( ___ ,Name,Value)
```

Description

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

`BasketStockSpec = basketstockspec( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Create a Basket Stock Structure for Three Stocks

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)
```

```
BasketStockSpec = struct with fields:
             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 = 3×3

    1.0000    0.3000    0.1600
    0.3000    1.0000    0.1100
    0.1600    0.1100    1.0000
```

Create a Basket Stock Structure for Two Stocks

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)
```

```
BasketStockSpec = struct with fields:
             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 = 2x1 cell
    {'cash'      }
    {'continuous'}
```

Input Arguments

`Sigma` — Annual price volatility of the underlying security

vector in decimals

Annual price volatility of the underlying security, specified as an `NINST`-by-1 vector in decimals.

Data Types: `double`

`AssetPrice` — Underlying asset price values at time 0

vector

Underlying asset price values at time 0, specified as a `NINST`-by-1 vector.

Data Types: `double`

`Quantity` — Quantities of the instruments contained in the basket

vector

Quantities of the instruments contained in the basket, specified as an `NINST`-by-1 vector.

Data Types: `double`

`Correlation` — Correlation values

matrix

Correlation values, specified as an `NINST`-by-1 matrix.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BasketStockSpec = basketstockspec(Sigma,AssetPrice,Quantity,Correlation,'DividendType',DivType, 'DividendAmounts',DivAmounts)`

`DividendAmounts` — Dividend amounts for basket instruments
row vector

Dividend amounts for basket instruments, specified as the comma-separated pair consisting of `'DividendAmounts'` and an `NINST`-by-`1` cell array. 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.

Data Types: `double`

`DividendType` — Stock dividend type
cell array of character vectors

Stock dividend type, specified as the comma-separated pair consisting of `'DividendType'` and an `NINST`-by-`1` cell array of character vectors specifying each stock's dividend type. Dividend type must be either `cash` for actual dollar dividends or `continuous` for continuous dividend yield.

Data Types: `char` | `cell`

`ExDividendDates` — Ex-dividend dates for the basket instruments
cell array

Ex-dividend dates for the basket instruments, specified as the comma-separated pair consisting of `'ExDividendDates'` and an `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`.

Data Types: `cell`

Output Arguments

`BasketStockSpec` — Structure encapsulating the properties of a basket stock structure
structure

Structure encapsulating the properties of a basket stock structure, returned as a structure.

Version History

Introduced in R2009b

See Also

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

Topics

“Portfolio Creation Using Functions” on page 1-6
“Hedging Functions” on page 4-3
“Basket Option” on page 3-22
“Instrument Constructors” on page 1-15

“Supported Equity Derivative Functions” on page 3-19

bdtprice

Instrument prices from Black-Derman-Toy interest-rate tree

Syntax

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

Description

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

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

`[Price,PriceTree] = bdtprice( ___ ,Options)` adds an optional input argument for `Options`.

Examples

Price a Cap and Bond Instruments Contained in an Instrument Set

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

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

instdisp(BDTSubSet)
```

```
instdisp(BDTSubSet)
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCouponDate LastCouponDate StartDate Face Na
1     Bond 0.1        01-Jan-2000    01-Jan-2003    1      NaN   NaN          NaN       NaN             NaN            NaN       NaN  10
2     Bond 0.1        01-Jan-2000    01-Jan-2004    2      NaN   NaN          NaN       NaN             NaN            NaN       NaN  10

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

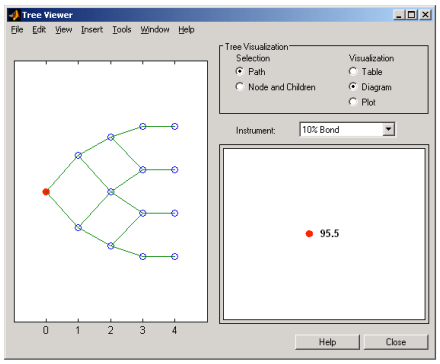
Price the bond and cap.

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

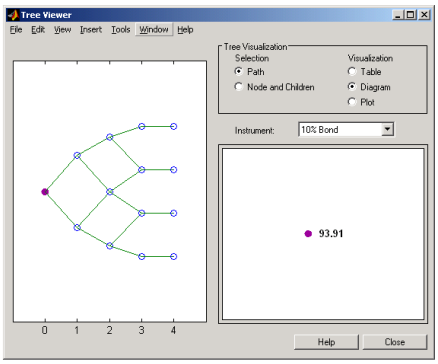
```
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In checktree at 289
  In bdtprice at 85

Price =

   95.5030
   93.9079
    1.4375


PriceTree =
```

```
    FinObj: 'BDTPriceTree'
     PTree: {[3x1 double]  [3x2 double]  [3x3 double]  [3x4 double]  [3x4 double]}
    AITree: {[3x1 double]  [3x2 double]  [3x3 double]  [3x4 double]  [3x4 double]}
      tObs: [0 1 2 3 4]
```

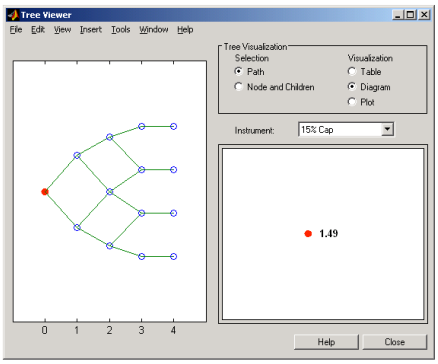
You can use the `treeviewer` function to see the prices of these three instruments along the price tree.



First 10% Bond (Maturity 2003)



Second 10% Bond (Maturity 2004)



15% Cap

Price Multi-Stepped Coupon Bonds

Price the following multi-stepped coupon bonds using the following data:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2010';
StartDates = ValuationDate;
EndDates = {'Jan-1-2011'; 'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'};
Compounding = 1;

% Create RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);

% Create a portfolio of stepped coupon bonds with different maturities
Settle = '01-Jan-2010';
Maturity = {'01-Jan-2011';'01-Jan-2012';'01-Jan-2013';'01-Jan-2014'};
```

```
CouponRate = {{'01-Jan-2011' .042;'01-Jan-2012' .05; '01-Jan-2013' .06; '01-Jan-2014' .07}};

% Display the instrument portfolio
ISet = instbond(CouponRate, Settle, Maturity, 1);
instdisp(ISet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond [Cell]     01-Jan-2010    01-Jan-2011    1      0     1            NaN       NaN
2     Bond [Cell]     01-Jan-2010    01-Jan-2012    1      0     1            NaN       NaN
3     Bond [Cell]     01-Jan-2010    01-Jan-2013    1      0     1            NaN       NaN
4     Bond [Cell]     01-Jan-2010    01-Jan-2014    1      0     1            NaN       NaN
```

Build a `BDTTree` to price the stepped coupon bonds. Assume the volatility to be 10%

```
Sigma = 0.1;
BDTTimeSpec = bdttimespec(ValuationDate, EndDates, Compounding);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVolSpec, RS, BDTTimeSpec);

% Compute the price of the stepped coupon bonds
PBDT = bdtprice(BDTT, ISet)
```

```
PBDT = 4×1

  100.6763
  100.7368
  100.9266
  101.0115
```

Price a Portfolio of Stepped Callable Bonds and Stepped Vanilla Bonds

Price a portfolio of stepped callable bonds and stepped vanilla bonds using the following data: The data for the interest rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2010';
StartDates = ValuationDate;
EndDates = {'Jan-1-2011'; 'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'};
Compounding = 1;

%Create RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);

% Create an instrument portfolio of 3 stepped callable bonds and three
% stepped vanilla bonds
Settle = '01-Jan-2010';
Maturity = {'01-Jan-2012';'01-Jan-2013';'01-Jan-2014'};
CouponRate = {{'01-Jan-2011' .042;'01-Jan-2012' .05; '01-Jan-2013' .06; '01-Jan-2014' .07}};
OptSpec='call';
Strike=100;
ExerciseDates='01-Jan-2011'; %Callable in one year

% Bonds with embedded option
```

```
ISet = instoptembnd(CouponRate, Settle, Maturity, OptSpec, Strike,...
ExerciseDates, 'Period', 1);

% Vanilla bonds
ISet = instbond(ISet, CouponRate, Settle, Maturity, 1);

% Display the instrument portfolio
instdisp(ISet)
```

```
Index Type      CouponRate Settle         Maturity       OptSpec Strike ExerciseDates  Period Bas
1     OptEmBond [Cell]     01-Jan-2010    01-Jan-2012    call    100    01-Jan-2011    1      0
2     OptEmBond [Cell]     01-Jan-2010    01-Jan-2013    call    100    01-Jan-2011    1      0
3     OptEmBond [Cell]     01-Jan-2010    01-Jan-2014    call    100    01-Jan-2011    1      0

Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCoup
4     Bond [Cell]     01-Jan-2010    01-Jan-2012    1      0     1            NaN       NaN
5     Bond [Cell]     01-Jan-2010    01-Jan-2013    1      0     1            NaN       NaN
6     Bond [Cell]     01-Jan-2010    01-Jan-2014    1      0     1            NaN       NaN
```

Build a `BDTTree` and price the instruments. Build the tree Assume the volatility to be 10%

```
Sigma = 0.1;
BDTTimeSpec = bdttimespec(ValuationDate, EndDates, Compounding);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVolSpec, RS, BDTTimeSpec);

%The first three rows corresponds to the price of the stepped callable bonds and the
%last three rows corresponds to the price of the stepped vanilla bonds.
PBDT = bdtprice(BDTT, ISet)
```

```
PBDT = 6×1

  100.4799
  100.3228
  100.0840
  100.7368
  100.9266
  101.0115
```

Price a Portfolio with Range Notes and a Floating Rate Note

Compute the price of a portfolio with range notes and a floating rate note using the following data: The data for the interest rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2011';
StartDates = ValuationDate;
EndDates = {'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'; 'Jan-1-2015'};
Compounding = 1;

%  Create RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);
```

```
% Create an instrument portfolio with two range notes and a floating rate
% note with the following data:
Spread = 200;
Settle = 'Jan-1-2011';
Maturity = 'Jan-1-2014';

% First Range Note:
RateSched(1).Dates = {'Jan-1-2012'; 'Jan-1-2013'  ; 'Jan-1-2014'};
RateSched(1).Rates  = [0.045 0.055; 0.0525  0.0675; 0.06 0.08];

% Second Range Note:
RateSched(2).Dates = {'Jan-1-2012'; 'Jan-1-2013' ; 'Jan-1-2014'};
RateSched(2).Rates  = [0.048 0.059; 0.055  0.068 ; 0.07 0.09];

% Create InstSet
InstSet = instadd('RangeFloat', Spread, Settle, Maturity, RateSched);

% Add a floating-rate note
InstSet = instadd(InstSet, 'Float', Spread, Settle, Maturity);

% Display the portfolio instrument
instdisp(InstSet)
```

```
Index Type       Spread Settle         Maturity       RateSched FloatReset Basis Principal EndMo
1     RangeFloat 200    01-Jan-2011    01-Jan-2014    [Struct]  1          0     100       1
2     RangeFloat 200    01-Jan-2011    01-Jan-2014    [Struct]  1          0     100       1

Index Type  Spread Settle         Maturity       FloatReset Basis Principal EndMonthRule CapRate
3     Float 200    01-Jan-2011    01-Jan-2014    1          0     100       1            Inf
```

Build a `BDTTree` and price the instruments. Build the tree Assume the volatility to be 10%.

```
Sigma = 0.1;
BDTTS = bdttimespec(ValuationDate, EndDates, Compounding);
BDTVS = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVS, RS, BDTTS);

% Price the portfolio
Price = bdtprice(BDTT, InstSet)
```

```
Price = 3×1

  100.2841
   98.0757
  105.5147
```

Create a Float-Float Swap and Price with `bdtprice`

Use `instswap` to create a float-float swap and price the swap with `bdtprice`.

```
RateSpec = intenvset('Rates',.05,'StartDate',today,'EndDate',datemnth(today,60));
IS = instswap([.02 .03],today,datemnth(today,60),[], [], [], [1 1]);
VolSpec = bdtvolspec(today,datemnth(today,[10 60]),[.01 .02]);
TimeSpec = bdttimespec(today,cfdates(today,datemnth(today,60),1));
```

```
BDTTree = bdttree(VolSpec,RateSpec,TimeSpec);
bdtprice(BDTTree,IS)
```

```
ans = -4.3220
```

Price Multiple Swaps with bdtprice

Use `instswap` to create multiple swaps and price the swaps with `bdtprice`.

```
RateSpec = intenvset('Rates',.05,'StartDate',today,'EndDate',datemnth(today,60));
IS = instswap([.03 .02],today,datemnth(today,60),[], [], [], [1 1]);
IS = instswap(IS,[200 300],today,datemnth(today,60),[], [], [], [0 0]);
IS = instswap(IS,[.08 300],today,datemnth(today,60),[], [], [], [1 0]);
VolSpec = bdtvolspec(today,datemnth(today,[10 60]),[.01 .02]);
TimeSpec = bdttimespec(today,cfdates(today,datemnth(today,60),1));
BDTTree = bdttree(VolSpec,RateSpec,TimeSpec);
bdtprice(BDTTree,IS)
```

```
ans = 3×1

    4.3220
   -4.3220
   -0.2701
```

Input Arguments

BDTTree — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bdttree`.

Data Types: `struct`

InstSet — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

Options — Derivatives pricing options structure

structure

(Optional) Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

Price — Price for each instrument at time 0

vector

Price for each instrument at time 0, returned as a `NINST`-by-`1` vector. The prices are computed by backward dynamic programming on the interest-rate tree. If an instrument cannot be priced, a `NaN` is returned in that entry.

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.
- `optfloatbybdt` — Price a floating-rate note with an option from a BDT tree.
- `optemfloatbybdt` — Price a floating-rate note with an embedded option from a BDT tree.
- `optembndbybdt` — Price a bond with embedded option by a BDT tree.
- `rangefloatbybdt` — Price range floating note using a BDT tree.
- `swapbybdt` — Price a swap from a BDT tree.
- `swaptionbybdt` — Price a swaption from a BDT tree.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.

Version History

Introduced before R2006a

See Also

`bdtsens` | `bdttree` | `intenvprice` | `instadd` | `intenvsens`

Topics

“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bdtsens

Instrument prices and sensitivities from Black-Derman-Toy interest-rate tree

Syntax

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

Description

`[Delta,Gamma,Vega,Price] = bdtsens(BDTTree,InstSet)` computes instrument sensitivities and prices for instruments using an interest-rate tree created with the `bdttree` function. All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide by the respective instrument price.

`bdtsens` handles instrument types: `'Bond'`, `'CashFlow'`, `'OptBond'`, `'OptEmBond'`, `'OptEmBond'`, `'OptFloat'`, `'OptEmFloat'`, `'Fixed'`, `'Float'`, `'Cap'`, `'Floor'`, `'RangeFloat'`, `'Swap'`. See `instadd` for information on instrument types.

`[Delta,Gamma,Vega,Price] = bdtsens(___,Options)` adds an optional input argument for `Options`.

Examples

Compute Instrument Sensitivities and Prices for Cap and Bond instruments

Load the tree and instruments from the `deriv.mat` data file.

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

instdisp(BDTSubSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.1        01-Jan-2000    01-Jan-2003    1      NaN   NaN          NaN       NaN
2     Bond 0.1        01-Jan-2000    01-Jan-2004    2      NaN   NaN          NaN       NaN

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

Compute `Delta` and `Gamma` for the cap and bond instruments contained in the instrument set.

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

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

Delta = 3×1

 -232.6681
 -281.0517
```

```
   63.8102


Gamma = 3×1
10^3 ×

    0.8037
    1.1819
    1.8535
```

Input Arguments

`BDTTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bdttree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

(Optional) Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Delta` — Rate of change of instruments prices with respect to changes in interest rate

vector

Rate of change of instruments prices with respect to changes in the interest rate, returned as a `NINST`-by-1 vector of deltas. `Delta` is computed by finite differences in calls to `bdttree`.

Note `Delta` is calculated based on yield shifts of 100 basis points.

`Gamma` — Rate of change of instruments deltas with respect to changes in interest rate

vector

Rate of change of instruments deltas with respect to changes in the interest rate, returned as a `NINST`-by-1 vector of gammas. `Gamma` is computed by finite differences in calls to `bdttree`.

Note `Gamma` is calculated based on yield shifts of 100 basis points.

`Vega` — Rate of change of instruments prices with respect to changes in volatility
vector

Rate of change of instruments prices with respect to changes in the volatility, returned as a `NINST`-by-`1` vector of vegas. Volatility is $\sigma(t, T)$ of the interest rate. `Vega` is computed by finite differences in calls to `bdttree`. For information on the volatility process, see `bdtvolspec`.

Note `Vega` is calculated based on 1% shift in the volatility process.

`Price` — Price of each instrument
vector

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

Version History

Introduced before R2006a

See Also

`bdtprice` | `bdttree` | `bdtvolspec` | `instadd`

Topics

“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bdttimespec

Specify time structure for Black-Derman-Toy interest-rate tree

Syntax

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

Description

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

`TimeSpec = bdttimespec( ___ ,Compounding)` adds the optional argument `Compounding`.

Examples

Specify a Five-Period Tree with Annual Nodes

This example shows how to specify a five-period tree with annual nodes and use annual compounding to report rates.

```
Compounding = 1;
ValuationDate = datetime(2000,1,1);
Maturity = [datetime(2001,1,1) ; datetime(2002,1,1) ; datetime(2003,1,1) ;
datetime(2004,1,1); datetime(2005,1,1)];

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

```
TimeSpec = struct with fields:
           FinObj: 'BDTTimeSpec'
    ValuationDate: 730486
         Maturity: [5x1 double]
      Compounding: 1
            Basis: 0
     EndMonthRule: 1
```

Input Arguments

`ValuationDate` — Pricing date and first observation in the tree

datetime scalar | string scalar | date character vector

Pricing date and first observation in the tree, specified as a scalar datetime, string, or date character vector.

To support existing code, `bdttimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Dates marking the cash flow dates of the tree
datetime array | string array | date character vector

Dates marking the cash flow dates of the tree, specified as `NLEVELS`-by-`1` vector using a datetime array, string array, or date character vectors. Cash flows with these maturities fall on tree nodes. `Maturity` should be in increasing order.

To support existing code, `bdttimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Compounding` — Rate at which the input zero rates were compounded when annualized
1 (default) | integer with value of 1, 2, 3, 4, 6, 12, 365, or -1

(Optional) Rate at which the input zero rates were compounded when annualized, specified as a scalar integer value.

- If `Compounding` = 1, 2, 3, 4, 6, 12:

  `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units; for example, `T = F` is one year.
- If `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.
- If `Compounding` = −1:

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

Data Types: `double`

Output Arguments

`TimeSpec` — Specification for the time layout for `bdttree`
structure

Specification for the time layout for `bdttree`, returned as a structure. 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(end)`.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `bdttimespec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdttree` | `bdtprice` | `bdtvolspec` | `instadd`

Topics

“Specifying the Time Structure (TimeSpec)” on page 2-70
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bdttree

Build Black-Derman-Toy interest-rate tree

Syntax

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

Description

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

Examples

Create a BDTTree

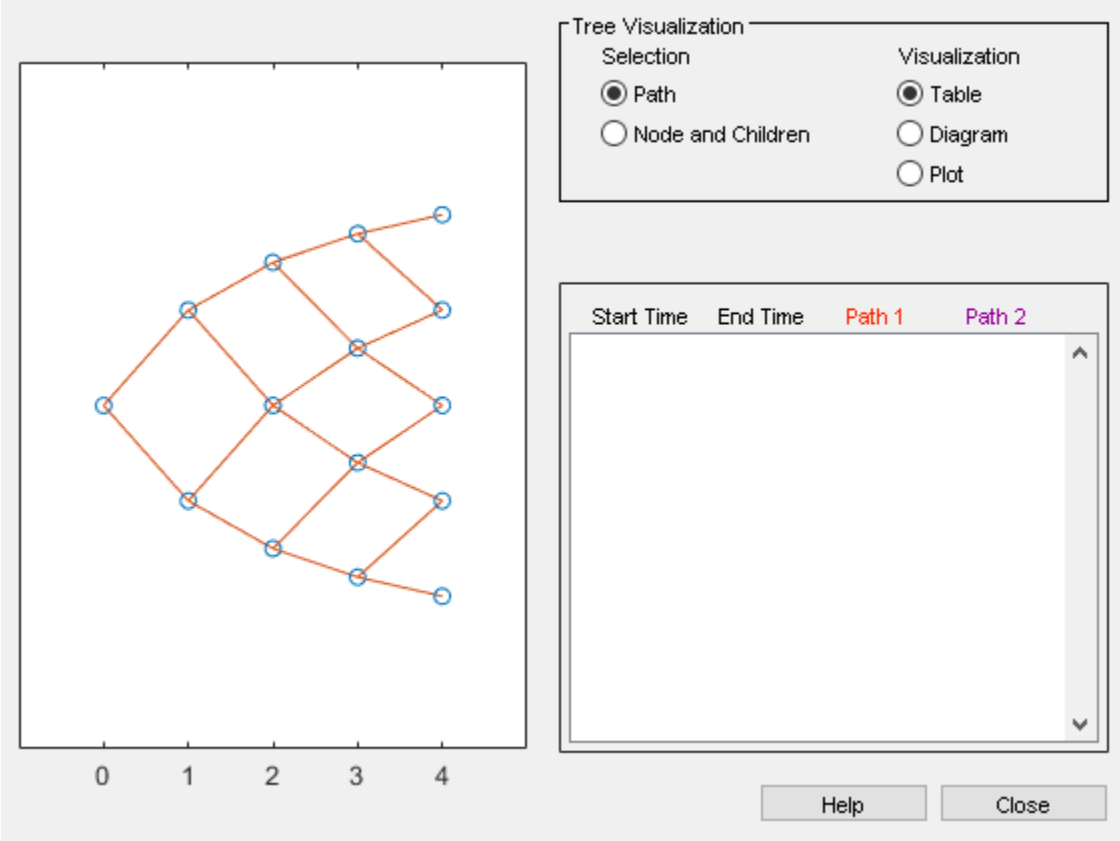
Using the data provided, create a BDT volatility specification (using `bdtvolspec`), rate specification (using `intenvset`), and tree time layout specification (using `bdttimespec`). Then use these specifications to create a BDT tree with `bdttree`.

```
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)
```

```
BDTTree = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 4]
        dObs: [730486 730852 731217 731582 731947]
        TFwd: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [4]}
      CFlowT: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [5]}
     FwdTree: {1x5 cell}
```

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

```
treeviewer(BDTTree)
```



Input Arguments

`VolSpec` — Volatility process specification

structure

Volatility process specification, specified using the `VolSpec` output obtained from `bdtvolspec`.

Data Types: `struct`

`RateSpec` — Interest-rate specification for initial risk-free rate curve

structure

Interest-rate specification for initial rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`TimeSpec` — Time tree layout specification

structure

Time tree layout specification, specified using the `TimeSpec` output obtained from `bdttimespec`. The TimeSpec defines the observation dates of the BDT tree and the `Compounding` rule for date to time mapping and price-yield formulas.

Data Types: `struct`

Output Arguments

`BDTTree` — Time and interest-rate information of a recombining tree

structure

Time and interest-rate information of a recombining tree, returned as a structure.

Version History

Introduced before R2006a

See Also

`bdttree` | `bdtprice` | `bdtvolspec` | `instadd` | `bdttimespec` | `intenvset`

Topics

“Specifying the Interest-Rate Term Structure (RateSpec)” on page 2-70
“Specifying the Time Structure (TimeSpec)” on page 2-70
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bdtvolspec

Specify Black-Derman-Toy interest-rate volatility process

Syntax

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

Description

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

`VolSpec = bdtvolspec( ___ ,InterpMethod)` adds the optional argument `InterpMethod`.

Examples

Create a BDT Volatility Specification

This example shows how to create a BDT volatility specification (VolSpec) using the following data.

```
ValuationDate = '01-01-2000';
EndDates = [datetime(2001,1,1) ; datetime(2002,1,1) ; datetime(2003,1,1) ;
datetime(2004,1,1); datetime(2005,1,1)];
Volatility = [.2; .19; .18; .17; .16];

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

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

Input Arguments

`ValuationDate` — Observation date of the investment horizon

character vector date | datetime

Observation date of the investment horizon, specified as a scalar datetime, string, or date character vector.

To support existing code, `bdtvolspec` also accepts serial date numbers as inputs, but they are not recommended.

`VolDates` — Number of points of yield volatility end dates

datetime array | string array | date character vector

Number of points of yield volatility end dates, specified as a `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bdtvolspec` also accepts serial date numbers as inputs, but they are not recommended.

`VolCurve` — Yield volatility values
decimal

Yield volatility values, specified as a `NPOINTS`-by-`1` vector of decimal values. The term structure of `VolCurve` is the yield volatility represented by the value of the volatility of the yield from time `t` = 0 to time `t` + *i*, where *i* is any point within the volatility curve.

Data Types: `double`

`InterpMethod` — Interpolation method
`'linear'` (default) | character vector with values supported by `interp1`

(Optional) Interpolation method, specified as a character vector with values supported by `interp1`.

Data Types: `char`

Output Arguments

`VolSpec` — Specification for the volatility model for `bdttree`
structure

Structure specifying the volatility model for `bdttree`.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `bdtvolspec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdttree` | `interp1`

Topics

“Specifying the Volatility Model (VolSpec)” on page 2-68
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bkprice

Instrument prices from Black-Karasinski interest-rate tree

Syntax

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

Description

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

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

`[Price,PriceTree] = bkprice( ___ ,Options)` adds an optional input argument for `Options`.

Examples

Price Cap and Bond Instruments in the Instrument Set

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)
```

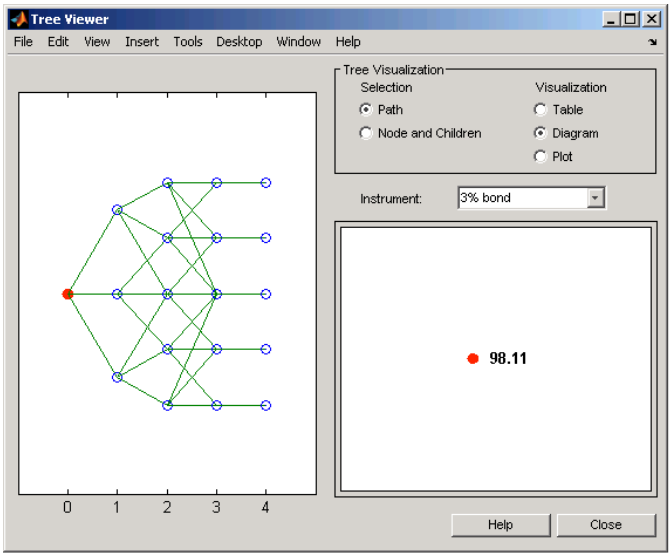
```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.03       01-Jan-2004    01-Jan-2007    1      0     1            NaN       NaN
2     Bond 0.03       01-Jan-2004    01-Jan-2008    1      0     1            NaN       NaN

Index Type Strike Settle         Maturity       CapReset Basis Principal Name   Quantity
3     Cap  0.04   01-Jan-2004    01-Jan-2008    1        0     100       4% Cap 10
```

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

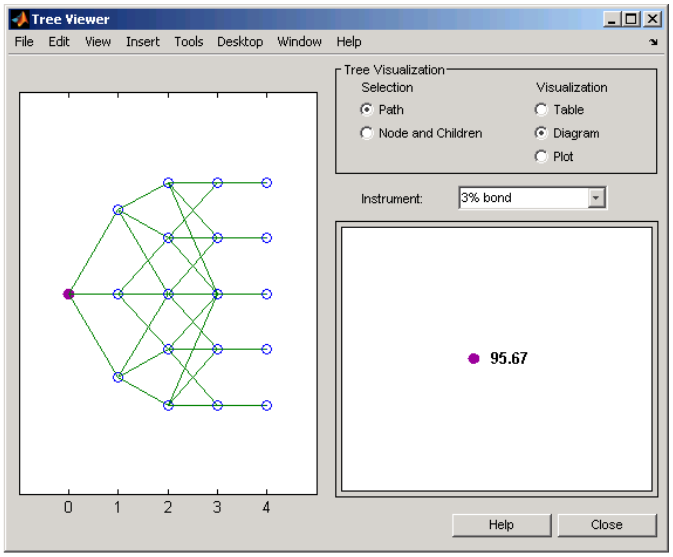
```
Price = 3×1

   98.1096
   95.6734
    2.2706


PriceTree = struct with fields:
     FinObj: 'BKPriceTree'
```

```
    PTree: {1x5 cell}
   AITree: {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]}
```

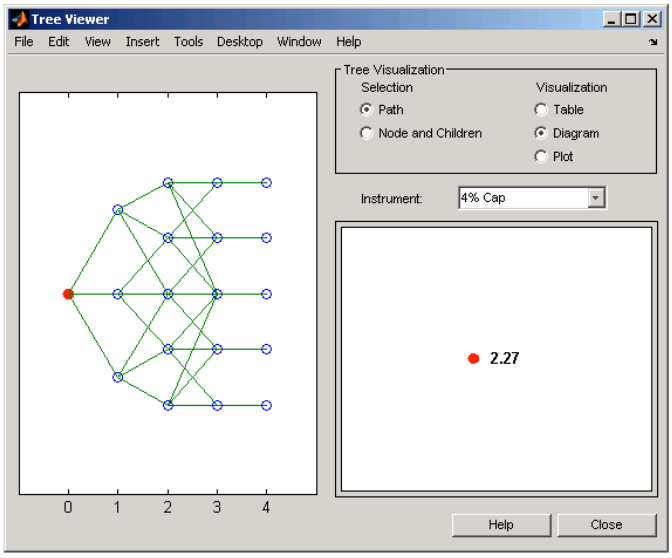
You can use `treeviewer` to see the prices of these three instruments along the price tree.



First 3% Bond (Maturity 2007)



Second 3% Bond (Maturity 2008)



4% Cap

Price Multi-Stepped Coupon Bonds

Price the following multi-stepped coupon bonds using the following data:

```
% The data for the interest rate term structure is as follows:
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2010';
StartDates = ValuationDate;
EndDates = {'Jan-1-2011'; 'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'};
Compounding = 1;

% Create RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);

% Create a portfolio of stepped coupon bonds with different maturities
Settle = '01-Jan-2010';
Maturity = {'01-Jan-2011';'01-Jan-2012';'01-Jan-2013';'01-Jan-2014'};
CouponRate = {{'01-Jan-2011' .042;'01-Jan-2012' .05; '01-Jan-2013' .06; '01-Jan-2014' .07}};

ISet = instbond(CouponRate, Settle, Maturity, 1);
instdisp(ISet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond [Cell]     01-Jan-2010    01-Jan-2011    1      0     1            NaN       NaN
2     Bond [Cell]     01-Jan-2010    01-Jan-2012    1      0     1            NaN       NaN
3     Bond [Cell]     01-Jan-2010    01-Jan-2013    1      0     1            NaN       NaN
4     Bond [Cell]     01-Jan-2010    01-Jan-2014    1      0     1            NaN       NaN
```

Build the `BKTree` with the following data:

```
VolDates = ['1-Jan-2011'; '1-Jan-2012'; '1-Jan-2013'; '1-Jan-2014'];
VolCurve = 0.01;
AlphaDates = '01-01-2014';
AlphaCurve = 0.1;

BKVolSpec = bkvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RS.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RS, BKTimeSpec);
```

Compute the price of the stepped coupon bonds.

```
PBK = bkprice(BKT, ISet)
```

```
PBK = 4×1

  100.6763
  100.7368
  100.9266
  101.0115
```

Price a Portfolio of Stepped Callable Bonds and Stepped Vanilla Bonds

Price a portfolio of stepped callable bonds and stepped vanilla bonds using the following data:

```
% The data for the interest rate term structure is as follows:
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2010';
StartDates = ValuationDate;
EndDates = {'Jan-1-2011'; 'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'};
Compounding = 1;

% Create RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);

% Create an instrument portfolio of 3 stepped callable bonds and three
% stepped vanilla bonds
Settle = '01-Jan-2010';
Maturity = {'01-Jan-2012';'01-Jan-2013';'01-Jan-2014'};
CouponRate = {{'01-Jan-2011' .042;'01-Jan-2012' .05; '01-Jan-2013' .06; '01-Jan-2014' .07}};
OptSpec='call';
Strike=100;
ExerciseDates='01-Jan-2011'; % Callable in one year

% Bonds with embedded option
ISet = instoptembnd(CouponRate, Settle, Maturity, OptSpec, Strike,...
ExerciseDates, 'Period', 1);

% Vanilla bonds
ISet = instbond(ISet, CouponRate, Settle, Maturity, 1);

% Display the instrument portfolio
instdisp(ISet)
```

```
Index Type      CouponRate Settle         Maturity       OptSpec Strike ExerciseDates  Period Bas
1     OptEmBond [Cell]     01-Jan-2010    01-Jan-2012    call    100    01-Jan-2011    1      0
2     OptEmBond [Cell]     01-Jan-2010    01-Jan-2013    call    100    01-Jan-2011    1      0
3     OptEmBond [Cell]     01-Jan-2010    01-Jan-2014    call    100    01-Jan-2011    1      0

Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
4     Bond [Cell]     01-Jan-2010    01-Jan-2012    1      0     1            NaN       NaN
5     Bond [Cell]     01-Jan-2010    01-Jan-2013    1      0     1            NaN       NaN
6     Bond [Cell]     01-Jan-2010    01-Jan-2014    1      0     1            NaN       NaN
```

Build the `BKTree` with the following data:

```
VolDates = ['1-Jan-2011'; '1-Jan-2012'; '1-Jan-2013'; '1-Jan-2014'];
VolCurve = 0.01;
AlphaDates = '01-01-2014';
AlphaCurve = 0.1;

BKVolSpec = bkvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RS.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RS, BKTimeSpec);
```

Compute the price, where the first three rows of the output corresponds to the price of the stepped callable bonds and the last three rows corresponds to the price of the stepped vanilla bonds.

```
PBK = bkprice(BKT, ISet)

PBK = 6×1

  100.6729
  100.6763
  100.6763
  100.7368
  100.9266
  101.0115
```

Price a Portfolio of Range Notes and Floating-Rate Notes

Price a portfolio of range notes and floating-rate notes using the following data:

```
% The data for the interest rate term structure is as follows:
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2011';
StartDates = ValuationDate;
EndDates = {'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'; 'Jan-1-2015'};
Compounding = 1;

%  Create RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates',...
StartDates, 'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);

% Create an instrument portfolio with two range notes and a floating rate
% note with the following data:
Spread = 200;
Settle = 'Jan-1-2011';
Maturity = 'Jan-1-2014';

% First Range Note:
RateSched(1).Dates = {'Jan-1-2012'; 'Jan-1-2013'  ; 'Jan-1-2014'};
RateSched(1).Rates  = [0.045 0.055; 0.0525  0.0675; 0.06 0.08];

% Second Range Note:
RateSched(2).Dates = {'Jan-1-2012'; 'Jan-1-2013' ; 'Jan-1-2014'};
RateSched(2).Rates  = [0.048 0.059; 0.055  0.068 ; 0.07 0.09];

% Create InstSet
InstSet = instadd('RangeFloat', Spread, Settle, Maturity, RateSched);

% Add a floating-rate note
InstSet = instadd(InstSet, 'Float', Spread, Settle, Maturity);

% Display the portfolio instrument
instdisp(InstSet)

Index Type       Spread Settle         Maturity       RateSched FloatReset Basis Principal EndMon
1     RangeFloat 200    01-Jan-2011    01-Jan-2014    [Struct]  1          0     100       1
2     RangeFloat 200    01-Jan-2011    01-Jan-2014    [Struct]  1          0     100       1

Index Type  Spread Settle         Maturity       FloatReset Basis Principal EndMonthRule CapRate
```

```
3     Float 200    01-Jan-2011    01-Jan-2014    1          0     100       1            Inf
```

Build the `BKTree` with the following data:

```
VolDates = ['1-Jan-2012'; '1-Jan-2013'; '1-Jan-2014';'1-Jan-2015'];
VolCurve = 0.01;
AlphaDates = '01-01-2015';
AlphaCurve = 0.1;

BKVS = bkvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTS = bktimespec(RS.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVS, RS, BKTS);
```

Price the portfolio.

```
Price = bkprice(BKT, InstSet)
```

```
Price = 3×1

  105.5147
  101.4805
  105.5147
```

Input Arguments

`BKTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bktree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

(Optional) Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Price for each instrument at time 0

vector

Price for each instrument at time 0, returned as a `NINST`-by-`1` vector. The prices are computed by backward dynamic programming on the interest-rate tree. If an instrument cannot be priced, a `NaN` is returned in that entry.

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.
- `optfloatbybk` — Price a floating-rate note with an option from a Black-Karasinski tree.
- `optemfloatbybk` — Price a floating-rate note with an embedded option from a Black-Karasinski tree.
- `rangefloatbybk` — Price range floating note from a Black-Karasinski tree.
- `swapbybk` — Price a swap from a Black-Karasinski tree.
- `swaptionbybk` — Price a swaption from a Black-Karasinski tree.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

Version History

Introduced before R2006a

See Also

`bksens` | `bdttree` | `instadd` | `intenvprice` | `intenvsens`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

bksens

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

Syntax

```
[Delta,Gamma,Vega,Price] = bktsens(BKTree,InstSet)
[Delta,Gamma,Vega,Price] = bksens(___,Options)
```

Description

`[Delta,Gamma,Vega,Price] = bktsens(BKTree,InstSet)` computes instrument sensitivities and prices for instruments using an interest-rate tree created with the `bktree` function. All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide by the respective instrument price.

`bksens` handles instrument types: `'Bond'`, `'CashFlow'`, `'OptBond'`, `'OptEmBond'`, `'OptEmBond'`, `'OptFloat'`, `'OptEmFloat'`, `'Fixed'`, `'Float'`, `'Cap'`, `'Floor'`, `'RangeFloat'`, `'Swap'`. See `instadd` for information on instrument types.

`[Delta,Gamma,Vega,Price] = bksens(___,Options)` adds an optional input argument for `Options`.

Examples

Compute Instrument Sensitivities and Prices for Cap and Bond Instruments

Load the tree and instruments from the `deriv.mat` data file.

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

instdisp(BKSubSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.03       01-Jan-2004    01-Jan-2007    1      0     1            NaN       NaN
2     Bond 0.03       01-Jan-2004    01-Jan-2008    1      0     1            NaN       NaN

Index Type Strike Settle         Maturity       CapReset Basis Principal Name   Quantity
3     Cap  0.04   01-Jan-2004    01-Jan-2008    1        0     100       4% Cap 10
```

Compute `Delta` and `Gamma` for the cap and bond instruments contained in the instrument set.

```
[Delta, Gamma] = bksens(BKTree, BKSubSet)
```

```
Delta = 3×1

 -285.7151
 -365.7048
  189.5319
```

```
Gamma = 3×1
10^3 ×

    0.8456
    1.4345
    6.9999
```

Input Arguments

`BKTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bktree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Delta` — Rate of change of instruments prices with respect to changes in interest rate

vector

Rate of change of instruments prices with respect to changes in the interest rate, returned as a `NINST`-by-`1` vector of deltas. `Delta` is computed by finite differences in calls to `bktree`.

Note `Delta` is calculated based on yield shifts of 100 basis points.

`Gamma` — Rate of change of instruments deltas with respect to changes in interest rate

vector

Rate of change of instruments deltas with respect to changes in the interest rate, returned as a `NINST`-by-`1` vector of gammas. `Gamma` is computed by finite differences in calls to `bktree`.

Note `Gamma` is calculated based on yield shifts of 100 basis points.

`Vega` — Rate of change of instruments prices with respect to changes in volatility

vector

Rate of change of instruments prices with respect to changes in the volatility, returned as a `NINST`-by-`1` vector of vegas. Volatility is $\sigma(t, T)$ of the interest rate. `Vega` is computed by finite differences in calls to `bktree`. For information on the volatility process, see `bkvolspec`.

Note `Vega` is calculated based on 1% shift in the volatility process.

`Price` — Price of each instrument

vector

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

Version History

Introduced before R2006a

See Also

`bkprice` | `bktree` | `bkvolspec` | `instadd`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bktimespec

Specify time structure for Black-Karasinski tree

Syntax

```
TimeSpec = bktimespec(ValuationDate,Maturity)
TimeSpec = bktimespec( ___ ,Compounding)
```

Description

`TimeSpec = bktimespec(ValuationDate,Maturity)` sets the number of levels and node times for a `bktree` and determines the mapping between dates and time for rate quoting.

`TimeSpec = bktimespec( ___ ,Compounding)` adds the optional argument `Compounding`.

Examples

Specify a Four-Period Tree with Annual Nodes

This example shows how to specify a four-period tree with annual nodes using annual compounding to report rates.

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

```
TimeSpec = struct with fields:
           FinObj: 'BKTimeSpec'
    ValuationDate: 731947
         Maturity: [4x1 double]
      Compounding: 1
            Basis: 0
     EndMonthRule: 1
```

Input Arguments

`ValuationDate` — Pricing date and first observation in the tree

datetime scalar | string scalar | date character vector

Pricing date and first observation in the tree, specified as a scalar datetime, string, or date character vector.

To support existing code, `bktimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Dates marking the cash flow dates of the tree
datetime array | string array | date character vector

Dates marking the cash flow dates of the tree, specified as `NLEVELS`-by-`1` vector using a datetime array, string array, or date character vectors. Cash flows with these maturities fall on tree nodes. `Maturity` should be in increasing order.

To support existing code, `bktimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Compounding` — Rate at which the input zero rates were compounded when annualized
`1` (default) | integer with value of `1`, `2`, `3`, `4`, `6`, `12`, `365`, or `-1`

(Optional) Rate at which the input zero rates were compounded when annualized, specified as a scalar integer value.

- If `Compounding` = `1`, `2`, `3`, `4`, `6`, `12`:

  `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units; for example, `T = F` is one year.
- If `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.
- If `Compounding` = `−1`:

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

Data Types: `double`

Output Arguments

`TimeSpec` — Specification for the time layout for `bktree`
structure

Specification for the time layout for `bktree`, returned as a structure. 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`(`end`).

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `bktimespec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bktree` | `bkvolspec` | `bksens`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Specifying the Time Structure (TimeSpec)” on page 2-70
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bktree

Build Black-Karasinski interest-rate tree

Syntax

```
BKTree = bktree(VolSpec,RateSpec,TimeSpec)
BKTree = bktree( ___ ,Name,Value)
```

Description

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

`BKTree = bktree( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Create a BKTree

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

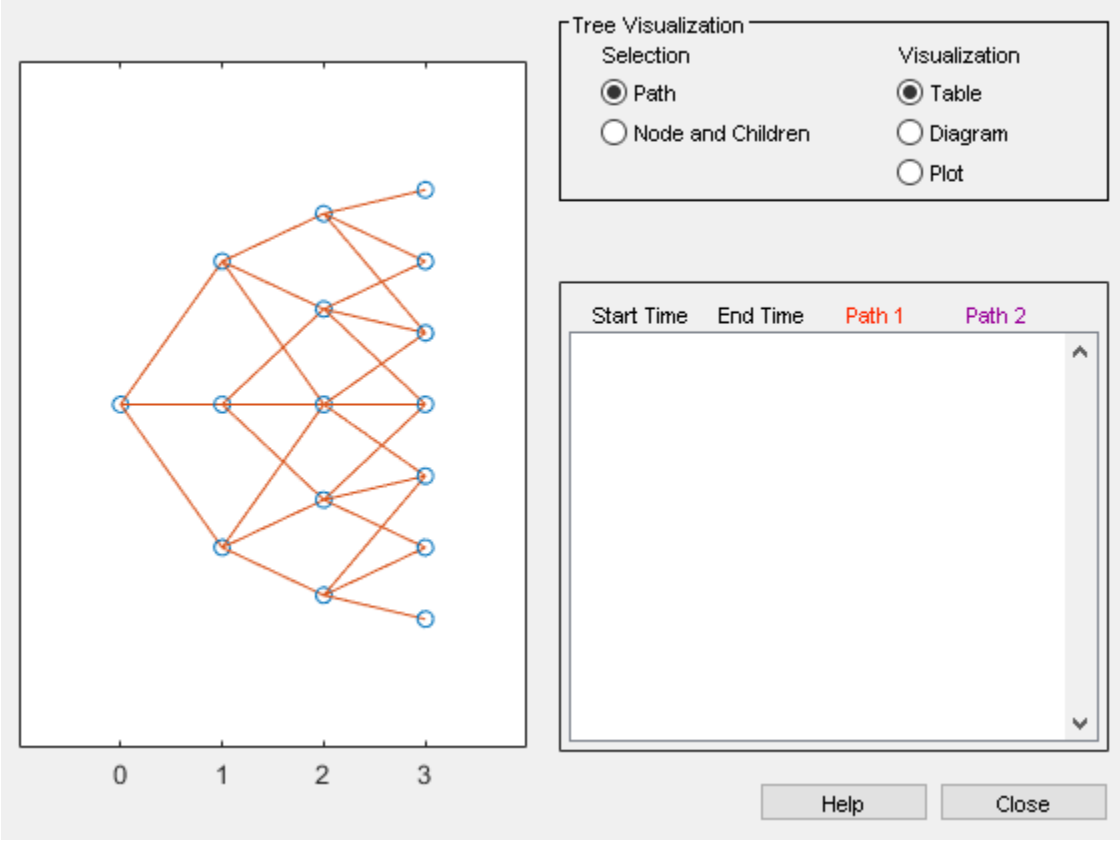
```
Compounding = -1;
ValuationDate = '01-01-2004';
StartDate = ValuationDate;
VolDates = ['12-31-2004'; '12-31-2005'; '12-31-2006';
'12-31-2007'];
VolCurve = 0.01;
AlphaDates = '01-01-2008';
AlphaCurve = 0.1;
Rates = [0.0275; 0.0312; 0.0363; 0.0415];

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 = struct with fields:
      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 3 4 5 6]}
 FwdTree: {1x4 cell}
```

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

```
treeviewer(BKTree)
```



Input Arguments

`VolSpec` — Volatility process specification

structure

Volatility process specification, specified using the `VolSpec` output obtained from `bdtvolspec`.

Data Types: `struct`

`RateSpec` — Interest-rate specification for initial risk-free rate curve

structure

Interest-rate specification for initial rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`TimeSpec` — Time tree layout specification
structure

Time tree layout specification, specified using the `TimeSpec` output obtained from `bdttimespec`. The TimeSpec defines the observation dates of the BK tree and the `Compounding` rule for date to time mapping and price-yield formulas.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BKTree = bktree(BKVolSpec, RateSpec, BKTimeSpec,'Method','HW1996')`

`Method` — Hull-White method upon which the tree-node connectivity algorithm is based
`'HW2000'` (default) | character vector with values `'HW2000'` or `'HW1996'`

Hull-White method upon which the tree-node connectivity algorithm is based, specified as a character vector with a value of `'HW2000'` or `'HW1996'`.

`bktree` supports two tree-node connectivity algorithms. `HW1996` is based on the original paper published in the *Journal of Derivatives*, and `HW2000` is the general version of the algorithm, as specified in the paper published in August 2000.

Data Types: `char`

Output Arguments

`BKTree` — Time and interest-rate information of a recombining tree
structure

Time and interest-rate information of a recombining tree, returned as a structure.

Version History

Introduced before R2006a

References

[1] Hull, J., and A. White. "Using Hull-White Interest Rate Trees." *Journal of Derivatives.* 1996.

[2] Hull, J., and A. White. "*The General Hull-White Model and Super Calibration*." August 2000.

See Also

`bkprice` | `bkvolspec` | `bktimespec` | `intenvset` | `bksens`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bkvolspec

Specify Black-Karasinski interest-rate volatility process

Syntax

```
VolSpec = bdtvolspec(ValuationDate,VolDates,VolCurve,AlphaDates,AlphaCurve)
VolSpec = bdtvolspec( ___ ,InterpMethod)
```

Description

`VolSpec = bdtvolspec(ValuationDate,VolDates,VolCurve,AlphaDates,AlphaCurve)` creates a structure specifying the volatility for `bktree`.

`VolSpec = bdtvolspec( ___ ,InterpMethod)` adds the optional argument `InterpMethod`.

Examples

Create a Black-Karasinski Volatility Specification

This example shows how to create a Black-Karasinski volatility specification (VolSpec) using the following data.

```
ValuationDate = datetime(2004,1,1);
StartDate = ValuationDate;
VolDates = [datetime(2004,12,31) ; datetime(2005,12,31) ; datetime(2006,12,31) ;
datetime(2007,12,31) ];
VolCurve = 0.01;
AlphaDates = datetime(2008,1,1);
AlphaCurve = 0.1;
BKVolSpec = bkvolspec(ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve)
```

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

Input Arguments

`ValuationDate` — Observation date of the investment horizon

datetime scalar | string scalar | date character vector

Observation date of the investment horizon, specified as a scalar datetime, string, or date character vector.

To support existing code, `bkvolspec` also accepts serial date numbers as inputs, but they are not recommended.

`VolDates` — Number of points of yield volatility end dates
datetime array | string array | date character vector

Number of points of yield volatility end dates, specified as a `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bkvolspec` also accepts serial date numbers as inputs, but they are not recommended.

`VolCurve` — Yield volatility values
decimal

Yield volatility values, specified as a `NPOINTS`-by-`1` vector of decimal values. The term structure of `VolCurve` is the yield volatility represented by the value of the volatility of the yield from time `t` = 0 to time `t` + *i*, where *i* is any point within the volatility curve.

Data Types: `double`

`AlphaDates` — Mean reversion end dates
datetime array | string array | date character vector

Mean reversion end dates, specified as a `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bkvolspec` also accepts serial date numbers as inputs, but they are not recommended.

`AlphaCurve` — Positive mean reversion values
positive decimal

Positive mean reversion values, specified as a `NPOINTS`-by-`1` vector of positive decimal values.

Data Types: `double`

`InterpMethod` — Interpolation method
`'linear'` (default) | character vector with values supported by `interp1`

(Optional) Interpolation method, specified as a character vector with values supported by `interp1`.

Data Types: `char`

Output Arguments

`VolSpec` — Specification for the volatility model for `bktree`
structure

Structure specifying the volatility model for `bktree`.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `bkvolspec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bktree` | `bkprice` | `bktimespec` | `interp1`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Specifying the Volatility Model (VolSpec)” on page 2-68
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bondbybdt

Price bond from Black-Derman-Toy interest-rate tree

Syntax

```
[Price,PriceTree] = bondbybdt(BDTTree,CouponRate,Settle,Maturity)
[Price,PriceTree] = bondbybdt(___,Name,Value)
```

Description

`[Price,PriceTree] = bondbybdt(BDTTree,CouponRate,Settle,Maturity)` prices bond from a Black-Derman-Toy interest-rate tree. `bondbybdt` computes prices of vanilla bonds, stepped coupon bonds and amortizing bonds.

`[Price,PriceTree] = bondbybdt(___,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Bond Using a BDT Tree

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

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

```
load deriv.mat;
```

Define the bond using the required arguments. Other arguments use defaults.

```
CouponRate = 0.10;
Settle = datetime(2000,1,1);
Maturity = datetime(2003,1,1);
Period = 1;
```

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

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

```
Price = 95.5030
```

Price a Stepped Coupon Bond

Price single stepped coupon bonds using market data.

Define the interest-rate term structure.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2010,1,1);
```

```
StartDates = ValuationDate;
EndDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
Compounding = 1;
```

Create the RateSpec.

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

```
RS = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
          EndTimes: [4x1 double]
        StartTimes: [4x1 double]
          EndDates: [4x1 double]
        StartDates: 734139
    ValuationDate: 734139
            Basis: 0
     EndMonthRule: 1
```

Create the stepped bond instrument.

```
Settle = datetime(2010,1,1);
Maturity = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
%CouponRate = {{'01-Jan-2012' .0425;'01-Jan-2014' .0750}};
CouponRate = {{datetime(2012,1,1) .0425; datetime(2014,1,1) .0750}};
Period = 1;
```

Build the BDT tree and assume the volatility to be 10% using the following market data:

```
Sigma = 0.1;
BDTTimeSpec = bdttimespec(ValuationDate, EndDates);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVolSpec, RS, BDTTimeSpec)
```

```
BDTT = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734139 734504 734869 735235]
        TFwd: {[4x1 double]  [3x1 double]  [2x1 double]  [3]}
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
     FwdTree: {[1.0350]  [1.0444 1.0543]  [1.0469 1.0573 1.0700]  [1.0505 ... ]}
```

Compute the price of the stepped coupon bonds.

```
PBDT= bondbybdt(BDTT, CouponRate, Settle,Maturity , Period)
```

```
PBDT = 4×1

  100.7246
  100.0945
  101.5900
```

```
  102.0820
```

Price Two Bonds with Amortization Schedules

Price two bonds with amortization schedules using the `Face` input argument to define the schedule.

Define the interest-rate term structure.

```
Rates = 0.035;
ValuationDate = datetime(2011,11,1);
StartDates = ValuationDate;
EndDates = datetime(2017,11,1);
Compounding = 1;
```

Create the `RateSpec`.

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

Create the bond instrument. The bonds have a coupon rate of 4% and 3.85%, a period of one year, and mature on 1-Nov-2017.

```
CouponRate = [0.04; 0.0385];
Settle = datetime(2011,11,1);
Maturity = datetime(2017,11,1);
Period = 1;
```

Define the amortizing schedule.

```
Face = {{datetime(2015,11,1) 100; datetime(2016,11,1) 85; datetime(2017,11,1) 70};
{datetime(2015,11,1) 100; datetime(2016,11,1) 90; datetime(2017,11,1) 80}};
```

Build the BDT tree and assume the volatility to be 10%.

```
MatDates = [datetime(2012,11,1) ; datetime(2013,11,1) ; datetime(2014,11,1) ; datetime(2015,11,1
BDTTimeSpec = bdttimespec(ValuationDate, MatDates);
Volatility = 0.1;
BDTVolSpec = bdtvolspec(ValuationDate, MatDates, Volatility*ones(1,length(MatDates))');
BDTT = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Compute the price of the amortizing bonds.

```
Price = bondbybdt(BDTT, CouponRate, Settle, Maturity, 'Period',Period,...
'Face', Face)
```

```
Price = 2×1

  102.4791
  101.7786
```

Input Arguments

BDTTree — Interest-rate structure
structure

Interest-rate tree structure, created by `bdttree`

Data Types: `struct`

CouponRate — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

Settle — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbybdt` also accepts serial date numbers as inputs, but they are not recommended.

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

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each bond.

To support existing code, `bondbybdt` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = bondbybdt(BDTTree,CouponRate,Settle,Maturity,'Period',4,'Face',10000)`

Period — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and an `NINST`-by-`1` vector. Values for `Period` are `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbybdt` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbybdt` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
`Settle` date (default) | datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbybdt` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector of nonnegative face values or an `NINST`-by-`1` cell array of face values or face value schedules. For the latter case, each element of the cell array is a `NumDates`-by-`2` cell array, where the first column is dates and the second column is its associated face value. The date indicates the last day that the face value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-`2` if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

Output Arguments

`Price` — Expected bond prices at time 0
vector

Expected bond prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.

More About

Vanilla Bond

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment.

Stepped Coupon Bond

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond.

Bond with an Amortization Schedule

An amortized bond is treated as an asset, with the discount amount being amortized to interest expense over the life of the bond.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `bondbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdttree` | `bdtprice` | `cfamounts` | `instbond`

Topics

“Computing Instrument Sensitivities” on page 2-89
“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Bond” on page 2-3
“Understanding the Interest-Rate Term Structure” on page 2-48
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

blackvolbyrebonato

Compute Black volatility for LIBOR Market Model using Rebonato formula

Syntax

```
outVol = blackvolbyrebonato(ZeroCurve,VolFunc,CorrMat,ExerciseDate,Maturity)
outVol = blackvolbyrebonato( ___ ,Name,Value)
```

Description

`outVol = blackvolbyrebonato(ZeroCurve,VolFunc,CorrMat,ExerciseDate,Maturity)` computes the Black volatility for a swaption using a LIBOR Market Model.

`outVol = blackvolbyrebonato( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price Swaption for LIBOR Market Model Using the Rebonato Formula

Define the input maturity and tenor for a LIBOR Market Model (LMM) specified by the cell array of volatility function handles, and a correlation matrix for the LMM.

```
Settle = datetime(2004,8,11);

% Zero Curve
CurveTimes = (1:10)';
CurveDates = daysadd(Settle,360*CurveTimes,1);

ZeroRates = [0.03 0.033 0.036 0.038 0.04 0.042 0.043 0.044 0.045 0.046]';

% Construct an IRCurve
irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

LMMVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
LMMVolParams = [.3 -.02 .7 .14];

numRates = length(ZeroRates);
VolFunc(1:numRates-1) = {@(t) LMMVolFunc(LMMVolParams,t)};

Beta = .08;
CorrFunc = @(i,j,Beta) exp(-Beta*abs(i-j));
CorrMat = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),Beta);

ExerciseDate = datetime(2009,8,11);
Maturity = daysadd(ExerciseDate,360*[3;4],1);

Vol = blackvolbyrebonato(irdc,VolFunc,CorrMat,ExerciseDate,Maturity,'Period',1)
```

```
Vol = 2×1

    0.2210
```

```
    0.2079
```

Input Arguments

`ZeroCurve` — Zero-curve for `LiborMarketModel` model

structure

Zero-curve for the `LiborMarketModel`, specified using `IRDataCurve` or `RateSpec`.

Data Types: `struct`

`VolFunc` — Function handle for volatility

cell array of function handles

Function handle for volatility, specified by a `NumRates`-by-`1` cell array of function handles. Each function handle must take time as an input and return a scalar volatility

Data Types: `cell` | `function_handle`

`CorrMat` — Correlation matrix

vector

Correlation matrix, specified by `NumRates`-by-`NumRates`.

Data Types: `double`

`ExerciseDate` — Swaption exercise date

datetime array | string array | date character vector

Swaption exercise dates, specified by a `NumSwaptions`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `blackvolbyrebonato` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Swap maturity date

datetime array | string array | date character vector

Swap maturity dates, specified using a `NumSwaptions`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `blackvolbyrebonato` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Vol = blackvolbyrebonato(irdc,VolFunc,CorrMat,ExerciseDate,Maturity,'Period',1)`

`Period` — Compounding frequency of curve and reset of swaptions
2 (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Compounding frequency of curve and reset of swaptions, specified as the comma-separated pair consisting of `'Period'` and a positive integer for the values `1,2,4,6,12` in a `NumSwaptions`-by-`1` vector.

Data Types: `single` | `double`

Output Arguments

`outVol` — Black volatility for specified swaption
scalar | vector

Black volatility, returned as a vector for the specified swaptions.

Algorithms

The Rebonato approximation formula relates the Black volatility for a European swaption, given a set of volatility functions and a correlation matrix

$$(v_{\alpha,\beta}^{LFM})^2 = \sum_{i,\,j\,=\,\alpha+1}^{\beta} \frac{w_i(0)w_j(0)F_i(0)F_j(0)\rho_{i,\,j}}{S_{\alpha,\beta}(0)^2} \int_0^{T_\alpha} \sigma_i(t)\sigma_j(t)dt$$

where:

$$w_i(t) = \frac{\tau_i P(t,T_i)}{\sum_{k\,=\,\alpha+1}^{\beta} \tau_K P(t,t_K)}$$

Version History

Introduced in R2013a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `blackvolbyrebonato` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

See Also

`LiborMarketModel`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Supported Interest-Rate Instrument Functions” on page 2-3

blackvolbysabr

Calculate implied Black volatility using SABR model

Syntax

```
outVol = blackvolbysabr(Alpha,Beta,Rho,Nu,Settle,ExerciseDate,ForwardValue,
Strike)
outVol = blackvolbysabr( ___ ,Name,Value)
```

Description

`outVol = blackvolbysabr(Alpha,Beta,Rho,Nu,Settle,ExerciseDate,ForwardValue, Strike)` calculates the implied Black volatility using the SABR stochastic volatility model.

`outVol = blackvolbysabr( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Implied Black Volatility Using the SABR Model

Define the model parameters and option data.

```
ForwardRate = 0.0357;
Strike = 0.03;
Alpha = 0.036;
Beta = 0.5;
Rho = -0.25;
Nu = 0.35;

Settle = datetime(2013,9,15);
ExerciseDate = datetime(2015,9,15);
```

Compute the Black volatility using the SABR model.

```
ComputedVols = blackvolbysabr(Alpha, Beta, Rho, Nu, Settle, ...
ExerciseDate, ForwardRate, Strike)
```

```
ComputedVols = 0.2122
```

Compute the Shifted Black Volatility Using the Shifted SABR Model

Define the model parameters and option data with a negative strike.

```
ForwardRate = 0.0002;
Strike = -0.001;  % -0.1% strike.
Alpha = 0.01;
Beta = 0.5;
Rho = -0.1;
```

```
Nu = 0.15;
Shift = 0.005;  % 0.5 percent shift

Settle = datetime(2016,3,1);
ExerciseDate = datetime(2017,3,1);
```

Compute the Shifted Black volatility using the Shifted SABR model.

```
ComputedVols = blackvolbysabr(Alpha, Beta, Rho, Nu, Settle, ...
ExerciseDate, ForwardRate, Strike, 'Shift', Shift)
```

```
ComputedVols = 0.1518
```

Input Arguments

`Alpha` — Current SABR volatility

scalar

Current SABR volatility, specified as a scalar.

Data Types: `double`

`Beta` — SABR constant elasticity of variance (CEV) exponent

scalar

SABR CEV exponent, specified as a scalar.

Data Types: `double`

`Rho` — Correlation between forward value and volatility

scalar

Correlation between forward value and volatility, specified as a scalar.

Data Types: `double`

`Nu` — Volatility of volatility

scalar

Volatility of volatility, specified as a scalar.

Data Types: `double`

`Settle` — Settlement date

scalar for serial nonnegative date number | scalar for date character vector

Settlement date, specified as a scalar using a serial nonnegative date number or date character vector.

Data Types: `double` | `char`

`ExerciseDate` — Option exercise date

scalar for serial nonnegative date number | scalar for date character vector

Option exercise date, specified as a scalar using a serial nonnegative date number or date character vector.

Data Types: `double` | `char`

`ForwardValue` — Current forward value of underlying asset
scalar | vector

Current forward value of the underlying asset, specified as a scalar or vector of size `NumVols`-by-`1`.

Data Types: `double`

`Strike` — Option strike price values
scalar | vector

Option strike price values, specified as a scalar value or a vector of size `NumVols`-by-`1`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `outVol = blackvolbysabr(Alpha,Beta,Rho,Nu,Settle,ExerciseDate,ForwardValue,Strike,'Basis',2,'Model','Obloj2008')`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer of the set `[1...13]`.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Model` — Version of SABR model
`'Hagan2002'` (default) | value `'Obloj2008'`

Version of SABR model, specified as the comma-separated pair consisting of `'Model'` and one of the following values:

- `'Hagan2002'` — Original version by Hagan et al. (2002)
- `'Obloj2008'` — Version by Obloj (2008)

Data Types: `char`

`Shift` — Shift in decimals for shifted SABR model
`0` (no shift) (default) | positive decimal

Shift in decimals for the shifted SABR model (to be used with the Shifted Black model), specified as the comma-separated pair consisting of `'Shift'` and a scalar positive decimal value. Set this parameter to a positive shift in decimals to add a positive shift to `ForwardValue` and `Strike`, which effectively sets a negative lower bound for `ForwardValue` and `Strike`. For example, a `Shift` value of 0.01 is equal to a 1% shift.

Data Types: `double`

Output Arguments

`outVol` — Implied Black volatility computed by SABR model
scalar | vector

Implied Black volatility computed by SABR model, returned as a scalar or vector of size `NumVols`-by-`1`.

Algorithms

The SABR stochastic volatility model treats the underlying forward $\widehat{F}$ and volatility $\widehat{\alpha}$ as separate random processes, which are related with correlation ρ:

$$d\widehat{F} = \widehat{\alpha}\widehat{F}^{\beta}dW_1$$
$$d\widehat{\alpha} = \nu\widehat{\alpha}dW_2$$
$$dW_1dW_2 = \rho dt$$
$$\widehat{F}(0) = F$$
$$\widehat{\alpha}(0) = \alpha$$

where

- $\widehat{F}$ is the underlying forward (a variable).
- F is the current underlying forward (a constant).
- $\widehat{\alpha}$ is the SABR volatility (a variable).
- α is the current SABR volatility (a constant).
- β is the SABR constant elasticity of variance (CEV) exponent.

- υ is the volatility of volatility.
- dW_1 is Brownian motion.
- dW_2 is Brownian motion.
- ρ is the correlation between forward value and volatility.

In contrast, Black's lognormal model assumes a constant volatility, σ_B.

$$d\widehat{F} = \sigma_B \widehat{F} dW$$

Hagan et al. (2002) derived the following closed-form approximation of implied Black lognormal volatility (σ_B) for the SABR model

$$\sigma_B(F,K) = \frac{\alpha\left\{1 + \left[\frac{(1-\beta)^2}{24}\frac{\alpha^2}{(FK)^{1-\beta}} + \frac{1}{4}\frac{\rho\beta\upsilon\alpha}{(FK)^{(1-\beta)/2}} + \frac{2-3\rho^2}{24}\upsilon^2\right]T + \ldots\right\}}{(FK)^{(1-\beta)/2}\left\{1 + \frac{(1-\beta)^2}{24}\log^2(F/K) + \frac{(1-\beta)^4}{1920}\log^4(F/K) + \ldots\right\}}\left(\frac{z}{x(z)}\right)$$

$$z = \frac{\upsilon}{\alpha}({}^{F}\log(F/K)$$

$$x(z) = \log\left\{\frac{\sqrt{1-2\rho z + z^2} + z - \rho}{1-\rho}\right\}$$

where

- F is the current forward value of the underlying.
- α is the current SABR volatility.
- K is the strike value.
- T is the time to option maturity.

Obloj (2008) advocated the following closed-form approximation of implied Black lognormal volatility for the SABR model (for $\beta < 1$)

$$\sigma_B(F,K) = \frac{\upsilon\log(F/K)}{x(z)}\left\{1 + \left[\frac{(1-\beta)^2}{24}\frac{\alpha^2}{(FK)^{1-\beta}} + \frac{1}{4}\frac{\rho\beta\upsilon\alpha}{(FK)^{(1-\beta)/2}} + \frac{2-3\rho^2}{24}\upsilon^2\right]T + \ldots\right\}$$

$$z = \frac{\upsilon}{\alpha}\frac{F^{(1-\beta)} - K^{(1-\beta)}}{1-\beta}$$

$$x(z) = \log\left\{\frac{\sqrt{1-2\rho z + z^2} + z - \rho}{1-\rho}\right\}$$

These expressions can be simplified in special situations, such as the at-the-money ($F = K$) and stochastic lognormal ($\beta = 1$) cases [1,2].

Version History

Introduced in R2014a

References

[1] Hagan, P. S., D. Kumar, A.S. Lesniewski, and D.E. Woodward. *"Managing Smile Risk."* Wilmott Magazine, September, pp. 84–108, 2002.

[2] Obloj, J. *"Fine-tune your smile: Correction to Hagan et. al."* Wilmott Magazine, 2008.

See Also

`swaptionbyblk` | `swaptionbynormal` | `optsensbysabr`

Topics

"Calibrate the SABR Model" on page 2-33
"Price a Swaption Using the SABR Model" on page 2-38
"Price Swaptions with Negative Strikes Using the Shifted SABR Model" on page 2-26
"Work with Negative Interest Rates Using Functions" on page 2-18
"Supported Interest-Rate Instrument Functions" on page 2-3

External Websites

How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

bondbybk

Price bond from Black-Karasinski interest-rate tree

Syntax

```
[Price,PriceTree] = bondbybk(BKTree,CouponRate,Settle,Maturity)
[Price,PriceTree] = bondbybk( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = bondbybk(BKTree,CouponRate,Settle,Maturity)` prices bond from a Black-Karasinski interest-rate tree. `bondbybk` computes prices of vanilla bonds, stepped coupon bonds and amortizing bonds.

`[Price,PriceTree] = bondbybk( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Bond Using a BK Tree

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

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

```
load deriv.mat;
```

Define the bond using the required arguments. Other arguments use defaults.

```
CouponRate = 0.04;
Settle = datetime(2004,1,1);
Maturity = datetime(2008,1,1);
```

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

```
Period = 1;
Price = bondbybk(BKTree, CouponRate, Settle, Maturity, Period)
```

```
Price = 99.3296
```

Price a Stepped Coupon Bond

Price single stepped coupon bonds using market data.

Define the interest-rate term structure.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2010,1,1);
```

```
StartDates = ValuationDate;
EndDates = [datetime(2011,1,1) ; datetime(2012,1,1) ;...
datetime(2013,1,1) ; datetime(2014,1,1)];
Compounding = 1;
```

Create the RateSpec.

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

```
RS = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
          EndTimes: [4x1 double]
        StartTimes: [4x1 double]
          EndDates: [4x1 double]
        StartDates: 734139
    ValuationDate: 734139
            Basis: 0
     EndMonthRule: 1
```

Create the stepped bond instrument.

```
Settle = datetime(2010,1,1);
Maturity = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
CouponRate = {{datetime(2012,1,1) .0425;datetime(2014,1,1) .0750}};
Period = 1;
```

Build the BK tree using the following market data:

```
VolDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ];
VolCurve = 0.01;
AlphaDates = datetime(2014,1,1);
AlphaCurve = 0.1;

BKVolSpec = bkvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RS.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RS, BKTimeSpec);
```

Compute the price of the stepped coupon bonds.

```
PBK= bondbybk(BKT, CouponRate, Settle,Maturity , Period)
```

```
PBK = 4×1

  100.7246
  100.0945
  101.5900
  102.0820
```

Price a Bond with an Amortization Schedule

Price a bond with an amortization schedule using the `Face` input argument to define the schedule.

Define the interest-rate term structure.

```
Rates = 0.065;
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates=  datetime(2017,1,1);
Compounding = 1;
```

Create the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.6853
            Rates: 0.0650
         EndTimes: 6
       StartTimes: 0
         EndDates: 736696
       StartDates: 734504
    ValuationDate: 734504
            Basis: 0
     EndMonthRule: 1
```

Create the bond instrument. The bond has a coupon rate of 7%, a period of one year, and matures on 1-Jan-2017.

```
CouponRate = 0.07;
Settle = datetime(2011,1,1);
Maturity = datetime(2017,1,1);
Period = 1;
Face = {{datetime(2015,1,1) 100; datetime(2016,1,1) 90; datetime(2017,1,1) 80}};
```

Build the BK tree with the following market data:

```
VolDates = [datetime(2012,1,1) ; datetime(2013,1,1) ;...
datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1) ; datetime(2017,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2017,1,1);
AlphaCurve = 0.1;

BKVolSpec = bkvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RateSpec.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RateSpec, BKTimeSpec);
```

Compute the price of the amortizing bond.

```
Price = bondbybk(BKT, CouponRate, Settle, Maturity, 'Period', Period,...
'Face', Face)
```

```
Price = 102.3155
```

Compare the results with price of a vanilla bond.

```
PriceVanilla = bondbybk(BKT, CouponRate, Settle, Maturity, Period)
```

```
PriceVanilla = 102.4205
```

Input Arguments

`BKTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `bktree`

Data Types: `struct`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbybk` also accepts serial date numbers as inputs, but they are not recommended.

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

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each bond.

To support existing code, `bondbybk` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = bondbybk(BKTree,CouponRate,Settle,Maturity,'Period',4,'Face',10000)`

`Period` — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector. Values for `Period` are `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer `[0, 1]` using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbybk` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbybk` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbybk` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
`Settle` date (default) | datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbybk` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector of nonnegative face values or an `NINST`-by-`1` cell array of face values or face value schedules. For the latter case, each element of the cell array is a `NumDates`-by-`2` cell array, where the first column is dates and the second column is its associated face value. The date indicates the last day that the face value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-`2` if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

Output Arguments

`Price` — Expected bond prices at time 0
vector

Expected bond prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

Vanilla Bond

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment.

Stepped Coupon Bond

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond.

Bond with an Amortization Schedule

An amortized bond is treated as an asset, with the discount amount being amortized to interest expense over the life of the bond.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `bondbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bkprice` | `bktree` | `cfamounts` | `hwprice` | `hwtree` | `instbond`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Bond” on page 2-3
“Understanding the Interest-Rate Term Structure” on page 2-48
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bondbyhjm

Price bond from Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree] = bondbyhjm(HJMTree,CouponRate,Settle,Maturity)
[Price,PriceTree] = bondbyhjm(___,Name,Value)
```

Description

`[Price,PriceTree] = bondbyhjm(HJMTree,CouponRate,Settle,Maturity)` prices bond from a Heath-Jarrow-Morton interest-rate tree. `bondbyhjm` computes prices of vanilla bonds, stepped coupon bonds and amortizing bonds.

`[Price,PriceTree] = bondbyhjm(___,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Bond Using an HJM Tree

Price a 4% bond using an HJM interest-rate tree.

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

```
load deriv.mat;
```

Define the bond using the required arguments. Other arguments use defaults.

```
CouponRate = 0.04;
Settle = datetime(2000,1,1);
Maturity = datetime(2004,1,1);
```

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

```
Period = 1;
Price = bondbyhjm(HJMTree, CouponRate, Settle, Maturity, Period)
```

```
Price = 97.3600
```

Price a Stepped Coupon Bond

Price single stepped coupon bonds using market data.

Define the interest-rate term structure.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2010,1,1);
```

```
StartDates = ValuationDate;
EndDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
Compounding = 1;
```

Create the RateSpec.

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

Create the stepped bond instrument.

```
Settle = datetime(2010,1,1);
Maturity = [ datetime(2011,1,1); datetime(2012,1,1); datetime(2013,1,1) ; datetime(2014,1,1)];
CouponRate = {{datetime(2012,1,1) .0425;datetime(2014,1,1) .0750}};
Period = 1;
```

Build the HJM tree using the following market data:

```
Volatility = [.2; .19; .18; .17];
CurveTerm = [ 1;  2;   3;   4];
HJMTimeSpec = hjmtimespec(ValuationDate, EndDates);
HJMVolSpec = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
HJMT = hjmtree(HJMVolSpec,RS,HJMTimeSpec);
```

Compute the price of the stepped coupon bonds.

```
PHJM= bondbyhjm(HJMT, CouponRate, Settle,Maturity , Period)
```

```
PHJM = 4×1

  100.7246
  100.0945
  101.5900
  102.0820
```

Price a Bond with an Amortization Schedule

Price a bond with an amortization schedule using the Face input argument to define the schedule.

Define the interest-rate term structure.

```
Rates = 0.065;
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates=  datetime(2017,1,1);
Compounding = 1;
```

Create the RateSpec.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
```

```
            Disc: 0.6853
           Rates: 0.0650
       EndTimes: 6
     StartTimes: 0
       EndDates: 736696
     StartDates: 734504
  ValuationDate: 734504
           Basis: 0
    EndMonthRule: 1
```

Create the bond instrument. The bond has a coupon rate of 7%, a period of one year, and matures on 1-Jan-2017.

```
CouponRate = 0.07;
Settle = datetime(2011,1,1);
Maturity = datetime(2017,1,1);
Period = 1;
Face = {{datetime(2015,1,1) 100;datetime(2016,1,1) 90;datetime(2017,1,1) 80}};
```

Build the HJM tree using the following market data:

```
Volatility = [.2; .19; .18; .17];
CurveTerm = [ 1;  2;   3;   4];
MaTree = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1);...
datetime(2016,1,1) ; datetime(2017,1,1)];
HJMTimeSpec = hjmtimespec(ValuationDate, MaTree);
HJMVolSpec = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
HJMT = hjmtree(HJMVolSpec,RateSpec,HJMTimeSpec);
```

Compute the price of the amortizing bond.

```
Price = bondbyhjm(HJMT, CouponRate, Settle, Maturity, 'Period',...
Period, 'Face' , Face)
```

```
Price = 102.3155
```

Compare the results with price of a vanilla bond.

```
PriceVanilla = bondbyhjm(HJMT, CouponRate, Settle, Maturity, Period)
```

```
PriceVanilla = 102.4205
```

Input Arguments

`HJMTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `hjmtree`

Data Types: `struct`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates

and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

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

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each bond.

To support existing code, `bondbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = bondbyhjm(HJMTree,CouponRate,Settle,Maturity,'Period',4,'Face',10000)`

`Period` — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-1 vector. Values for `Period` are `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-1 vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

LastCouponDate — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

StartDate — Forward starting date of payments
`Settle` date (default) | datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

Face — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector of nonnegative face values or an `NINST`-by-`1` cell array of face values or face value schedules. For the latter case, each element of the cell array is a `NumDates`-by-`2` cell array, where the first column is dates and the second column is its associated face value. The date indicates the last day that the face value is valid.

Data Types: `cell` | `double`

Options — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

AdjustCashFlowsBasis — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

BusinessDayConvention — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-`2` if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

Output Arguments

`Price` — Expected bond prices at time 0
vector

Expected bond prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.PBush` contains the clean prices.
- `PriceTree.AIBush` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.

More About

Vanilla Bond

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment.

Stepped Coupon Bond

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond.

Bond with an Amortization Schedule

An amortized bond is treated as an asset, with the discount amount being amortized to interest expense over the life of the bond.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `bondbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmtree` | `cfamounts` | `hjmprice` | `instbond`

Topics

“Computing Instrument Prices” on page 2-81
“Bond” on page 2-3
“Understanding the Interest-Rate Term Structure” on page 2-48
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

bondbyhw

Price bond from Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = bondbyhw(HWTree,CouponRate,Settle,Maturity)
[Price,PriceTree] = bondbyhw(___,Name,Value)
```

Description

`[Price,PriceTree] = bondbyhw(HWTree,CouponRate,Settle,Maturity)` prices bond from a Hull-White interest-rate tree. `bondbyhw` computes prices of vanilla bonds, stepped coupon bonds and amortizing bonds.

`[Price,PriceTree] = bondbyhw(___,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Bond Using the HW Tree

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

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

```
load deriv.mat;
```

Define the bond using the required arguments. Other arguments use defaults.

```
CouponRate = 0.04;
Settle = datetime(2004,1,1);
Maturity = datetime(2006,1,1);
```

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

```
Period = 1;
Price = bondbyhw(HWTree, CouponRate, Settle, Maturity, Period)
```

```
Price = 101.6002
```

Price a Stepped Coupon Bond

Price single stepped coupon bonds using market data.

Define the interest-rate term structure.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2010,1,1);
```

```
StartDates = ValuationDate;
EndDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
Compounding = 1;
```

Create the RateSpec.

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

```
RS = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
          EndTimes: [4x1 double]
        StartTimes: [4x1 double]
          EndDates: [4x1 double]
        StartDates: 734139
    ValuationDate: 734139
            Basis: 0
     EndMonthRule: 1
```

Create the stepped bond instrument.

```
Settle = datetime(2010,1,1);
Maturity = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
CouponRate = {{datetime(2012,1,1) .0425;datetime(2014,1,1) .0750}};
Period = 1;
```

Build the HW tree using the following market data:

```
VolDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2014,1,1);
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RS.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RS, HWTimeSpec);
```

Compute the price of the stepped coupon bonds.

```
PHW= bondbyhw(HWT, CouponRate, Settle,Maturity , Period)
```

```
PHW = 4×1

  100.7246
  100.0945
  101.5900
  102.0820
```

Price Two Bonds with Amortization Schedules

Price two bonds with amortization schedules using the Face input argument to define the schedules.

Define the interest rate term structure.

```
Rates = 0.035;
ValuationDate = datetime(2011,11,1);
StartDates = ValuationDate;
EndDates = datetime(2017,11,1);
Compounding = 1;
```

Create the `RateSpec`.

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

Create the bond instrument. The bonds have a coupon rate of 4% and 3.85%, a period of one year, and mature on 1-Nov-2017.

```
CouponRate = [0.04; 0.0385];
Settle = datetime(2011,11,1);
Maturity = datetime(2017,11,1);
Period = 1;
```

Define the amortizing schedule.

```
Face = {{datetime(2015,11,1) 100;datetime(2016,11,1) 85;datetime(2017,11,1) 70};
{datetime(2015,11,1) 100;datetime(2016,11,1) 90;datetime(2017,11,1) 80}};
```

Build the HW tree and assume the volatility to be 10%.

```
VolDates = [datetime(2012,11,1) ; datetime(2013,11,1) ; datetime(2014,11,1) ; datetime(2015,11,1
VolCurve = 0.1;
AlphaDates = datetime(2017,1,1);
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RateSpec, HWTimeSpec);
```

Compute the price of the amortizing bonds.

```
Price = bondbyhw(HWT, CouponRate, Settle, Maturity, 'Period',Period,...
'Face', Face)
```

```
Price = 2×1

  102.4791
  101.7786
```

Input Arguments

`HWTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `hwtree`.

Data Types: `struct`

CouponRate — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an NINST-by-1 decimal annual rate or NINST-by-1 cell array, where each element is a NumDates-by-2 cell array. The first column of the NumDates-by-2 cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: double | cell

Settle — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, bondbyhw also accepts serial date numbers as inputs, but they are not recommended.

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

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each bond.

To support existing code, bondbyhw also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree] =
bondbyhw(HWTree,CouponRate,Settle,Maturity,'Period',4,'Face',10000)

Period — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of 'Period' and a NINST-by-1 vector. Values for Period are 1, 2, 3, 4, 6, and 12.

Data Types: double

Basis — Day-count basis
0 (actual/actual) (default) | integer from 0 to 13

Day-count basis of the instrument, specified as the comma-separated pair consisting of 'Basis' and a NINST-by-1 vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbyhw` also accepts serial date numbers as inputs, but they are not recommended.

When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

LastCouponDate — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, bondbyhw also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

StartDate — Forward starting date of payments
Settle date (default) | datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of 'StartDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, bondbyhw also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify StartDate, the effective start date is the Settle date.

Face — Face value
100 (default) | nonnegative value | cell array of nonnegative values

Face or par value, specified as the comma-separated pair consisting of 'Face' and a NINST-by-1 vector of nonnegative face values or an NINST-by-1 cell array of face values or face value schedules. For the latter case, each element of the cell array is a NumDates-by-2 cell array, where the first column is dates and the second column is its associated face value. The date indicates the last day that the face value is valid.

Data Types: cell | double

Options — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of 'Options' and a structure that is created with derivset.

Data Types: struct

AdjustCashFlowsBasis — Flag to adjust cash flows based on actual period day count
false (default) | value of 0 (false) or 1 (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of 'AdjustCashFlowsBasis' and a NINST-by-1 vector of logicals with values of 0 (false) or 1 (true).

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-`2` if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

Output Arguments

`Price` — Expected bond prices at time 0
vector

Expected bond prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.

- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

Vanilla Bond

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment.

Stepped Coupon Bond

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond.

Bond with an Amortization Schedule

An amortized bond is treated as an asset, with the discount amount being amortized to interest expense over the life of the bond.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `bondbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bkprice` | `bktree` | `cfamounts` | `hwprice` | `hwtree` | `instbond`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Bond” on page 2-3
“Understanding the Interest-Rate Term Structure” on page 2-48
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

bondbycir

Price bond from Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = bondbycir(CIRTree,CouponRate,Settle,Maturity)
[Price,PriceTree] = bondbycir(___,Name,Value)
```

Description

`[Price,PriceTree] = bondbycir(CIRTree,CouponRate,Settle,Maturity)` prices bond from a Cox-Ingersoll-Ross (CIR) interest-rate tree. `bondbycir` computes prices of vanilla bonds, stepped coupon bonds, and amortizing bonds using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = bondbycir(___,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Bond Using a CIR Interest-Rate Tree

Define the `CouponRate` for a bond.

```
CouponRate = 0.035;
```

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = datetime(2017,1,1);
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',EndD
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = datetime(2017,1,1);
Maturity = datetime(2021,1,1);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
       FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
```

```
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [736696 737061 737426 737791]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Price the bond.

```
[Price,PriceTree] = bondbycir(CIRT,CouponRate,Settle,Maturity)
```

```
Price = 94.0880
```

```
PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
       tObs: [0 1 2 3 4]
       dObs: [736696 737061 737426 737791 738157]
      PTree: {1x5 cell}
     AITree: {[0]  [0 0 0]  [0 0 0 0 0]  [0 0 0 0 0 0 0]  [0 0 0 0 0 0 0]}
    Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Input Arguments

`CIRTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `cirtree`.

Data Types: `struct`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbycir` also accepts serial date numbers as inputs, but they are not recommended.

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

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each bond.

To support existing code, bondbycir also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree] = bondbycir(CIRTree,CouponRate,Settle,Maturity,'Period',4,'Face',10000)

Period — Coupons per year

2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of 'Period' and an NINST-by-1 vector. Values for Period are 1, 2, 3, 4, 6, and 12.

Data Types: double

Basis — Day-count basis

0 (actual/actual) (default) | integer from 0 to 13

Day-count basis of the instrument, specified as the comma-separated pair consisting of 'Basis' and a NINST-by-1 vector.

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

For more information, see “Basis” on page 2-228.

Data Types: double

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbycir` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbycir` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbycir` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
`Settle` date (default) | datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbycir` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector of nonnegative face values or a `NINST`-by-`1` cell array of face values or face value schedules. For the latter case, each element of the cell array is a `NumDates`-by-`2` cell array, where the first column is dates and the second column is its associated face value. The date indicates the last day that the face value is valid.

Data Types: `cell` | `double`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-`2` if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

Output Arguments

`Price` — Expected bond prices at time 0
vector

Expected bond prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.tObs` contains the observation times.
- `PriceTree.dObs` contains the observation dates.
- `PriceTree.PTree` contains the clean prices.
- `PriceTree.AITree` contains the accrued interest.

More About

Vanilla Bond

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment.

Stepped Coupon Bond

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond.

Bond with an Amortization Schedule

An amortized bond is treated as an asset, with the discount amount being amortized to interest expense over the life of the bond.

Version History

Introduced in R2018a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `bondbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J.,and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir` | `instbond`

Topics
“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Bond” on page 2-3
“Understanding the Interest-Rate Term Structure” on page 2-48
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

bondbyzero

Price bond from set of zero curves

Syntax

```
[Price,DirtyPrice,CFlowAmounts,CFlowDates] = bondbyzero(RateSpec,CouponRate,
Settle,Maturity)
[Price,DirtyPrice,CFlowAmounts,CFlowDates] = bondbyzero( ___ ,Name,Value)
```

Description

`[Price,DirtyPrice,CFlowAmounts,CFlowDates] = bondbyzero(RateSpec,CouponRate, Settle,Maturity)` prices a bond from a set of zero curves. `bondbyzero` computes prices of vanilla bonds, stepped coupon bonds and amortizing bonds.

Note Alternatively, you can use the `FixedBond` object to price fixed-rate bond instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,DirtyPrice,CFlowAmounts,CFlowDates] = bondbyzero( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Vanilla Bond

Price a 4% bond using a zero curve.

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

```
load deriv.mat;
CouponRate = 0.04;
Settle = datetime(2000,1,1);
Maturity = datetime(2004,1,1);
Price = bondbyzero(ZeroRateSpec, CouponRate, Settle, Maturity)
```

```
Price = 97.5334
```

Price a Stepped Coupon Bond

Price single stepped coupon bonds using market data.

Define data for the interest-rate term structure.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2010,1,1);
```

```
StartDates = ValuationDate;
EndDates = [datetime(2011,1,1) ; datetime(20102,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
Compounding = 1;
```

Create the RateSpec.

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

```
RS = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
          EndTimes: [4x1 double]
        StartTimes: [4x1 double]
          EndDates: [4x1 double]
        StartDates: 734139
    ValuationDate: 734139
            Basis: 0
     EndMonthRule: 1
```

Create the stepped bond instrument.

```
Settle = datetime(2010,1,1);
Maturity = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
CouponRate = {{datetime(2012,1,1) .0425;datetime(2014,1,1) .0750}};
Period = 1;
```

Compute the price of the stepped coupon bonds.

```
PZero= bondbyzero(RS, CouponRate, Settle, Maturity ,Period)
```

```
PZero = 4×1

  100.7246
  101.4247
  101.6442
  102.1362
```

Price a Bond with an Amortizing Schedule

Price a bond with an amortizing schedule using the Face input argument to define the schedule.

Define data for the interest-rate term structure.

```
Rates = 0.065;
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates=  datetime(2017,1,1);
Compounding = 1;
```

Create the RateSpec.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.6853
            Rates: 0.0650
         EndTimes: 6
       StartTimes: 0
         EndDates: 736696
       StartDates: 734504
    ValuationDate: 734504
            Basis: 0
     EndMonthRule: 1
```

Create and price the amortizing bond instrument. The bond has a coupon rate of 7%, a period of one year, and matures on 1-Jan-2017.

```
CouponRate = 0.07;
Settle = datetime(2011,1,1);
Maturity = datetime(2017,1,1);
Period = 1;
Face = {{datetime(2015,1,1) 100;datetime(2016,1,1) 90;datetime(2017,1,1) 80}};
Price = bondbyzero(RateSpec, CouponRate, Settle, Maturity, 'Period',...
Period, 'Face', Face)
```

```
Price = 102.3155
```

Compare the results with price of a vanilla bond.

```
PriceVanilla = bondbyzero(RateSpec, CouponRate, Settle, Maturity,Period)
```

```
PriceVanilla = 102.4205
```

Price both the amortizing and vanilla bonds.

```
Face = {{datetime(2015,1,1) 100;datetime(2016,1,1) 90;datetime(2017,1,1) 80};
         100};
PriceBonds = bondbyzero(RateSpec, CouponRate, Settle, Maturity, 'Period',...
               Period, 'Face', Face)
```

```
PriceBonds = 2×1

  102.3155
  102.4205
```

Price a Bond in a Holding Period

When a bond is first issued, it can be priced with `bondbyzero` on that day by setting the `Settle` date to the issue date. Later on, if the bond needs to be traded someday between the issue date and the maturity date, its new price can be computed by updating the `Settle` date, as well as the `RateSpec` input.

Note that the bond's price is determined by its remaining cash flows and the zero-rate term structure, which can both change as the bond matures. While `bondbyzero` automatically updates the bond's remaining cash flows with respect to the new `Settle` date, you must supply a new `RateSpec` input in order to reflect the new zero-rate term structure for that new `Settle` date.

Use the following Bond information.

```
IssueDate = datetime(2014,5,20);
CouponRate = 0.01;
Maturity = datetime(2019,5,20);
```

Determine the bond price on 20-May-2014.

```
Settle1 = datetime(2014,5,20);
ZeroDates1 = datemnth(Settle1,12*[1 2 3 5 7 10 20]');
ZeroRates1 = [0.23 0.63 1.01 1.60 2.01 2.27 2.79]'/100;
RateSpec1 = intenvset('StartDate',Settle1,'EndDates',ZeroDates1,'Rates',ZeroRates1);
[Price1, ~, CFlowAmounts1, CFlowDates1] = bondbyzero(RateSpec1, ...
    CouponRate, Settle1, Maturity, 'IssueDate', IssueDate);
Price1
```

```
Price1 = 97.1899
```

Determine the bond price on 10-Aug-2015.

```
Settle2 = datetime(2015,8,10);
ZeroDates2 = datemnth(Settle2,12*[1 2 3 5 7 10 20]');
ZeroRates2 = [0.40 0.73 1.09 1.62 1.98 2.24 2.58]'/100;
RateSpec2 = intenvset('StartDate',Settle2,'EndDates',ZeroDates2,'Rates',ZeroRates2);
[Price2, ~, CFlowAmounts2, CFlowDates2] = bondbyzero(RateSpec2, ...
    CouponRate, Settle2, Maturity, 'IssueDate', IssueDate);
Price2
```

```
Price2 = 98.9384
```

Price Three Bonds Using Two Different Curves

To price three bonds using two different curves, define the `RateSpec`:

```
StartDates = datetime(2016,4,1);
EndDates = [datetime(2017,4,1) ; datetime(2018,4,1) ; datetime(2019,4,1) ; datetime(2020,4,1)];
Rates = [[0.0356;0.041185;0.04489;0.047741],[0.0325;0.0423;0.0437;0.0465]];
RateSpec = intenvset('Rates', Rates, 'StartDates',StartDates,...
'EndDates', EndDates, 'Compounding', 1)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x2 double]
            Rates: [4x2 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
       StartDates: 736421
    ValuationDate: 736421
            Basis: 0
```

```
    EndMonthRule: 1
```

Price three bonds with the same `Maturity` and different coupons.

```
Settle = datetime(2016,4,1);
Maturity = datetime(2020,4,1);
Price = bondbyzero(RateSpec,[0.025;0.028;0.035],Settle,Maturity)
```

```
Price = 3×2

   92.0766   92.4888
   93.1680   93.5823
   95.7145   96.1338
```

Price a Vanilla Bond Using the Optional Input Argument `AdjustCashFlowsBasis`

To adjust the cash flows according to the accrual amount, use the optional input argument `AdjustCashFlowsBasis` when calling `bondbyzero`.

Use the following data to define the interest-rate term structure and to create a `RateSpec`.

```
Rates = 0.065;
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates=  datetime(2017,1,1);
Compounding = 1;
RateSpec = intenvset('ValuationDate',ValuationDate,'StartDates',StartDates,...
'EndDates', EndDates,'Rates',Rates,'Compounding',Compounding);
CouponRate = 0.07;
Settle = datetime(2011,1,1);
Maturity = datetime(2017,1,1);
Period = 1;
Face = {{datetime(2015,1,1) 100;datetime(2016,1,1) 90;datetime(2017,1,1) 80}};
```

Use `cfamounts` and cycle through the `Basis` of `0` to `13`, using the optional argument `AdjustCashFlowsBasis` to determine the cash flow amounts for accrued interest due at settlement.

```
AdjustCashFlowsBasis = true;
CFlowAmounts =  cfamounts(CouponRate,Settle,Maturity,'Period',Period,'Basis',0:13,'AdjustCashFlo
```

```
CFlowAmounts = 14×7

         0    7.0000    7.0000    7.0000    7.0000    7.0000  107.0000
         0    7.0000    7.0000    7.0000    7.0000    7.0000  107.0000
         0    7.0972    7.1167    7.0972    7.0972    7.0972  107.1167
         0    7.0000    7.0192    7.0000    7.0000    7.0000  107.0192
         0    7.0000    7.0000    7.0000    7.0000    7.0000  107.0000
         0    7.0000    7.0000    7.0000    7.0000    7.0000  107.0000
         0    7.0000    7.0000    7.0000    7.0000    7.0000  107.0000
         0    7.0000    7.0000    7.0000    7.0000    7.0000  107.0000
         0    7.0000    7.0000    7.0000    7.0000    7.0000  107.0000
         0    7.0972    7.1167    7.0972    7.0972    7.0972  107.1167
    ⋮
```

Notice that the cash flow amounts have been adjusted according to `Basis`.

Price a vanilla bond using the input argument `AdjustCashFlowsBasis`.

```
PriceVanilla = bondbyzero(RateSpec,CouponRate,Settle,Maturity,'Period',Period,'Basis',0:13,'Adjus
```

```
PriceVanilla = 14×1

  102.4205
  102.4205
  102.9216
  102.4506
  102.4205
  102.4205
  102.4205
  102.4205
  102.4205
  102.9216
      ⋮
```

Input Arguments

`RateSpec` — Interest-rate structure

structure

Interest-rate structure, specified using `intenvset` to create a `RateSpec` for an annualized zero rate term structure.

Data Types: `struct`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondbyzero` also accepts serial date numbers as inputs, but they are not recommended.

`Settle` must be earlier than `Maturity`.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each bond.

To support existing code, `bondbyzero` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = bondbyzero(RateSpec,CouponRate,Settle,Maturity,'Period',4,'Face',10000)`

`Period` — Coupons per year

`2` per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector. Values for `Period` are `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`Basis` — Day-count basis

`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

EndMonthRule — End-of-month rule flag for generating dates when Maturity is end-of-month date for month having 30 or fewer days
1 (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag for generating dates when Maturity is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of 'EndMonthRule' and a nonnegative integer [0, 1] using a NINST-by-1 vector.

- 0 = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- 1 = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: logical

IssueDate — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of 'IssueDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, bondbyzero also accepts serial date numbers as inputs, but they are not recommended.

FirstCouponDate — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, bondbyzero also accepts serial date numbers as inputs, but they are not recommended.

LastCouponDate — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, bondbyzero also accepts serial date numbers as inputs, but they are not recommended.

StartDate — Forward starting date of payments
Settle date (default) | datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of 'StartDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, bondbyzero also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify StartDate, the effective start date is the Settle date.

Face — Face value
100 (default) | scalar of nonnegative value | cell array of nonnegative values

Face value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` scalar of nonnegative face values or an `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is the associated face value. The date indicates the last day that the face value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-`2` if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

Output Arguments

`Price` — Fixed-rate note prices
matrix

Floating-rate note prices, returned as a (`NINST`) by number of curves (`NUMCURVES`) matrix. Each column arises from one of the zero curves.

`DirtyPrice` — Dirty bond price
matrix

Dirty bond price (clean + accrued interest), returned as a `NINST`- by-`NUMCURVES` matrix. Each column arises from one of the zero curves.

`CFlowAmounts` — Cash flow amounts
matrix

Cash flow amounts, returned as a `NINST`- by-`NUMCFS` matrix of cash flows for each bond.

`CFlowDates` — Cash flow dates
matrix

Cash flow dates, returned as a `NINST`- by-`NUMCFS` matrix of payment dates for each bond.

More About

Vanilla Bond

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment.

Stepped Coupon Bond

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond.

Bond with an Amortization Schedule

An amortized bond is treated as an asset, with the discount amount being amortized to interest expense over the life of the bond.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `bondbyzero` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`swapbyzero` | `cfamounts` | `cfbyzero` | `fixedbyzero` | `floatbyzero` | `FixedBond`

Topics

“Pricing Using Interest-Rate Term Structure” on page 2-61
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond Portfolio Optimization Using Portfolio Object”
“Bond” on page 2-3
“Understanding the Interest-Rate Term Structure” on page 2-48
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

bushpath

Extract entries from node of bushy tree

Syntax

```
Values = bushpath(Tree,BranchList)
```

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.

Examples

Extract Entries From Node of Bushy Tree

Create an HJM tree by loading the example file.

```
load deriv.mat;
```

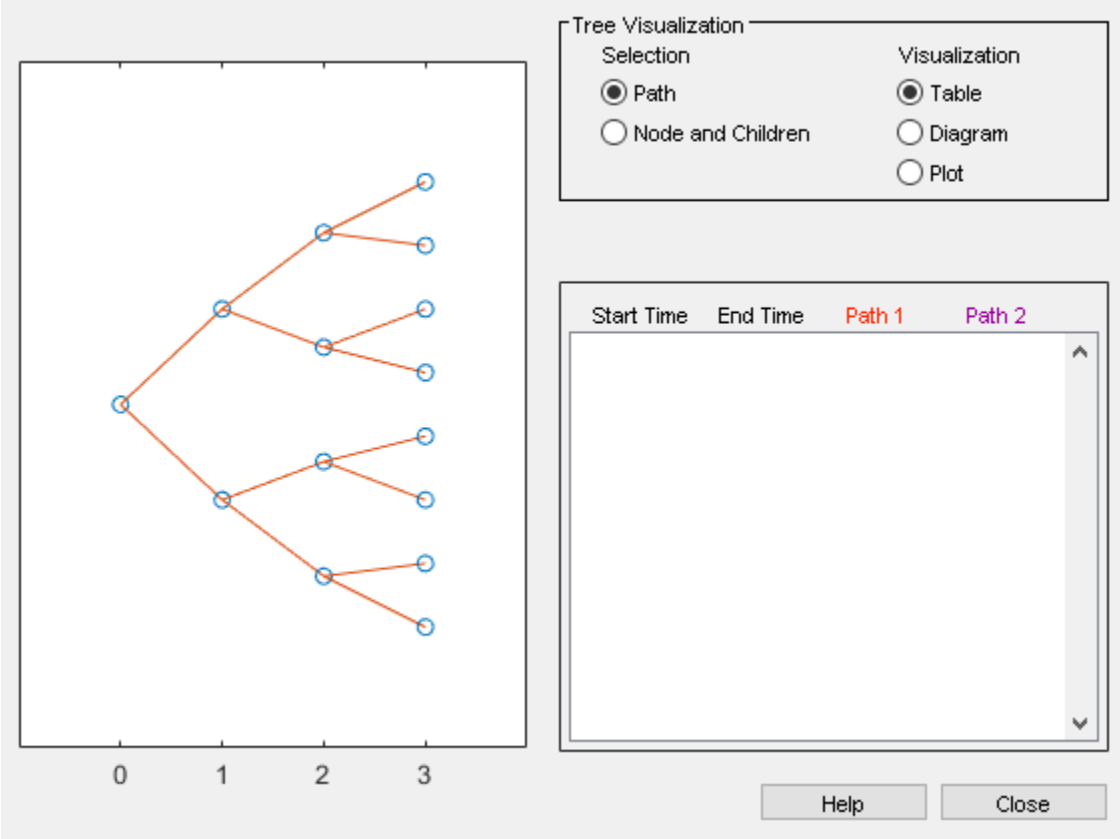
Use `bushpath` to return the rates at the tree nodes located by taking the up branch, then the down branch, and finally the up branch again.

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

```
FwdRates = 4×1

    1.0356
    1.0364
    1.0526
    1.0463
```

You can visualize this with the `treeviewer` function.

```
treeviewer(HJMTree)
```



Input Arguments

`Tree` — Bushy tree

structure

Bushy tree, specified using an HJM, BDT, HW, BK, or CIR tree.

Data Types: `struct`

`BranchList` — Sequence of branching

matrix

Sequence of branching, specified as a number of paths (`NUMPATHS`) by path length (`PATHLENGTH`) matrix.

Data Types: `double`

Output Arguments

`Values` — Retrieved entries of a bushy tree

numeric

Retrieved entries of a bushy tree, returned as a number of values (`NUMVALS`)-by-`NUMPATHS` matrix.

Version History

Introduced before R2006a

See Also

`bushshape` | `mkbush`

Topics

“Graphical Representation of Trees” on page 2-219

bushshape

Retrieve shape of bushy tree

Syntax

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

Description

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

Examples

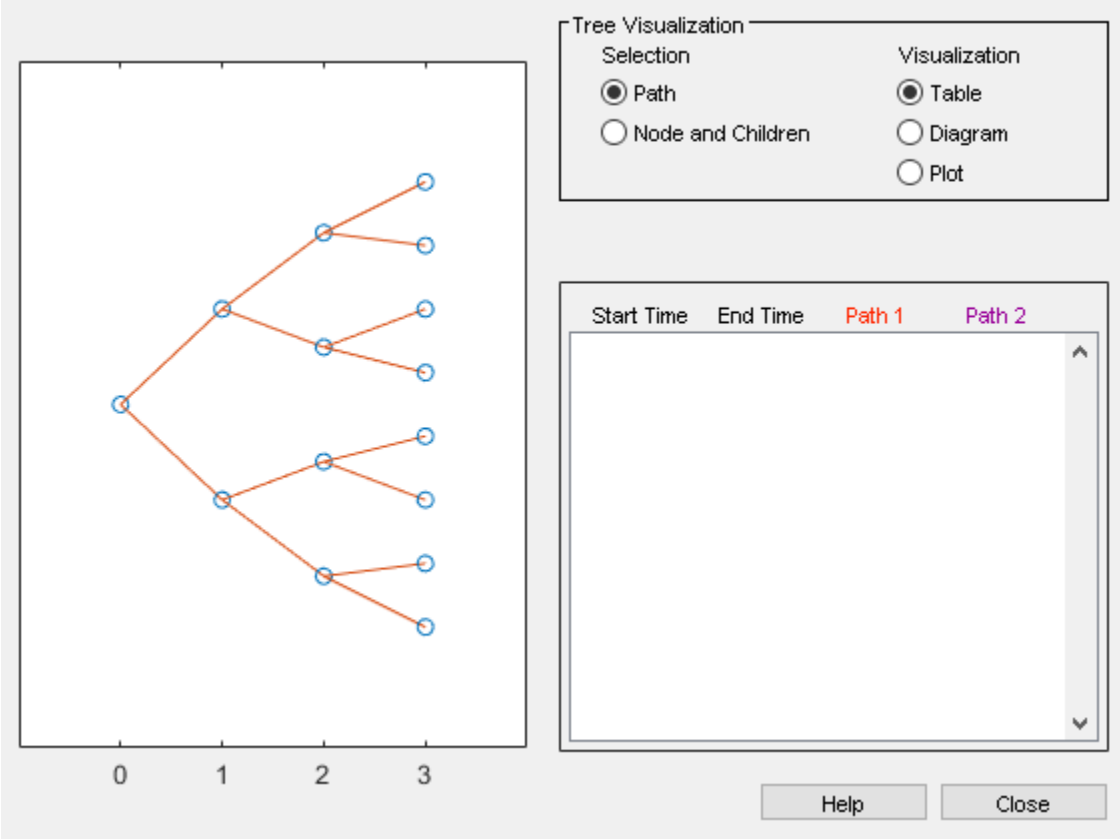
Retrieve Shape of Bushy Tree

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.

```
treeviewer(HJMTree)
```



Use `bushshape` with the `HJMTree`.

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

```
NumLevels = 4

NumChild = 1×4

     2     2     2     0

NumPos = 1×4

     4     3     2     1

NumStates = 1×4

     1     2     4     8

Trim = logical
   1
```

You can recreate this tree using the `mkbush` function.

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

```
Tree=1×4 cell array
    {4x1 double}    {3x1x2 double}    {2x2x2 double}    {1x4x2 double}
```

```
Tree = mkbush(NumLevels, NumChild, NumPos)
```

```
Tree=1×4 cell array
    {4x1 double}    {3x1x2 double}    {2x2x2 double}    {1x4x2 double}
```

Input Arguments

`Tree` — Bushy tree

structure

Bushy tree, specified using an HJM, BDT, HW, BK, or CIR tree.

Data Types: `struct`

Output Arguments

`NumLevels` — Number of tree levels

numeric

Number of tree levels, returned as a numeric.

`NumChild` — Number of branches of the nodes in each level

vector

Number of branches (children) of the nodes in each level, returned as a 1-by-number of levels (`NUMLEVELS`) vector.

`NumPos` — Length of the state vectors in each level

vector

Length of the state vectors in each level, returned as a 1-by-number of levels (`NUMLEVELS`) vector.

`NumStates` — Number of state vectors in each level

vector

Number of state vectors in each levels, returned as a 1-by-number of levels (`NUMLEVELS`) vector.

`Trim` — Trim

numeric

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

Version History

Introduced before R2006a

See Also

`bushpath` | `mkbush`

Topics

“Graphical Representation of Trees” on page 2-219

capbybdt

Price cap instrument from Black-Derman-Toy interest-rate tree

Syntax

```
[Price,PriceTree] = capbybdt(BDTTree,Strike,Settle,Maturity)
[Price,PriceTree] = capbybdt( ___ ,CapReset,Basis,Principal,Options)
```

Description

`[Price,PriceTree] = capbybdt(BDTTree,Strike,Settle,Maturity)` computes the price of a cap instrument from a Black-Derman-Toy interest-rate tree. `capbybdt` computes prices of vanilla caps and amortizing caps.

Note Alternatively, you can use the `Cap` object to price cap instruments. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`[Price,PriceTree] = capbybdt( ___ ,CapReset,Basis,Principal,Options)` adds optional arguments.

Examples

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 = datetime(2000,1,1);
Maturity = datetime(2004,1,1);
```

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

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

```
Price = 28.4001
```

Price a 10% Cap Instrument Using a BDT Interest-Rate Tree

Set the required arguments for the three specifications required to create a BDT tree.

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

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);
```

Create the BDT tree from the specifications.

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

```
BDTTree = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 4]
        dObs: [730486 730852 731217 731582 731947]
        TFwd: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [4]}
      CFlowT: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [5]}
     FwdTree: {1x5 cell}
```

Set the cap arguments. Remaining arguments will use defaults.

```
CapStrike = 0.10;
Settlement = ValuationDate;
Maturity = datetime(2002,1,1);
CapReset = 1;
```

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

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

```
Price = 1.7169
```

Compute the Price of an Amortizing Cap Using the BDT Model

Define the `RateSpec`.

```
Rates = [0.03583; 0.042147; 0.047345; 0.052707; 0.054302];
ValuationDate = datetime(2011,11,15);
StartDates = ValuationDate;
EndDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
```

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
          EndTimes: [5x1 double]
        StartTimes: [5x1 double]
          EndDates: [5x1 double]
        StartDates: 734822
    ValuationDate: 734822
            Basis: 0
     EndMonthRule: 1
```

Define the cap instrument.

```
Settle = datetime(2011,11,15);
Maturity = datetime(2015,11,15);
Strike = 0.04;
CapReset = 1;
Principal ={{datetime(2012,11,15) 100;datetime(2013,11,15) 70;datetime(2014,11,15) 40;datetime(20
```

Build the BDT Tree.

```
BDTTimeSpec = bdttimespec(ValuationDate, EndDates);
Volatility = 0.10;
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Volatility*ones(1,length(EndDates))');
BDTTree = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec)
```

```
BDTTree = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 4]
        dObs: [734822 735188 735553 735918 736283]
        TFwd: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [4]}
      CFlowT: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [5]}
     FwdTree: {1x5 cell}
```

Price the amortizing cap.

```
Basis = 0;
Price = capbybdt(BDTTree, Strike, Settle, Maturity, CapReset, Basis, Principal)
```

```
Price = 1.4042
```

Input Arguments

BDTTree — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bdttree`.

Data Types: `struct`

`Strike` — Rate at which cap is exercised
decimal

Rate at which cap is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for cap
datetime array | string array | date character vector

Settlement date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every cap is set to the `ValuationDate` of the BDT tree. The cap argument `Settle` is ignored.

To support existing code, `capbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for cap
datetime array | string array | date character vector

Maturity date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`CapReset` — Reset frequency payment per year
`1` (default) | numeric

(Optional) Reset frequency payment per year, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis representing the basis used when annualizing the input forward rate, specified as a `NINST`-by-`1` vector of integers.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing cap.

Data Types: `double` | `cell`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected price of cap at time 0
vector

Expected price of the cap at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure with values of cap at each node
vector

Tree structure with values of the cap at each node, returned as a MATLAB structure of trees containing vectors of instrument prices and a vector of observation times for each node:

- `PriceTree.PTree` contains cap prices.
- `PriceTree.tObs` contains the observation times.

More About

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:

$\max(CurrentRate - CapRate, 0)$

For more information, see “Cap” on page 2-12.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `capbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdttree` | `cfbybdt` | `floorbybdt` | `swapbybdt` | `capbynormal` | `Cap`

Topics

“Computing Instrument Prices” on page 2-81
“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Cap” on page 2-12
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

capbybk

Price cap instrument from Black-Karasinski interest-rate tree

Syntax

```
[Price,PriceTree] = capbybk(BKTree,Strike,Settle,Maturity)
[Price,PriceTree] = capbybk(___,CapReset,Basis,Principal,Options)
```

Description

`[Price,PriceTree] = capbybk(BKTree,Strike,Settle,Maturity)` computes the price of a cap instrument from a Black-Karasinski interest-rate tree. `capbybk` computes prices of vanilla caps and amortizing caps.

Note Alternatively, you can use the `Cap` object to price cap instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = capbybk(___,CapReset,Basis,Principal,Options)` adds optional arguments.

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 = datetime(2004,1,1);
Maturity = datetime(2007,1,1);
```

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

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

```
Price = 2.0965
```

Compute the Price of an Amortizing and Vanilla Caps Using the BK Model

Load `deriv.mat` to specify the `BKTree` and then define the cap instrument.

```
load deriv.mat;
Settle = datetime(2004,1,1);
Maturity = datetime(2008,1,1);
Strike = 0.05;
CapReset = 1;
Principal ={{datetime(2005,1,1) 100;datetime(2006,1,1) 60;datetime(2007,1,1) 30;datetime(2008,1,1
            100};
```

Price the amortizing and vanilla caps.

```
Basis = 1;
Price = capbybk(BKTree, Strike, Settle, Maturity, CapReset, Basis, Principal)
```

```
Price = 2×1

    0.2226
    0.7422
```

Input Arguments

`BKTree` — Interest-rate tree structure
structure

Interest-rate tree structure, specified by using `bktree`.

Data Types: `struct`

`Strike` — Rate at which cap is exercised
decimal

Rate at which cap is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for cap
datetime array | string array | date character vector

Settlement date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every cap is set to the `ValuationDate` of the BK tree. The cap argument `Settle` is ignored.

To support existing code, `capbybk` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for cap
datetime array | string array | date character vector

Maturity date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capbybk` also accepts serial date numbers as inputs, but they are not recommended.

`CapReset` — Reset frequency payment per year
1 (default) | numeric

(Optional) Reset frequency payment per year, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis representing the basis used when annualizing the input forward rate, specified as a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing cap.

Data Types: `double` | `cell`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected price of cap at time 0
vector

Expected price of the cap at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure with values of cap at each node
vector

Tree structure with values of the cap at each node, returned as a MATLAB structure of trees containing vectors of instrument prices and a vector of observation times for each node:

- `PriceTree.PTree` contains cap prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

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:

$\max(CurrentRate - CapRate, 0)$

For more information, see “Cap” on page 2-12.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `capbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cfbybk` | `floorbybk` | `bktree` | `swapbybk` | `capbynormal` | `Cap`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Cap” on page 2-12
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

capbyblk

Price caps using Black option pricing model

Syntax

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

Description

`[CapPrice,Caplets] = capbyblk(RateSpec,Strike,Settle,Maturity,Volatility)` price caps using the Black option pricing model. `capbyblk` computes prices of vanilla caps and amortizing caps.

Note Alternatively, you can use the `Cap` object to price cap instruments. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`[CapPrice,Caplets] = capbyblk( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Cap Using the Black Option Pricing Model

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. First, calculate the `RateSpec`:

```
ValuationDate = datetime(2008,1,1);
EndDates = datetime(2010,4,1);
Rates = 0.069394;
Compounding = -1;
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8554
            Rates: 0.0694
         EndTimes: 2.2500
       StartTimes: 0
         EndDates: 734229
```

```
       StartDates: 733408
    ValuationDate: 733408
            Basis: 1
     EndMonthRule: 1
```

Compute the price of the cap.

```
Settle = datetime(2009,1,1); % cap starts in a year
Maturity = datetime(2009,4,1);
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
```

Price a Cap Using a Different Curve to Generate the Future Forward Rates

Define the OIS and Libor rates.

```
Settle = datetime(2013,3,15);
CurveDates = daysadd(Settle,360*[1/12 2/12 3/12 6/12 1 2 3 4 5 7 10],1);
OISRates = [.0018 .0019 .0021 .0023 .0031 .006  .011 .017 .021 .026 .03]';
LiborRates = [.0045 .0047 .005 .0055 .0075 .0109  .0162 .0216 .0262 .0309 .0348]';
```

Create an associated `RateSpec` for the OIS and Libor curves.

```
OISCurve = intenvset('Rates',OISRates,'StartDate',Settle,'EndDates',CurveDates,'Compounding',2,'
LiborCurve = intenvset('Rates',LiborRates,'StartDate',Settle,'EndDates',CurveDates,'Compounding'
```

Define the Cap instruments.

```
Maturity = [ datetime(2018,3,15) ; datetime(2020,3,15)];
Strike = [0.04;0.05];
BlackVol = 0.2;
```

Price the cap instruments using the term structure `OISCurve` both for discounting the cash flows and generating future forward rates.

```
[Price, Caplets] = capbyblk(OISCurve, Strike, Settle, Maturity, BlackVol)
```

```
Price = 2×1

    0.7472
    0.9890


Caplets = 2×7

         0    0.0000    0.0033    0.2996    0.4443       NaN       NaN
         0    0.0000    0.0003    0.1134    0.2112    0.2292    0.4349
```

Price the cap instruments using the term structure `LiborCurve` to generate future forward rates. The term structure `OISCurve` is used for discounting the cash flows.

```
[PriceLC, CapletsLC] = capbyblk(OISCurve, Strike, Settle, Maturity, BlackVol,'ProjectionCurve',Li
```

```
PriceLC = 2×1

    1.3293
    1.6329


CapletsLC = 2×7

         0    0.0000    0.0337    0.4250    0.8706       NaN       NaN
         0    0.0000    0.0052    0.1767    0.4849    0.3663    0.5998
```

Compute the Price of Two Amortizing Caps Using the Black Model

Define the `RateSpec`.

```
Rates = [0.0358; 0.0421; 0.0473; 0.0527; 0.0543];
ValuationDate = datetime(2011,11,15);
StartDates = ValuationDate;
EndDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate,'StartDates', StartDates,...
             'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: [5x1 double]
       StartDates: 734822
    ValuationDate: 734822
            Basis: 0
     EndMonthRule: 1
```

Define the cap instruments.

```
Settle = datetime(2011,11,15);
Maturity = datetime(2015,11,15);
Strike = [0.03;0.035];
Reset = 1;
Principal ={{datetime(2012,11,15) 100;datetime(2013,11,15) 70;datetime(2014,11,15) 40;datetime(20
```

Price the amortizing caps.

```
Volatility = 0.10;
Price = capbyblk (RateSpec, Strike, Settle, Maturity, Volatility,...
'Reset', Reset,'Principal', Principal)
```

```
Price = 2×1

    3.0339
    2.0141
```

Price a Cap Using the Shifted Black Model

Create the RateSpec.

```
ValuationDate = datetime(2016,3,1);
EndDates = [datetime(2017,3,1) ; datetime(2018,3,1) ; datetime(2019,3,1) ; datetime(2020,3,1) ; d
Rates = [-0.21; -0.12; 0.01; 0.10; 0.20]/100;
Compounding = 1;
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
          EndTimes: [5x1 double]
        StartTimes: [5x1 double]
          EndDates: [5x1 double]
        StartDates: 736390
    ValuationDate: 736390
            Basis: 1
     EndMonthRule: 1
```

Price the cap with a negative strike using the Shifted Black model.

```
Settle = datetime(2016,6,1); % Cap starts in 3 months.
Maturity = datetime(2016,9,1);
ShiftedBlackVolatility = 0.31;
CapRate = -0.003;  % -0.3 percent strike.
CapReset = 4;
Principal = 100000;
Shift = 0.01; % 1 percent shift.

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

```
CapPrice = 26.0733
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`Strike` — Rate at which cap is exercised
decimal

Rate at which cap is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for cap
datetime array | string array | date character vector

Settlement date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capbyblk` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for cap
datetime array | string array | date character vector

Maturity date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capbyblk` also accepts serial date numbers as inputs, but they are not recommended.

`Volatility` — Volatilities values
numeric

Volatilities values, specified as a `NINST`-by-`1` vector of numeric values.

The `Volatility` input is not intended for volatility surfaces or cubes. If you specify a matrix for the `Volatility` input, `capbyblk` internally converts it into a vector. `capbyblk` assumes that the volatilities specified in the `Volatility` input are flat volatilities, which are applied equally to each of the caplets.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[CapPrice,Caplets] = capbyblk(RateSpec,Strike,Settle,Maturity,Volatility,'Reset',CapReset,'Principal',100000,'Basis',7)`

`Reset` — Reset frequency payment per year
`1` (default) | numeric

Reset frequency payment per year, specified as a NINST-by-1 vector.

Data Types: double

Principal — Notional principal amount
100 (default) | numeric

Notional principal amount, specified as a NINST-by-1 vector or a NINST-by-1 cell array. When Principal is a NINST-by-1 cell array, each element is a NumDates-by-2 cell array, where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use Principal to pass a schedule to compute the price for an amortizing cap.

Data Types: double | cell

Basis — Day-count basis of instrument
0 (actual/actual) (default) | integer from 0 to 13

Day-count basis representing the basis used when annualizing the input forward rate, specified as a NINST-by-1 vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: double

ProjectionCurve — Rate curve used in generating future forward rates
if ProjectionCurve is not specified, then RateSpec is used both for discounting cash flows and projecting future forward rates (default) | structure

The rate curve to be used in generating the future forward rates. This structure must be created using intenvset. Use this optional input if the forward curve is different from the discount curve.

Data Types: struct

Shift — Shift in decimals for shifted Black model
0 (no shift) (default) | positive decimal

Shift in decimals for the shifted Black model, specified using a scalar or `NINST`-by-`1` vector of rate shifts in positive decimals. Set this parameter to a positive rate shift in decimals to add a positive shift to the forward rate and strike, which effectively sets a negative lower bound for the forward rate. For example, a `Shift` of `0.01` is equal to a 1% shift.

Data Types: `double`

Output Arguments

`CapPrice` — Expected price of cap

vector

Expected price of the cap, returned as a `NINST`-by-`1` vector.

`Caplets` — Caplets

array

Caplets, returned as a `NINST`-by-`NCF` array of caplets, padded with `NaN`s.

More About

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:

$\max(CurrentRate - CapRate, 0)$

For more information, see “Cap” on page 2-12.

Shifted Black

The Shifted Black model is essentially the same as the Black’s model, except that it models the movements of ($F + Shift$) as the underlying asset, instead of F (which is the forward rate in the case of caplets).

This model allows negative rates, with a fixed negative lower bound defined by the amount of shift; that is, the zero lower bound of Black’s model has been shifted.

Algorithms

Black Model

$$dF = \sigma_{Black} F dw$$

$$call = e^{-\gamma T}[FN(d_1) - KN(d_2)]$$

$$put = e^{-\gamma T}[KN(-d_2) - FN(-d_1)]$$

$$d_1 = \frac{\ln\left(\frac{F}{K}\right) + \left(\frac{\sigma_B{}^2}{2}\right)T}{\sigma_B\sqrt{T}}, \quad d_2 = d_1 - \sigma_B\sqrt{T}$$

$$\sigma_B = \sigma_{Black}$$

Where F is the forward value and K is the strike.

Shifted Black Model

$$dF = \sigma_{Shifted\_Black}(F + Shift)dw$$

$$call = e^{-\gamma T}[(F + Shift)N(d_{s1}) - (K + Shift)N(d_{s2})]$$

$$put = e^{-\gamma T}[(K + Shift)N(-d_{s2}) - (F + Shift)N(-d_{s1})]$$

$$d_{s1} = \frac{\ln\left(\frac{F + Shift}{K + Shift}\right) + \left(\frac{\sigma_{sB}{}^2}{2}\right)T}{\sigma_{sB}\sqrt{T}}, \quad d_{s2} = d_{s1} - \sigma_{sB}\sqrt{T}$$

$$\sigma_{sB} = \sigma_{Shifted\_Black}$$

Where $F+Shift$ is the forward value and $K+Shift$ is the strike for the shifted version.

Version History

Introduced in R2009a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `capbyblk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`floorbyblk` | `intenvset` | `capbynormal` | `Cap`

Topics

“Cap” on page 2-12
“Work with Negative Interest Rates Using Functions” on page 2-18
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

External Websites

How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

capbycir

Price cap instrument from Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = capbycir(CIRTree,Strike,Settle,Maturity)
[Price,PriceTree] = capbycir(___,Name,Value)
```

Description

`[Price,PriceTree] = capbycir(CIRTree,Strike,Settle,Maturity)` computes the price of a cap instrument from a Cox-Ingersoll-Ross (CIR) interest-rate tree. `capbycir` computes prices of vanilla caps and amortizing caps using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = capbycir(___,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Cap Using a CIR Interest-Rate Tree

Define the `Strike` for a cap.

```
Strike = 0.03;
```

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = 'Jan-1-2017';
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',EndD
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = datetime(2017,1,1);
Maturity = datetime(2021,1,1);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
```

```
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [736696 737061 737426 737791]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Price the 3% cap.

```
[Price,PriceTree] = capbycir(CIRT,Strike,Settle,Maturity)
```

```
Price = 7.9081
```

```
PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
       tObs: [0 1 2 3 4]
      PTree: {1x5 cell}
    Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
      Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `cirtree`.

Data Types: `struct`

`Strike` — Rate at which cap is exercised

decimal

Rate at which cap is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for cap

datetime array | string array | date character vector

Settlement date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every cap is set to the `ValuationDate` of the CIR tree. The cap argument `Settle` is ignored.

To support existing code, `capbycir` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for cap

datetime array | string array | date character vector

Maturity date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capbycir` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = capbycir(CIRTree,CouponRate,Settle,Maturity,'Basis',3)`

`CapReset` — Reset frequency payment per year

`1` (default) | numeric

Reset frequency payment per year, specified as the comma-separated pair consisting of `'CapReset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward rate, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount

`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` of notional principal amounts or a `NINST`-by-`1` cell array.

For the NINST-by-1 cell array, each element is a NumDates-by-2 cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use Principal to pass a schedule to compute the price for an amortizing cap.

Data Types: double | cell

Output Arguments

Price — Expected price of cap at time 0

vector

Expected price of the cap at time 0, returned as a NINST-by-1 vector.

PriceTree — Tree structure with values of cap at each node

vector

Tree structure with values of the cap at each node, returned as a MATLAB structure of trees containing vectors of instrument prices and a vector of observation times for each node:

- PriceTree.PTree contains cap prices.
- PriceTree.tObs contains the observation times.
- PriceTree.Connect contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are NumNodes elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- PriceTree.Probs contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

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:

$\max(CurrentRate - CapRate, 0)$

For more information, see “Cap” on page 2-12.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `capbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J.,and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir` | `instcap`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Cap” on page 2-12
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

capbyhjm

Price cap instrument from Heath-Jarrow-Morton interest-rate tree

Syntax

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

Description

`[Price,PriceTree] = capbyhjm(HJMTree,Strike,Settle,Maturity)` computes the price of a cap instrument from a Heath-Jarrow-Morton interest-rate tree. `capbyhjm` computes prices of vanilla caps and amortizing caps.

`[Price,PriceTree] = capbyhjm(___,CapReset,Basis,Principal,Options)` adds optional arguments.

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 = datetime(2000,1,1);
Maturity = datetime(2004,1,1);
```

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

```
Price = capbyhjm(HJMTree, Strike, Settle, Maturity)
```

```
Price = 6.2831
```

Compute the Price of an Amortizing Cap Using the HJM Model

Load `deriv.mat` to specify the `HJMTree` and then define the cap instrument.

```
load deriv.mat;
Settle = datetime(2000,1,1);
Maturity = datetime(2004,1,1);
Strike = 0.045;
CapReset = 1;
Principal ={{datetime(2001,1,1) 100;datetime(2002,1,1) 80;datetime(2003,1,1) 70;datetime(2004,1,1
```

Price the amortizing cap.

```
Basis = 1;
Price = capbyhjm(HJMTree, Strike, Settle, Maturity, CapReset, Basis, Principal)
```

```
Price = 1.4588
```

Input Arguments

`HJMTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hjmtree`.

Data Types: `struct`

`Strike` — Rate at which cap is exercised

decimal

Rate at which cap is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for cap

datetime array | string array | date character vector

Settlement date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every cap is set to the `ValuationDate` of the HJM tree. The cap argument `Settle` is ignored.

To support existing code, `capbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for cap

datetime array | string array | date character vector

Maturity date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`CapReset` — Reset frequency payment per year

`1` (default) | numeric

(Optional) Reset frequency payment per year, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis representing the basis used when annualizing the input forward rate, specified as a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing cap.

Data Types: `double` | `cell`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected price of cap at time 0
vector

Expected price of the cap at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure with values of cap at each node
vector

Tree structure with values of the cap at each node, returned as a MATLAB structure of trees containing vectors of instrument prices and a vector of observation times for each node:

- `PriceTree.tObs` contains the observation times.
- `PriceTree.PBush` contains the clean prices.

More About

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:

$\max(CurrentRate - CapRate, 0)$

For more information, see “Cap” on page 2-12.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `capbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cfbyhjm` | `floorbyhjm` | `hjmtree` | `swapbyhjm` | `capbynormal`

Topics

“Computing Instrument Prices” on page 2-81
“Cap” on page 2-12
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

capbyhw

Price cap instrument from Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = capbyhw(HWTree,Strike,Settle,Maturity)
[Price,PriceTree] = capbyhw(___,CapReset,Basis,Principal,Options)
```

Description

`[Price,PriceTree] = capbyhw(HWTree,Strike,Settle,Maturity)` computes the price of a cap instrument from a Hull-White interest-rate tree. `capbyhw` computes prices of vanilla caps and amortizing caps.

Note Alternatively, you can use the `Cap` object to price cap instruments. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`[Price,PriceTree] = capbyhw(___,CapReset,Basis,Principal,Options)` adds optional arguments.

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 = datetime(2004,1,1);
Maturity = datetime(2007,1,1);
```

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

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

```
Price = 2.3090
```

Compute the Price of an Amortizing and Vanilla Caps Using the HW Model

Define the `RateSpec`.

```
Rates = [0.035; 0.042; 0.047; 0.052; 0.054];
ValuationDate = datetime(2014,4,1);
StartDates = ValuationDate;
EndDates = datetime(2019,4,1);
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate,'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding)

RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: 737516
       StartDates: 735690
    ValuationDate: 735690
            Basis: 0
     EndMonthRule: 1
```

Define the cap instruments.

```
Settle = datetime(2014,4,1);
Maturity = datetime(2018,4,1);
Strike = 0.055;
CapReset = 1;
Principal ={{datetime(2015,4,1) 100;datetime(2016,4,1) 60;datetime(2017,4,1) 40;datetime(2018,4,1
            100};
```

Build the HW Tree.

```
VolDates = [datetime(2015,4,1) ; datetime(2016,4,1) ; datetime(2017,4,1) ; datetime(2018,4,1)];
VolCurve = 0.05;
AlphaDates = datetime(2018,4,1);
AlphaCurve = 0.10;

HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
                      AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec)

HWTree = struct with fields:
      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [735690 736055 736421 736786]
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
     FwdTree: {1x4 cell}
```

Price the amortizing and vanilla caps.

```
Basis = 0;
Price  = capbyhw(HWTree, Strike, Settle, Maturity, CapReset, Basis, Principal)
```

```
Price = 2×1

    1.6754
    4.6149
```

Input Arguments

`HWTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hwtree`.

Data Types: `struct`

`Strike` — Rate at which cap is exercised

decimal

Rate at which cap is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for cap

datetime array | string array | date character vector

Settlement date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every cap is set to the `ValuationDate` of the HW tree. The cap argument `Settle` is ignored.

To support existing code, `capbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for cap

datetime array | string array | date character vector

Maturity date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`CapReset` — Reset frequency payment per year

`1` (default) | numeric

(Optional) Reset frequency payment per year, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis representing the basis used when annualizing the input forward rate, specified as a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing cap.

Data Types: `double` | `cell`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected price of cap at time 0
vector

Expected price of the cap at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure with values of cap at each node
vector

Tree structure with values of the cap at each node, returned as a MATLAB structure of trees containing vectors of instrument prices and a vector of observation times for each node:

- `PriceTree.PTree` contains cap prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

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:

$\max(CurrentRate - CapRate, 0)$

For more information, see “Cap” on page 2-12.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `capbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cfbyhw` | `floorbyhw` | `hwtree` | `swapbyhw` | `capbynormal` | `Cap`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Cap” on page 2-12

“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

capbylg2f

Price cap using Linear Gaussian two-factor model

Syntax

```
CapPrice = capbylg2f(ZeroCurve,a,b,sigma,eta,rho,Strike,Maturity)
CapPrice = capbylg2f( ___ ,Name,Value)
```

Description

`CapPrice = capbylg2f(ZeroCurve,a,b,sigma,eta,rho,Strike,Maturity)` returns cap price for a two-factor additive Gaussian interest-rate model.

Note Alternatively, you can use the `Cap` object to price cap instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`CapPrice = capbylg2f( ___ ,Name,Value)` adds optional name-value pair arguments.

Note Use the optional name-value pair argument, `Notional`, to pass a schedule to compute the price for an amortizing cap.

Examples

Price a Cap Using a Linear Gaussian Two-Factor Model

Define the `ZeroCurve`, `a`, `b`, `sigma`, `eta`, and `rho` parameters to price the cap.

```
Settle = datetime(2007,12,15);

ZeroTimes = [3/12 6/12 1 5 7 10 20 30]';
ZeroRates = [0.033 0.034 0.035 0.040 0.042 0.044 0.048 0.0475]';
CurveDates = daysadd(Settle,360*ZeroTimes);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

a = .07;
b = .5;
sigma = .01;
eta = .006;
rho = -.7;

CapMaturity = daysadd(Settle,360*[1:5 7 10 15 20 25 30],1);

Strike = [0.035 0.037 0.038 0.039 0.040 0.042 0.044 0.046 0.047 0.047 0.047]';

Price = capbylg2f(irdc,a,b,sigma,eta,rho,Strike,CapMaturity)
```

```
Price = 11×1

    0.0218
    0.3167
    0.7640
    1.3055
    1.9152
    3.0909
    4.7998
    7.3122
    9.7917
   11.4568
      ⋮
```

Price an Amortizing Cap Using a Linear Gaussian Two-Factor Model

Define the `ZeroCurve`, `a`, `b`, `sigma`, `eta`, `rho`, and `Notional` parameters for the amortizing cap.

```
Settle = datetime(2007,12,15);
% Define ZeroCurve
ZeroTimes = [3/12 6/12 1 5 7 10 20 30]';
ZeroRates = [0.033 0.034 0.035 0.040 0.042 0.044 0.048 0.0475]';
CurveDates = daysadd(Settle,360*ZeroTimes);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

% Define a, b, sigma, eta, and rho
a = .07;
b = .5;
sigma = .01;
eta = .006;
rho = -.7;

% Define the amortizing caps
CapMaturity = daysadd(Settle,360*[1:5 7 10 15 20 25 30],1);
Strike = [0.035 0.037 0.038 0.039 0.040 0.042 0.044 0.046 0.047 0.047 0.047]';
Notional = {{datetime(2010,12,15) 100;datetime(2014,12,15) 70;datetime(2022,12,15) 40;datetime(20

% Price the amortizing caps
Price = capbylg2f(irdc,a,b,sigma,eta,rho,Strike,CapMaturity, 'Notional', Notional)
```

```
Price = 11×1

    0.0218
    0.3167
    0.7640
    1.1150
    1.5162
    2.2952
    2.8006
    3.6532
    3.6963
    3.8628
      ⋮
```

Input Arguments

`ZeroCurve` — Zero-curve for Linear Gaussian two-factor model

structure

Zero-curve for the Linear Gaussian two-factor model, specified using `IRDataCurve` or `RateSpec`.

Data Types: `struct`

`a` — Mean reversion for first factor for Linear Gaussian two-factor model

scalar numeric

Mean reversion for first factor for the Linear Gaussian two-factor model, specified as a scalar numeric.

Data Types: `double`

`b` — Mean reversion for second factor for Linear Gaussian two-factor model

scalar numeric

Mean reversion for second factor for the Linear Gaussian two-factor model, specified as a scalar numeric.

Data Types: `double`

`sigma` — Volatility for first factor for Linear Gaussian two-factor model

scalar numeric

Volatility for first factor for the Linear Gaussian two-factor model, specified as a scalar numeric.

Data Types: `double`

`eta` — Volatility for second factor for Linear Gaussian two-factor model

scalar numeric

Volatility for second factor for the Linear Gaussian two-factor model, specified as a scalar numeric.

Data Types: `double`

`rho` — Scalar correlation of the factors

scalar numeric

Scalar correlation of the factors, specified as a scalar numeric.

Data Types: `double`

`Strike` — Cap strike price

nonnegative integer | vector of nonnegative integers

Cap strike price, specified as a nonnegative integer using a `NumCaps`-by-`1` vector.

Data Types: `double`

`Maturity` — Cap maturity date

datetime array | string array | date character vector

Cap maturity date, specified using a `NumCaps`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capbylg2f` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = capbylg2f(irdc,a,b,sigma,eta,rho,Strike,CapMaturity,'Reset',1,'Notional',100)`

`Reset` — Frequency of cap payments per year
`2` (default) | positive integer from the set`[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of cap payments per year, specified as the comma-separated pair consisting of `'Reset'` and a positive integers for the values `[1,2,4,6,12]` in a `NumCaps`-by-`1` vector.

Data Types: `single` | `double`

`Notional` — Notional value of cap
`100` (default) | nonnegative integer | vector of nonnegative integers

Notional value of cap, specified as the comma-separated pair consisting of `'Notional'` and a `NINST`-by-`1` of notional principal amounts or `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is the associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `single` | `double`

Output Arguments

`CapPrice` — Cap price
scalar | vector

Expected prices of cap, returned as a scalar or an `NumCaps`-by-`1` vector.

More About

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:

$\max(CurrentRate - CapRate, 0)$

For more information, see “Cap” on page 2-12.

Algorithms

The following defines the two-factor additive Gaussian interest rate model, given the `ZeroCurve`, `a`, `b`, `sigma`, `eta`, and `rho` parameters:

$$r(t) = x(t) + y(t) + \phi(t)$$

$$dx(t) = -a(x)(t)dt + \sigma(dW_1(t), x(0) = 0$$

$$dy(t) = -b(y)(t)dt + \eta(dW_2(t), y(0) = 0$$

where $dW_1(t)dW_2(t) = \rho dt$ is a two-dimensional Brownian motion with correlation ρ and ϕ is a function chosen to match the initial zero curve.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `capbylg2f` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

See Also

`floorbylg2f` | `swaptionbylg2f` | `LinearGaussian2F` | `Cap`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Cap” on page 2-12
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

capbynormal

Price caps using Normal or Bachelier pricing model

Syntax

```
[CapPrice,Caplets] = capbynormal(RateSpec,Strike,Settle,Maturity,Volatility)
[CapPrice,Caplets] = capbynormal( ___ ,Name,Value)
```

Description

`[CapPrice,Caplets] = capbynormal(RateSpec,Strike,Settle,Maturity,Volatility)` prices caps using the Normal (Bachelier) pricing model for negative rates. `capbynormal` computes prices of vanilla caps and amortizing caps.

Note Alternatively, you can use the `Cap` object to price cap instruments. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`[CapPrice,Caplets] = capbynormal( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Cap Using Normal Model for Negative Rates

Consider an investor who gets into a contract that caps the interest rate on a $100,000 loan at –.08% quarterly compounded for 3 months, starting on January 1, 2009. Assuming that on January 1, 2008 the zero rate is .069394% continuously compounded and the volatility is 20%, use this data to compute the cap price. First, calculate the `RateSpec`, and then use `capbynormal` to compute the `CapPrice`.

```
ValuationDate = datetime(2008,1,1);
EndDates = datetime(2010,4,1);
Rates = 0.0069394;
Compounding = -1;
Basis = 1;

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


Settle = datetime(2009,1,1); % cap starts in a year
Maturity = datetime(2009,4,1);
Volatility = 0.20;
CapRate = -0.008;
CapReset = 4;
Principal=100000;
```

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

```
CapPrice = 2.1682e+03
```

Price a Cap Using capbynormal and Compare to capbyblk

Define the RateSpec.

```
Settle = datetime(2016,1,20);
ZeroTimes = [.5 1 2 3 4 5 7 10 20 30]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = datemnth(Settle,12*ZeroTimes);
RateSpec = intenvset('StartDate',Settle,'EndDates',ZeroDates,'Rates',ZeroRates)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [10x1 double]
            Rates: [10x1 double]
         EndTimes: [10x1 double]
       StartTimes: [10x1 double]
         EndDates: [10x1 double]
       StartDates: 736349
    ValuationDate: 736349
            Basis: 0
     EndMonthRule: 1
```

Define the cap instrument and price with capbyblk.

```
ExerciseDate = datetime(2026,1,20);

[~,ParSwapRate] = swapbyzero(RateSpec,[NaN 0],Settle,ExerciseDate)
```

```
ParSwapRate = 0.0216
```

```
Strike = .01;
BlackVol = .3;
NormalVol = BlackVol*ParSwapRate;

Price = capbyblk(RateSpec,Strike,Settle,ExerciseDate,BlackVol)
```

```
Price = 11.8693
```

Price the cap instrument using capbynormal.

```
Price_Normal = capbynormal(RateSpec,Strike,Settle,ExerciseDate,NormalVol)
```

```
Price_Normal = 12.5495
```

Price the cap instrument using capbynormal for a negative strike.

```
 Price_Normal = capbynormal(RateSpec,-.005,Settle,ExerciseDate,NormalVol)
```

```
Price_Normal = 24.4816
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

Strike — Rate at which cap is exercised

decimal

Rate at which cap is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

Settle — Settlement date for cap

datetime array | string array | date character vector

Settlement date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capbynormal` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `double` | `char` | `datetime` | `string`

Maturity — Maturity date for cap

datetime array | string array | date character vector

Maturity date for the cap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capbynormal` also accepts serial date numbers as inputs, but they are not recommended.

Volatility — Normal volatilities values

numeric

Normal volatilities values, specified as a `NINST`-by-`1` vector of numeric values.

For more information on the Normal model, see “Work with Negative Interest Rates Using Functions” on page 2-18.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[CapPrice,Caplets] = capbynormal(RateSpec,Strike,Settle,Maturity,Volatility,'Reset',CapReset,'Principal',100000,'Basis',7)`

`Reset` — Reset frequency payment per year
`1` (default) | numeric

Reset frequency payment per year, specified as the comma-separated pair consisting of `'Reset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array. Each element in the `NINST`-by-`1` cell array is a `NumDates`-by-`2` cell array, where the first column is dates, and the second column is the associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing cap.

Data Types: `double` | `cell`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis of instrument representing the basis used when annualizing the input forward rate, specified as the comma-separated pair consisting of `'Basis'`and a `NINST`-by-`1` vector of integers. Values are:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`ValuationDate` — Observation date of investment horizon
if `ValuationDate` is not specified, then `Settle` is used (default) | datetime scalar | string scalar | date character vector

Observation date of the investment horizon, specified as the comma-separated pair consisting of `'ValuationDate'` and a scalar datetime, string, or date character vector.

To support existing code, `capbynormal` also accepts serial date numbers as inputs, but they are not recommended.

`ProjectionCurve` — Rate curve used in generating future cash flows
if `ProjectionCurve` is not specified, then `RateSpec` is used both for discounting cash flows and projecting future cash flows (default) | structure

The rate curve to be used in projecting the future cash flows, specified as the comma-separated pair consisting of `'ProjectionCurve'` and rate curve structure. This structure must be created using `intenvset`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `struct`

Output Arguments

`CapPrice` — Expected price of cap
vector

Expected price of the cap, returned as a `NINST`-by-`1` vector.

`Caplets` — Caplets
array

Caplets, returned as a `NINST`-by-`NCF` array of caplets, padded with `NaN`s.

More About

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)

For more information, see “Cap” on page 2-12.

Version History

Introduced in R2017a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `capbynormal` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`floorbynormal` | `intenvset` | `swaptionbynormal` | `capbyblk` | `Cap`

Topics

“Calibrating Caplets Using the Normal (Bachelier) Model” on page 2-155
“Cap” on page 2-12
“Work with Negative Interest Rates Using Functions” on page 2-18
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

External Websites

How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

capvolstrip

Strip caplet volatilities from flat cap volatilities

Syntax

```
[CapletVols,CapletPaymentDates,CapStrikes] = capvolstrip(ZeroCurve,CapSettle,
CapMaturity,CapVolatility)
[CapletVols,CapletPaymentDates,CapStrikes] = capvolstrip( ___ ,Name,Value)
```

Description

`[CapletVols,CapletPaymentDates,CapStrikes] = capvolstrip(ZeroCurve,CapSettle, CapMaturity,CapVolatility)` strips caplet volatilities from the flat cap volatilities by using the bootstrapping method. The function interpolates the cap volatilities on each caplet payment date before stripping the caplet volatilities.

`[CapletVols,CapletPaymentDates,CapStrikes] = capvolstrip( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Stripping Caplet Volatilities from At-The-Money (ATM) Caps

Compute the zero curve for discounting and projecting forward rates.

```
ValuationDate = datetime(2015,6,23);
ZeroRates = [0.01 0.09 0.30 0.70 1.07 1.71]/100;
CurveDates = datemnth(ValuationDate, [0.25 0.5 1 2 3 5]*12);
ZeroCurve = IRDataCurve('Zero',ValuationDate,CurveDates,ZeroRates)
```

```
ZeroCurve =
             Type: Zero
           Settle: 736138 (23-Jun-2015)
      Compounding: 2
            Basis: 0 (actual/actual)
     InterpMethod: linear
            Dates: [6x1 double]
             Data: [6x1 double]
```

Define the ATM cap volatility data.

```
CapSettle = datetime(2015,6,25);
CapMaturity = [datetime(2016,6,27) ; datetime(2017,6,26) ; datetime(2018,6,25) ; datetime(2019,6
CapVolatility = [0.29;0.38;0.42;0.40;0.38];
```

Strip caplet volatilities from ATM caps.

```
[CapletVols, CapletPaymentDates, ATMCapStrikes] = capvolstrip(ZeroCurve, ...
    CapSettle, CapMaturity, CapVolatility);
```

```
PaymentDates = cellstr(datestr(CapletPaymentDates));
format;
table(PaymentDates, CapletVols, ATMCapStrikes)
```

```
ans=9×3 table
     PaymentDates      CapletVols    ATMCapStrikes
    _______________    __________    _____________

    {'27-Jun-2016'}        0.29        0.0052014
    {'27-Dec-2016'}     0.34657        0.0071594
    {'26-Jun-2017'}     0.41404        0.0091175
    {'26-Dec-2017'}     0.42114         0.010914
    {'25-Jun-2018'}     0.45297         0.012698
    {'26-Dec-2018'}     0.37257         0.014222
    {'25-Jun-2019'}     0.36184         0.015731
    {'26-Dec-2019'}      0.3498         0.017262
    {'25-Jun-2020'}     0.33668         0.018774
```

Stripping Caplet Volatilities from Caps with the Same Strikes

Compute the zero curve for discounting and projecting forward rates.

```
ValuationDate = datetime(2015,2,17);
ZeroRates = [0.02 0.07 0.25 0.70 1.10 1.62]/100;
CurveDates = datemnth(ValuationDate, [0.25 0.5 1 2 3 5]*12);
ZeroCurve = IRDataCurve('Zero',ValuationDate,CurveDates,ZeroRates)
```

```
ZeroCurve =
            Type: Zero
          Settle: 736012 (17-Feb-2015)
     Compounding: 2
           Basis: 0 (actual/actual)
    InterpMethod: linear
           Dates: [6x1 double]
            Data: [6x1 double]
```

Define the cap volatility data.

```
CapSettle = datetime(2015,2,19);
CapMaturity = [datetime(2016,2,19) ; datetime(2017,2,21) ; datetime(2018,2,20) ; datetime(2019,2
CapVolatility = [0.44;0.45;0.44;0.41;0.39];
CapStrike = 0.013;
```

Strip caplet volatilities from caps with the same strike.

```
[CapletVols, CapletPaymentDates, CapStrikes] = capvolstrip(ZeroCurve, ...
    CapSettle, CapMaturity, CapVolatility, 'Strike', CapStrike);

PaymentDates = cellstr(datestr(CapletPaymentDates));
format;
table(PaymentDates, CapletVols, CapStrikes)
```

```
ans=9×3 table
     PaymentDates      CapletVols    CapStrikes
```

```
    ________________    __________    _________

    {'19-Feb-2016'}        0.44         0.013
    {'19-Aug-2016'}     0.44495         0.013
    {'21-Feb-2017'}     0.45256         0.013
    {'21-Aug-2017'}     0.43835         0.013
    {'20-Feb-2018'}     0.42887         0.013
    {'20-Aug-2018'}     0.38157         0.013
    {'19-Feb-2019'}     0.35237         0.013
    {'19-Aug-2019'}      0.3525         0.013
    {'19-Feb-2020'}     0.33136         0.013
```

Stripping Caplet Volatilities Using Manually Specified Caplet Dates

Compute the zero curve for discounting and projecting forward rates.

```
ValuationDate = datetime(2015,3,6);
ZeroRates = [0.01 0.08 0.27 0.73 1.16 1.70]/100;
CurveDates = datemnth(ValuationDate, [0.25 0.5 1 2 3 5]*12);
ZeroCurve = IRDataCurve('Zero',ValuationDate,CurveDates,ZeroRates)
```

```
ZeroCurve =
            Type: Zero
          Settle: 736029 (06-Mar-2015)
    Compounding: 2
           Basis: 0 (actual/actual)
    InterpMethod: linear
           Dates: [6x1 double]
            Data: [6x1 double]
```

Define the cap volatility data.

```
CapSettle = datetime(2015,3,6);
CapMaturity = [datetime(2016,3,7) ; datetime(2017,3,6) ; datetime(2018,3,6) ; datetime(2019,3,6)
CapVolatility = [0.43;0.44;0.44;0.43;0.41];
CapStrike = 0.011;
```

Specify quarterly and semiannual dates.

```
CapletDates = [cfdates(CapSettle, datetime(2016,3,6), 4) ...
     cfdates(datetime(2016,3,6), datetime(2020,3,6), 2)]';
CapletDates(~isbusday(CapletDates)) =  ...
    busdate(CapletDates(~isbusday(CapletDates)), 'modifiedfollow');
```

Strip caplet volatilities using specified `CapletDates`.

```
[CapletVols, CapletPaymentDates, CapStrikes] = capvolstrip(ZeroCurve, ...
    CapSettle, CapMaturity, CapVolatility, 'Strike', CapStrike, ...
    'CapletDates', CapletDates);

PaymentDates = cellstr(datestr(CapletPaymentDates));
format;
table(PaymentDates, CapletVols, CapStrikes)
```

```
ans=11×3 table
     PaymentDates      CapletVols    CapStrikes
    _______________    __________    __________

    {'08-Sep-2015'}        0.43         0.011
    {'07-Dec-2015'}     0.42999         0.011
    {'07-Mar-2016'}        0.43         0.011
    {'06-Sep-2016'}     0.43538         0.011
    {'06-Mar-2017'}     0.44396         0.011
    {'06-Sep-2017'}     0.43999         0.011
    {'06-Mar-2018'}     0.44001         0.011
    {'06-Sep-2018'}     0.41934         0.011
    {'06-Mar-2019'}     0.40985         0.011
    {'06-Sep-2019'}     0.36818         0.011
    {'06-Mar-2020'}     0.34657         0.011
```

Stripping Caplet Volatilities from Caps Using the Shifted Black Model

Compute the zero curve for discounting and projecting forward rates.

```
ValuationDate = datetime(2016,3,1);
ZeroRates = [-0.38 -0.25 -0.21 -0.12 0.01 0.2]/100;
CurveDates = datemnth(ValuationDate, [0.25 0.5 1 2 3 5]*12);
ZeroCurve = IRDataCurve('Zero',ValuationDate,CurveDates,ZeroRates)
```

```
ZeroCurve =
            Type: Zero
          Settle: 736390 (01-Mar-2016)
     Compounding: 2
           Basis: 0 (actual/actual)
    InterpMethod: linear
           Dates: [6x1 double]
            Data: [6x1 double]
```

Define the cap volatility (Shifted Black) data.

```
CapSettle = datetime(2016,3,1);
CapMaturity = [datetime(2017,3,1); datetime(2018,3,1) ; datetime(2019,3,1) ; datetime(2020,3,2)
CapVolatility = [0.35;0.40;0.37;0.34;0.32]; % Shifted Black volatilities
Shift = 0.01; % 1 percent shift.
CapStrike = -0.001; % -0.1 percent strike.
```

Strip caplet volatilities from caps using the Shifted Black Model.

```
[CapletVols, CapletPaymentDates, CapStrikes] = capvolstrip(ZeroCurve, ...
CapSettle,CapMaturity,CapVolatility,'Strike',CapStrike,'Shift',Shift);

PaymentDates = string(datestr(CapletPaymentDates));
format;
table(PaymentDates,CapletVols,CapStrikes)
```

```
ans=9×3 table
    PaymentDates      CapletVols    CapStrikes
    _____________    __________    __________
```

```
    "01-Mar-2017"        0.35        -0.001
    "01-Sep-2017"     0.39129        -0.001
    "01-Mar-2018"      0.4335        -0.001
    "04-Sep-2018"     0.35284        -0.001
    "01-Mar-2019"      0.3255        -0.001
    "03-Sep-2019"      0.3011        -0.001
    "02-Mar-2020"     0.27266        -0.001
    "01-Sep-2020"     0.27698        -0.001
    "01-Mar-2021"     0.25697        -0.001
```

Stripping Caplet Volatilities from Caps Using Normal Model

Compute the zero curve for discounting and projecting forward rates.

```
ValuationDate = datetime(2018,6,1);
ZeroRates = [-0.38 -0.25 -0.21 -0.12 0.01 0.2]/100;
CurveDates = datemnth(ValuationDate, [0.25 0.5 1 2 3 5]*12);
ZeroCurve = IRDataCurve('Zero',ValuationDate,CurveDates,ZeroRates)
```

```
ZeroCurve =
             Type: Zero
           Settle: 737212 (01-Jun-2018)
      Compounding: 2
            Basis: 0 (actual/actual)
     InterpMethod: linear
            Dates: [6x1 double]
             Data: [6x1 double]
```

Define the normal cap volatility data.

```
CapSettle = datetime(2018,6,1);
CapMaturity = [datetime(2019,6,3) ; datetime(2020,6,1) ; datetime(2021,6,1) ; datetime(2022,6,1)
CapVolatility = [0.0057;0.0059;0.0057;0.0053;0.0051]; % Normal volatilities
CapStrike = -0.002; % -0.2 percent strike.
```

Strip caplet volatilities from caps using the Normal (Bachelier) model.

```
[CapletVols, CapletPaymentDates, CapStrikes] = capvolstrip(ZeroCurve, ...
    CapSettle,CapMaturity,CapVolatility,'Strike',CapStrike,'Model','normal');

PaymentDates = string(datestr(CapletPaymentDates));
format;
table(PaymentDates,CapletVols,CapStrikes)
```

```
ans=9×3 table
    PaymentDates     CapletVols    CapStrikes
    ____________     __________    __________

    "03-Jun-2019"       0.0057       -0.002
    "02-Dec-2019"    0.0058686       -0.002
    "01-Jun-2020"    0.0060472       -0.002
    "01-Dec-2020"    0.0055705       -0.002
    "01-Jun-2021"    0.0053912       -0.002
```

```
    "01-Dec-2021"    0.0047404       -0.002
    "01-Jun-2022"     0.004357       -0.002
    "01-Dec-2022"    0.0046481       -0.002
    "01-Jun-2023"    0.0044477       -0.002
```

Input Arguments

`ZeroCurve` — Zero rate curve

`RateSpec` object | `IRDataCurve` object

Zero rate curve, specified using a `RateSpec` or `IRDataCurve` object containing the zero rate curve for discounting according to its day count convention. If you do not specify the optional argument `ProjectionCurve`, the function uses `ZeroCurve` to compute the underlying forward rates as well. The observation date of the `ZeroCurve` specifies the valuation date. For more information on creating a `RateSpec`, see `intenvset`. For more information on creating an `IRDataCurve` object, see `IRDataCurve`.

Data Types: `struct`

`CapSettle` — Common cap settle date

datetime scalar | string scalar | date character vector

Common cap settle date, specified as a scalar datetime, string, or date character vector. The `CapSettle` date cannot be earlier than the `ZeroCurve` valuation date.

To support existing code, `capvolstrip` also accepts serial date numbers as inputs, but they are not recommended.

`CapMaturity` — Cap maturity dates

datetime array | string array | date character vector

Cap maturity dates, specified using a `NCap`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `capvolstrip` also accepts serial date numbers as inputs, but they are not recommended.

`CapVolatility` — Flat cap volatilities

vector of positive decimals

Flat cap volatilities, specified as an `NCap`-by-`1` vector of positive decimals.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[CapletVols,CapletPaymentDates,CapStrikes] = capvolstrip(ZeroCurve,CapSettle,CapMaturity,CapVolatility,'Strike',.2)`

Strike — Cap strike rate
If not specified, all caps are at-the-money and the function computes the ATM strike for each cap maturing on each caplet payment date (default) | scalar decimal | vector

Cap strike rate, specified as the comma-separated pair consisting of `'Strike'` and a scalar decimal value or an `NCapletVols`-by-1 vector. Use `Strike` as a scalar to specify a single strike that applies equally to all caps. Or, specify an `NCapletVols`-by-1 vector of strikes for the caps.

Data Types: `double`

CapletDates — Caplet reset and payment dates
if not specified, the default is to automatically generate periodic caplet dates (default) | datetime array | string array | date character vector

Caplet reset and payment dates, specified as the comma-separated pair consisting of `'CapletDates'` and an `NCapletDates`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `capvolstrip` also accepts serial date numbers as inputs, but they are not recommended.

Use `CapletDates` to manually specify all caplet reset and payment dates. For example, some date intervals may be quarterly, while others may be semiannual. All dates must be later than `CapSettle` and cannot be later than the last `CapMaturity` date. Dates are adjusted according to the `BusDayConvention` and `Holidays` inputs.

If `CapletDates` is not specified, the default is to automatically generate periodic caplet dates after `CapSettle` based on the last `CapMaturity` date as the reference date, using the following optional inputs: `Reset`, `EndMonthRule`, `BusDayConvention`, and `Holidays`.

Reset — Frequency of periodic payments per year within a cap
2 (default) | positive scalar integer with values 1,2, 3, 4, 6, or 12

Frequency of periodic payments per year within a cap, specified as the comma-separated pair consisting of `'Reset'` and a positive scalar integer with values 1,2, 3, 4, 6, or 12.

Note If you specify `CapletDates`, the function ignores the input for `Reset`.

Data Types: `double`

EndMonthRule — End-of-month rule flag for generating caplet dates
1 (in effect) (default) | scalar nonnegative integer `[0,1]`

End-of-month rule flag for generating caplet dates, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar nonnegative integer [0, 1].

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

BusinessDayConvention — Business day conventions
`'modifiedfollow'` (default) | character vector with values `'actual'`, `'follow'`, `'modifiedfollow'`, `'previous'`, `'modifiedprevious'`

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector. Use this argument to specify how the function treats non-business days, which are days on which businesses are not open (such as weekends and statutory holidays).

- `'actual'` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `'follow'` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `'modifiedfollow'` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- `'previous'` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `'modifiedprevious'` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | vector of MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and `NHolidays`-by-`1` vector of MATLAB dates.

Data Types: `datetime`

`ProjectionCurve` — Rate curve for computing underlying forward rates
if not specified, the default is to use the `ZeroCurve` input for computing the underlying forward rates (default) | `RateSpec` object | `IRDatCurve` object

Rate curve for computing underlying forward rates, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a `RateSpec` object or `IRDatCurve` object. For more information on creating a `RateSpec`, see `intenvset`. For more information on creating an `IRDataCurve` object, see `IRDataCurve`.

Data Types: `struct`

`MaturityInterpMethod` — Method for interpolating the cap volatilities on each caplet maturity date before stripping the caplet volatilities
`'linear'` (default) | character vector with values: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'spline'`, `'pchip'`

Method for interpolating the cap volatilities on each caplet maturity date before stripping the caplet volatilities, specified as the comma-separated pair consisting of `'MaturityInterpMethod'` and a character vector with values: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'spline'`, or `'pchip'`.

- `'linear'` — Linear interpolation. The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.
- `'nearest'` — Nearest neighbor interpolation. The interpolated value at a query point is the value at the nearest sample grid point.

- `'next'` — Next neighbor interpolation. The interpolated value at a query point is the value at the next sample grid point.
- `'previous'` — Previous neighbor interpolation. The interpolated value at a query point is the value at the previous sample grid point.
- `'spline'` — Spline interpolation using not-a-knot end conditions. The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension.
- `'pchip'` — Shape-preserving piecewise cubic interpolation. The interpolated value at a query point is based on a shape-preserving piecewise cubic interpolation of the values at neighboring grid points.

For more information on interpolation methods, see `interp1`.

Note The function uses constant extrapolation to calculate volatilities falling outside the range of user-supplied data.

Data Types: `char`

`Limit` — Upper bound of implied volatility search interval
`10` (or 1000% per annum) (default) | positive scalar decimal

Upper bound of implied volatility search interval, specified as the comma-separated pair consisting of `'Limit'` and a positive scalar decimal.

Data Types: `double`

`Tolerance` — Implied volatility search termination tolerance
`1e-5` (default) | positive numeric scalar

Implied volatility search termination tolerance, specified as the comma-separated pair consisting of `'Tolerance'` and a positive numeric scalar.

Data Types: `double`

`OmitFirstCaplet` — Flag to omit the first caplet payment in the caps
`true` (always omit the first caplet) (default) | logical

Flag to omit the first caplet payment in the caps, specified as the comma-separated pair consisting of `'OmitFirstCaplet'` and a scalar logical.

If the caps are spot-starting, the first caplet payment is omitted. If the caps are forward-starting, the first caplet payment is included. Regardless of the status of the caps, if you set this logical to `false`, then the function includes the first caplet payment.

In general, “spot lag” is the delay between the fixing date and the effective date for LIBOR-like indices. "Spot lag" determines whether a cap is spot-starting or forward-starting (Corb, 2012). Caps are considered to be spot-starting if they settle within “spot lag” business days after the valuation date. Those that settle later are considered to be forward-starting. The first caplet is omitted if caps are spot-starting, while it is included if they are forward-starting (Tuckman, 2012).

Data Types: `logical`

`Shift` — Shift in decimals for shifted SABR model

`0` (no shift) (default) | positive scalar decimal

Shift in decimals for the shifted SABR model (to be used with the Shifted Black model), specified as the comma-separated pair consisting of `'Shift'` and a positive scalar decimal value. Set this parameter to a positive shift in decimals to add a positive shift to the forward rate and strike, which effectively sets a negative lower bound for the forward rate and strike. For example, a `Shift` value of 0.01 is equal to a 1% shift.

Data Types: `double`

`Model` — Model used for implied volatility

`'lognormal'` (default) | character vector with value of `'lognormal'` or `'normal'` | string scalar with value of `"lognormal"` or `"normal"`

Model used for the implied volatility calculation, specified as the comma-separated pair consisting of `'Model'` and a scalar character vector or string scalar with one of the following values:

- `'lognormal'` - Implied Black (no shift) or Shifted Black volatility.
- `'normal'` - Implied Normal (Bachelier) volatility. If you specify `'normal'`, `Shift` must be zero.

The `capvolstrip` function supports three volatility types.

'Model' Value	'Shift' Value	Volatility Type
`'lognormal'`	`Shift = 0`	Black
`'lognormal'`	`Shift > 0`	Shifted Black
`'normal'`	`Shift = 0`	Normal (Bachelier)

Data Types: `char` | `string`

Output Arguments

`CapletVols` — Stripped caplet volatilities

vector of decimals

Stripped caplet volatilities, returned as an `NCapletVols`-by-`1` vector of decimals.

Note `capvolstrip` can output `NaN`s for some caplet volatilities. You might encounter this output if no volatility matches the caplet price implied by the user-supplied cap data.

`CapletPaymentDates` — Payment dates

vector of date numbers

Payment dates (in date numbers), returned as an `NCapletVols`-by-`1` vector of date numbers corresponding to `CapletVols`.

`CapStrikes` — Cap strikes

vector of decimals

Cap strikes, returned as an `NCapletVols`-by-`1` vector of strikes in decimals for caps maturing on the corresponding `CapletPaymentDates`. `CapStrikes` are the same as the strikes of the corresponding caplets that have been stripped.

Limitations

When bootstrapping the caplet volatilities from ATM caps, the function reuses the caplet volatilities stripped from the shorter maturity caps in the longer maturity caps without adjusting for the difference in strike. `capvolstrip` follows the simplified approach described in Gatarek, 2006.

More About

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:

$\max(CurrentRate - CapRate, 0)$

For more information, see “Cap” on page 2-12.

At-The-Money

A cap or floor is at-the-money (ATM) if its strike is equal to the forward swap rate.

The forward swap rate is the fixed rate of a swap that makes the present value of the floating leg equal to that of the fixed leg. In comparison, a caplet or floorlet is ATM if its strike is equal to the forward rate (not the forward swap rate). In general (except over a single period), the forward rate is not equal to the forward swap rate. So, to be precise, the individual caplets in an ATM cap have slightly different moneyness and are only approximately ATM (Alexander, 2003).

In addition, the swap rate changes with swap maturity. Similarly, the ATM cap strike also changes with cap maturity, so the ATM cap strikes are computed for each cap maturity before stripping the caplet volatilities. As a result, when stripping the caplet volatilities from the ATM caps with increasing maturities, the ATM strikes of consecutive caps are different.

Version History

Introduced in R2016a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `capvolstrip` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Alexander, C. "Common Correlation and Calibrating the Lognormal Forward Rate Model." *Wilmott Magazine*, 2003.

[2] Corb, H. *Interest Rate Swaps and Other Derivatives.* Columbia Business School Publishing, 2012.

[3] Gatarek, D., P. Bachert, and R. Maksymiuk. *The LIBOR Market Model in Practice.* Chichester, UK: Wiley, 2006.

[4] Tuckman, B., and Serrat, A. *Fixed Income Securities: Tools for Today's Markets.* Hoboken, NJ: Wiley, 2012.

See Also

`interp1` | `intenvset` | `floorvolstrip` | `capbyblk` | `capbynormal`

Topics

“Price Swaptions with Negative Strikes Using the Shifted SABR Model” on page 2-26
“Cap” on page 2-12
“Work with Negative Interest Rates Using Functions” on page 2-18

External Websites

How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

cashbybls

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

Syntax

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

Description

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

Note Alternatively, you can use the `Binary` object to price digital options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Compute Cash-or-Nothing Option Prices Using the Black-Scholes Option Pricing Model

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. First, create the `RateSpec`:

```
Settle = datetime(2008,1,1);
Maturity = datetime(2008,10,1);
Rates = 0.045;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9668
            Rates: 0.0450
         EndTimes: 0.7500
       StartTimes: 0
         EndDates: 733682
       StartDates: 733408
    ValuationDate: 733408
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec`.

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

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2500
         AssetPrice: 110
       DividendType: {'continuous'}
    DividendAmounts: 0.0450
    ExDividendDates: []
```

Define the call and put options.

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

Calculate the prices.

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

```
Pcon = 2×1

    7.6716
    1.9965
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cashbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cashbybls` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as an `NINST`-by-`1` vector.

Data Types: `char` | `cell`

`Strike` — Strike price value
vector

Strike price value, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Payoff` — Payoff values
vector

Payoff values (or the amount to be paid at expiration), specified as an `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected prices for cash-or-nothing option
vector

Expected prices for cash-or-nothing option, returned as a `NINST`-by-`1` vector.

Version History

Introduced in R2009a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `cashbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`assetbybls` | `cashsensbybls` | `gapbybls` | `supersharebybls` | `Binary`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing Using the Black-Scholes Model” on page 3-82
“Supported Equity Derivative Functions” on page 3-19

cashsensbybls

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

Syntax

```
PriceSens = cashsensbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,
Payoff)
PriceSens = cashsensbybls( ___ ,Name,Value)
```

Description

`PriceSens = cashsensbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike, Payoff)` computes the price or sensitivities for cash-or-nothing European digital options using the Black-Scholes option pricing model.

Note Alternatively, you can use the `Binary` object to calculate price or sensitivities for digital options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = cashsensbybls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute Cash-or-Nothing Option Prices and Sensitivities Using the Black-Scholes Option Pricing Model

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 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 and sensitivity of the call and put cash-or-nothing options on the futures contract. First, create the `RateSpec`:

```
Settle = datetime(2008,1,1);
Maturity = datetime(2008,10,1);
Rates = 0.045;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9668
            Rates: 0.0450
         EndTimes: 0.7500
       StartTimes: 0
```

```
         EndDates: 733682
       StartDates: 733408
    ValuationDate: 733408
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec`.

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

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2500
         AssetPrice: 110
       DividendType: {'continuous'}
    DividendAmounts: 0.0450
    ExDividendDates: []
```

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 = 2×1

   -0.0050
    0.0050


Theta = 2×1

   -2.2489
    1.8139


Price = 2×1

    7.6716
    1.9965
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset
structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cashsensbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cashsensbybls` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as an `NINST`-by-`1` vector.

Data Types: `char` | `cell`

`Strike` — Strike price value
vector

Strike price value, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Payoff` — Payoff values
vector

Payoff values (or the amount to be paid at expiration), specified as an `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

`OutSpec` — Define outputs

`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities for cash-or-nothing option

vector

Expected prices or sensitivities (defined using `OutSpec`) for cash-or-nothing option, returned as a `NINST`-by-`1` vector.

Version History

Introduced in R2009a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `cashsensbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cashbybls` | `Binary`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing Using the Black-Scholes Model” on page 3-82
“Supported Equity Derivative Functions” on page 3-19

cbondbycrr

Price convertible bonds from CRR binomial tree

Syntax

```
Price = cbondbycrr(CRRTree,CouponRate,Settle,Maturity,ConvRatio)
[Price,PriceTree] = cbondbycrr(CRRTree,CouponRate,Settle,Maturity,ConvRatio)
[Price,PriceTree,EquityTree,DebtTree] = cbondbycrr( ___ ,Name,Value)
```

Description

`Price = cbondbycrr(CRRTree,CouponRate,Settle,Maturity,ConvRatio)` prices convertible bonds from a CRR binomial tree using the Tsiveriotis and Fernandes method.

`[Price,PriceTree] = cbondbycrr(CRRTree,CouponRate,Settle,Maturity,ConvRatio)` prices convertible bonds from a CRR binomial tree using the Tsiveriotis and Fernandes method.

`[Price,PriceTree,EquityTree,DebtTree] = cbondbycrr( ___ ,Name,Value)` prices convertible bonds from a CRR binomial tree using a credit spread or incorporating the risk of bond default.

To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional name-value pair input argument `Spread`. To incorporate default risk into the algorithm, specify the optional name-value pair input arguments `DefaultProbability` and `RecoveryRate`.

Examples

Price Convertible Bond Using a CRR Tree

Price a convertible bond using the following data for the interest-rate term structure:

```
StartDates = datetime(2014,1,1);
EndDates = datetime(2015,1,1);
Rates = 0.1;
Basis = 1;
```

Create the `RateSpec` and `StockSpec`.

```
Sigma = 0.3;
Price = 50;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9048
            Rates: 0.1000
         EndTimes: 1
```

```
       StartTimes: 0
         EndDates: 735965
       StartDates: 735600
    ValuationDate: 735600
            Basis: 1
     EndMonthRule: 1
```

```
StockSpec = stockspec(Sigma,Price)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3000
         AssetPrice: 50
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Create the CRR tree for the equity.

```
Settle = datetime(2014,1,1);
Maturity = datetime(2014,10,1);
NumSteps = 3;
TimeSpec = crrtimespec(Settle,Maturity,NumSteps);
CRRT = crrtree(StockSpec,RateSpec,TimeSpec)
```

```
CRRT = struct with fields:
       FinObj: 'BinStockTree'
       Method: 'CRR'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.2491 0.4982 0.7473]
         dObs: [735600 735691 735782 735873]
        STree: {[50]  [58.0757 43.0472]  [67.4558 50.0000 37.0613]  [78.3509 ... ]}
      UpProbs: [0.5465 0.5465 0.5465]
```

Define and price the convertible bond.

```
CouponRate = 0;
Period = 1;
ConvRatio = 2;
CallExDates = datetime(2014,10,1);
CallStrike = 115;
AmericanCall = 1;
Spread = 0.05;

[Price,PriceTree,EqtTree,DbtTree] = cbondbycrr(CRRT,CouponRate,Settle,Maturity,ConvRatio,...
'Period',Period,'Spread',Spread,'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall',
```

```
Price = 104.9490
```

```
PriceTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {1x4 cell}
      tObs: [0 0.2491 0.4982 0.7473]
```

```
      dObs: [735600 735691 735782 735873]


EqtTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {[76.5211]  [116.1515 33.0103]  [134.9117 61.9209 0]  [156.7019 ... ]}
      tObs: [0 0.2491 0.4982 0.7473]
      dObs: [735600 735691 735782 735873]


DbtTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {[28.4278]  [0 65.0790]  [0 43.6821 96.3327]  [0 0 100.0000 100.0000]}
      tObs: [0 0.2491 0.4982 0.7473]
      dObs: [735600 735691 735782 735873]
```

Price a Convertible Bond Using a CRR Tree and Incorporate Default Risk Using `DefaultProbability` and `RecoveryRate`

Create the interest-rate term structure `RateSpec`.

```
StartDates = 'Jan-1-2014';
EndDates = 'Jan-1-2016';
Rates = 0.025;
Basis = 1;
RateSpec = intenvset('ValuationDate',StartDates,'StartDates',...
StartDates,'EndDates',EndDates,'Rates',Rates,'Compounding',-1,'Basis',Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0250
         EndTimes: 2
       StartTimes: 0
         EndDates: 736330
       StartDates: 735600
    ValuationDate: 735600
            Basis: 1
     EndMonthRule: 1
```

Create the `StockSpec`.

```
AssetPrice = 110;
Sigma = 0.22;
Div = 0.02;
StockSpec = stockspec(Sigma,AssetPrice,'continuous',Div)
```

```
StockSpec = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.2200
          AssetPrice: 110
       DividendType: {'continuous'}
    DividendAmounts: 0.0200
```

```
    ExDividendDates: []
```

Create the CRR tree for the equity.

```
Settle = '1-Jan-2014';
Maturity = '1-Oct-2014';
NumSteps = 3;
TimeSpec = crrtimespec(Settle,Maturity,NumSteps);
CRRT = crrtree(StockSpec,RateSpec,TimeSpec)
```

```
CRRT = struct with fields:
       FinObj: 'BinStockTree'
       Method: 'CRR'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.2491 0.4982 0.7473]
         dObs: [735600 735691 735782 735873]
        STree: {1x4 cell}
      UpProbs: [0.4782 0.4782 0.4782]
```

Define and price the convertible bond using the optional `DefaultProbability` and `RecoveryRate` arguments.

```
CouponRate = 0;
Period = 1;
ConvRatio = 2;
CallExDates = '1-Oct-2014';
CallStrike = 115;
AmericanCall = 1;
DefaultProbability = .30;
RecoveryRate = .82;

[Price,PriceTree,EqtTree,DbtTree] = cbondbycrr(CRRT,CouponRate,Settle,Maturity,ConvRatio,...
'Period',Period,'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall',AmericanCall,...
'DefaultProbability',DefaultProbability,'RecoveryRate',RecoveryRate)
```

```
Price = 220
```

```
PriceTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {[220]  [245.5317 197.1233]  [274.0263 220.0000 176.6254]  [1x4 double]}
      tObs: [0 0.2491 0.4982 0.7473]
      dObs: [735600 735691 735782 735873]


EqtTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {[220]  [245.5317 197.1233]  [274.0263 220.0000 176.6254]  [1x4 double]}
      tObs: [0 0.2491 0.4982 0.7473]
      dObs: [735600 735691 735782 735873]


DbtTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {[0]  [0 0]  [0 0 0]  [0 0 0 0]}
      tObs: [0 0.2491 0.4982 0.7473]
```

```
dObs: [735600 735691 735782 735873]
```

Input Arguments

`CRRTree` — Stock tree structure

structure

Stock tree structure, specified by using `crrtree`.

Data Types: `struct`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every convertible bond is set to the `ValuationDate` of the CRR stock tree. The bond argument, `Settle`, is ignored.

To support existing code, `cbondbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`ConvRatio` — Number of shares convertible to one bond

nonnegative number

Number of shares convertible to one bond, specified as an `NINST`-by-`1` with a nonnegative number.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree,EquityTree,DebtTree] = cbondbycrr(CRRT,CouponRate,Settle, Maturity, ConvRatio,'Spread',Spread,'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall',1)`

Spread — Number of basis points over the reference rate

`0` (default) | vector

Number of basis points over the reference rate, specified as the comma-separated pair consisting of `'Spread'` and a `NINST`-by-`1` vector.

Note To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument `Spread`. To incorporate default risk into the algorithm, specify the optional input arguments `DefaultProbability` and `RecoveryRate`. Do not use `Spread` with `DefaultProbability` and `RecoveryRate`.

Data Types: `double`

Period — Coupons per year

2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector.

Data Types: `double`

IssueDate — Bond issue date

datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbycrr` also accepts serial date numbers as inputs, but they are not recommended.

FirstCouponDate — Irregular first coupon date

datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

Face value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector of nonnegative face values or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is the associated face value. The date indicates the last day that the face value is valid.

Data Types: `cell` | `double`

`CallStrike` — Call strike price for European, Bermuda, or American option
nonnegative integer | vector of nonnegative integers

Call strike price for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'CallStrike'` and one of the following values:

- For a European call option — `NINST`-by-`1` vector of nonnegative integers
- For a Bermuda call option — `NINST`-by-`NSTRIKES` matrix of call strike price values, where each row is the schedule for one call option. If a call option has fewer than `NSTRIKES` exercise opportunities, the end of the row is padded with `NaN`s.
- For an American call option — `NINST`-by-`1` vector of strike price values for each option.

Data Types: `single` | `double`

`CallExDates` — Call exercise date for European, Bermuda, or American option
datetime array | string array | date character vector

Call exercise date for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'CallExDates'` and a datetime array, string array, or date character vectors for the following values:

- For a European option — `NINST`-by-`1` vector of date character vectors.
- For a Bermuda option — `NINST`-by-`NSTRIKES` matrix of exercise dates, where each row is the schedule for one option. For a European option, there is only one `CallExDate` on the option expiry date.
- For an American option — `NINST`-by-`1` or `NINST`-by-`2` matrix of exercise date boundaries. For each instrument, the call option can be exercised on any tree date between or including the pair of dates on that row. If `CallExDates` is `NINST`-by-`1`, the option can be exercised between the `ValuationDate` of the CRR stock tree and the single listed `CallExDate`.

To support existing code, `cbondbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanCall` — Call option type indicator
`0` if `AmericanCall` is `NaN` or not entered (default) | scalar | vector of positive integers`[0,1]`

Call option type, specified as the comma-separated pair consisting of `'AmericanCall'` and a `NINST`-by-`1` vector of positive integer flags with values `0` or `1`.

- For a European or Bermuda option — `AmericanCall` is `0` for each European or Bermuda option.
- For an American option — `AmericanCall` is `1` for each American option. The `AmericanCall` argument is required to invoke American exercise rules.

Data Types: `single` | `double`

`PutStrike` — Put strike values for European, Bermuda, or American option
positive integer | vector of positive integers

Put strike values for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'PutStrike'` and one of the following values:

- For a European put option — `NINST`-by-`1` vector of nonnegative integers
- For a Bermuda put option — `NINST`-by-`NSTRIKES` matrix of strike price values where each row is the schedule for one option. If a put option has fewer than `NSTRIKES` exercise opportunities, the end of the row is padded with `NaN`s.
- For an American put option — `NINST`-by-`1` vector of strike price values for each option.

Data Types: `single` | `double`

`PutExDates` — Put exercise date for European, Bermuda, or American option
datetime array | string array | date character vector

Put exercise date for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'PutExDates'` and a datetime array, string array, or date character vectors for the following values:

- For a European option — `NINST`-by-`1` vector of date character vectors.
- For a Bermuda option — `NINST`-by-`NSTRIKES` matrix of exercise dates where each row is the schedule for one option. For a European option, there is only one `PutExDate` on the option expiry date.
- For an American option — `NINST`-by-`1` or `NINST`-by-`2` matrix of exercise date boundaries. For each instrument, the put option can be exercised on any tree date between or including the pair of dates on that row. If `PutExDates` is `NINST`-by-`1`, the put option can be exercised between the `ValuationDate` of the CRR stock tree and the single listed `PutExDate`.

To support existing code, `cbondbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanPut` — Put option type indicator
`0` if `AmericanPut` is `NaN` or not entered (default) | positive integer `[0,1]` | vector of positive integers `[0,1]`

Put option type, specified as the comma-separated pair consisting of `'AmericanPut'` and a `NINST`-by-`1` vector of positive integer flags with values `0` or `1`.

- For a European or Bermuda option — `AmericanPut` is `0` for each European or Bermuda option.
- For an American option — `AmericanPut` is `1` for each American option. The `AmericanPut` argument is required to invoke American exercise rules.

Data Types: `double`

`ConvDates` — Convertible dates
`MaturityDate` (default) | datetime array | string array | date character vector

Convertible dates, specified as the comma-separated pair consisting of `'ConvDates'` and a `NINST`-by-`1` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors. If `ConvDates` is not specified, the bond is always convertible until maturity.

To support existing code, `cbondbycrr` also accepts serial date numbers as inputs, but they are not recommended.

For each instrument, the bond can be converted on any tree date between or including the pair of dates on that row.

If `ConvDates` is `NINST`-by-`1`, the bond can be converted between the `ValuationDate` of the CRR stock tree and the single listed `ConvDates`.

`DefaultProbability` — Annual probability of default rate
`0` (default) | nonnegative decimal

Annual probability of default rate, specified as the comma-separated pair consisting of `'DefaultProbability'` and a `NINST`-by-`1` nonnegative decimal value.

Note To incorporate default risk into the algorithm, specify the optional input arguments `DefaultProbability` and `RecoveryRate`. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument `Spread`. Do not use `DefaultProbability` and `RecoveryRate` with `Spread`.

Data Types: `double`

`RecoveryRate` — Recovery rate
`1` (default) | nonnegative decimal

Recovery rate, specified as the comma-separated pair consisting of `'RecoveryRate'` and a `NINST`-by-`1` nonnegative decimal.

Note To incorporate default risk into the algorithm, specify the optional input arguments `DefaultProbability` and `RecoveryRate`. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument `Spread`. Do not use `DefaultProbability` and `RecoveryRate` with `Spread`.

Data Types: `double`

Output Arguments

`Price` — Expected price at time `0`
array

Expected price at time `0`, returned as an `NINST`-by-`1` array.

`PriceTree` — Structure with vector of convertible bond prices at each node
tree structure

Structure with a vector of convertible bond prices at each node, returned as a tree structure.

`EquityTree` — Structure with vector of convertible bond equity component at each node
tree structure

Structure with a vector of convertible bond equity component at each node, returned as a tree structure.

`DebtTree` — Structure with vector of convertible bond debt component at each node
tree structure

Structure with a vector of convertible bond debt component at each node, returned as a tree structure.

More About

Callable Convertible

A convertible bond that is callable by the issuer. The issuer of the bond forces conversion, removing the advantage that conversion is at the discretion of the bondholder.

Upon call, the bondholder can either convert the bond or redeem at the call price. This option enables the issuer to control the price of the convertible bond and, if necessary, refinance the debt with a new cheaper one.

Puttable Convertible

A convertible bond with a put feature allows the bondholder to sell back the bond at a premium on a specific date.

This option protects the holder against rising interest rates by reducing the year to maturity.

Algorithms

`cbondbycrr`, `cbondbyeqp`, `cbondbyitt`, and `cbondbysttr`eturn price information in the form of a price vector and a price tree. These functions implement the risk in the form of either a credit spread or incorporating the risk of bond default. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional name-value pair argument `Spread`. To incorporate default risk into the algorithm, specify the optional name-value pair arguments `DefaultProbability` and `RecoveryRate`.

Version History

Introduced in R2015a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `cbondbycrr` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Tsiveriotis, K., and C. Fernandes. “Valuing Convertible Bonds with Credit Risk.” *Journal of Fixed Income.* Vol. 8, 1998, pp. 95–102.

[2] Hull, J. *Options, Futures and Other Derivatives.* Fourth Edition. Prentice Hall, 2000, pp. 646–649.

See Also

`crrtree` | `cbondbyeqp` | `instcbond` | `intenvset` | `stockspec` | `instadd` | `instdisp` | `eqpprice` | `crrsens` | `eqpsens`

Topics

“Convertible Bond” on page 2-4

cbondbyeqp

Price convertible bonds from EQP binomial tree

Syntax

```
Price = cbondbyeqp(EQPTree,CouponRate,Settle,Maturity,ConvRatio)
[Price,PriceTree] = cbondbyeqp(EQPTree,CouponRate,Settle,Maturity,ConvRatio)
[Price,PriceTree,EquityTree,DebtTree] = cbondbyeqp( ___ ,Name,Value)
```

Description

`Price = cbondbyeqp(EQPTree,CouponRate,Settle,Maturity,ConvRatio)` prices convertible bonds from an EQP binomial tree using the Tsiveriotis and Fernandes method.

`[Price,PriceTree] = cbondbyeqp(EQPTree,CouponRate,Settle,Maturity,ConvRatio)` prices convertible bonds from an EQP binomial tree using the Tsiveriotis and Fernandes method.

`[Price,PriceTree,EquityTree,DebtTree] = cbondbyeqp( ___ ,Name,Value)` prices convertible bonds from an EQP binomial tree using a credit spread or incorporating the risk of bond default.

To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional name-value pair input argument `Spread`. To incorporate default risk into the algorithm, specify the optional name-value pair input arguments `DefaultProbability` and `RecoveryRate`.

Examples

Price Convertible Bond Using an EQP Tree

Create the interest-rate term structure `RateSpec`.

```
StartDates = datetime(2014,1,1);
EndDates = 'Jan-1-2016';
Rates = 0.025;
Basis = 1;
RateSpec = intenvset('ValuationDate',StartDates,'StartDates',StartDates,'EndDates',EndDates,...
'Rates',Rates,'Compounding',-1,'Basis',Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0250
         EndTimes: 2
       StartTimes: 0
         EndDates: 736330
       StartDates: 735600
    ValuationDate: 735600
            Basis: 1
     EndMonthRule: 1
```

Create the `StockSpec`.

```
AssetPrice = 110;
Sigma = 0.22;
Div = 0.02;
StockSpec = stockspec(Sigma,AssetPrice,'continuous',Div)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2200
         AssetPrice: 110
       DividendType: {'continuous'}
    DividendAmounts: 0.0200
    ExDividendDates: []
```

Create the EQP tree for the equity.

```
NumSteps = 6;
TimeSpec = eqptimespec(StartDates,EndDates,NumSteps);
EQPTree = eqptree(StockSpec,RateSpec,TimeSpec)
```

```
EQPTree = struct with fields:
       FinObj: 'BinStockTree'
       Method: 'EQP'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.3333 0.6667 1 1.3333 1.6667 2]
         dObs: [735600 735721 735843 735965 736086 736208 736330]
        STree: {1x7 cell}
      UpProbs: [0.5000 0.5000 0.5000 0.5000 0.5000 0.5000]
```

Define the convertible bond. The convertible bond can be called starting on Jan 1, 2015 with a strike price of 125.

```
Settle = datetime(2014,1,1);
Maturity = datetime(2016,1,1);
CouponRate = 0.03;
CallStrike = 125;
Period = 1;
CallExDates = [datetime(2015,1,1) ; datetime(2016,1,1)];
ConvRatio = 1.5;
```

Price the convertible bond.

```
Spread = 0.045;

[Price,PriceTree,EqtTre,DbtTree] = cbondbyeqp(EQPTree,CouponRate,Settle,...
Maturity,ConvRatio,'Period',Period,'Spread',Spread,'CallExDates',...
CallExDates,'CallStrike',CallStrike,'AmericanCall',1)
```

```
Price = 2×1

   165
   165
```

```
PriceTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {1x7 cell}
      tObs: [0 0.3333 0.6667 1 1.3333 1.6667 2]
      dObs: [735600 735721 735843 735965 736086 736208 736330]


EqtTre = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {1x7 cell}
      tObs: [0 0.3333 0.6667 1 1.3333 1.6667 2]
      dObs: [735600 735721 735843 735965 736086 736208 736330]


DbtTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {1x7 cell}
      tObs: [0 0.3333 0.6667 1 1.3333 1.6667 2]
      dObs: [735600 735721 735843 735965 736086 736208 736330]
```

Price a Convertible Bond Using an EQP Tree and Incorporate Default Risk Using `DefaultProbability` and `RecoveryRate`

Create the interest-rate term structure `RateSpec`.

```
StartDates = datetime(2014,1,1);
EndDates = datetime(2016,1,1);
Rates = 0.025;
Basis = 1;
RateSpec = intenvset('ValuationDate',StartDates,'StartDates',...
StartDates,'EndDates',EndDates,'Rates',Rates,'Compounding',-1,'Basis',Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0250
         EndTimes: 2
       StartTimes: 0
         EndDates: 736330
       StartDates: 735600
    ValuationDate: 735600
            Basis: 1
     EndMonthRule: 1
```

Create the `StockSpec`.

```
AssetPrice = 110;
Sigma = 0.22;
Div = 0.02;
StockSpec = stockspec(Sigma,AssetPrice,'continuous',Div)
```

```
StockSpec = struct with fields:
            FinObj: 'StockSpec'
```

```
             Sigma: 0.2200
         AssetPrice: 110
       DividendType: {'continuous'}
    DividendAmounts: 0.0200
    ExDividendDates: []
```

Create the EQP tree for the equity.

```
NumSteps = 6;
TimeSpec = eqptimespec(StartDates,EndDates,NumSteps);
EQPTree = eqptree(StockSpec,RateSpec,TimeSpec)
```

```
EQPTree = struct with fields:
       FinObj: 'BinStockTree'
       Method: 'EQP'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.3333 0.6667 1 1.3333 1.6667 2]
         dObs: [735600 735721 735843 735965 736086 736208 736330]
        STree: {1x7 cell}
      UpProbs: [0.5000 0.5000 0.5000 0.5000 0.5000 0.5000]
```

Define and price the convertible bond using the optional `DefaultProbability` and `RecoveryRate` arguments.

```
Settle = datetime(2014,1,1);
Maturity = datetime(2016,1,1);
CouponRate = 0.03;
CallStrike = 125;
Period = 1;
CallExDates = [datetime(2015,1,1) ; datetime(2016,1,1)];
ConvRatio = 1.5;
DefaultProbability = .30;
RecoveryRate = .82;

[Price,PriceTree,EqtTre,DbtTree] = cbondbyeqp(EQPTree,CouponRate,Settle,...
Maturity,ConvRatio,'Period',Period,'CallExDates',...
CallExDates,'CallStrike',CallStrike,'AmericanCall',1,...
'DefaultProbability',DefaultProbability,'RecoveryRate',RecoveryRate)
```

```
Price = 2×1

   165
   165


PriceTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {1x7 cell}
      tObs: [0 0.3333 0.6667 1 1.3333 1.6667 2]
      dObs: [735600 735721 735843 735965 736086 736208 736330]


EqtTre = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {1x7 cell}
```

```
      tObs: [0 0.3333 0.6667 1 1.3333 1.6667 2]
      dObs: [735600 735721 735843 735965 736086 736208 736330]


DbtTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {1x7 cell}
      tObs: [0 0.3333 0.6667 1 1.3333 1.6667 2]
      dObs: [735600 735721 735843 735965 736086 736208 736330]
```

Input Arguments

`EQPTree` — Stock tree structure
structure

Stock tree structure, specified by using `eqptree`.

Data Types: `struct`

`CouponRate` — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every convertible bond is set to the `ValuationDate` of the EQP stock tree. The bond argument, `Settle`, is ignored.

To support existing code, `cbondbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`ConvRatio` — Number of shares convertible to one bond
nonnegative number

Number of shares convertible to one bond, specified as an NINST-by-1 with a nonnegative number.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree,EquityTree,DebtTree] = cbondbyeqp(EQPT,CouponRate,Settle, Maturity, ConvRatio,'Spread',Spread,'CallExDates',CallExDates, 'CallStrike',CallStrike,'AmericanCall',1)

Spread — Number of basis points over the reference rate

0 (default) | vector

Number of basis points over the reference rate, specified as the comma-separated pair consisting of 'Spread' and a NINST-by-1 vector.

Note To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument Spread. To incorporate default risk into the algorithm, specify the optional input arguments DefaultProbability and RecoveryRate. Do not use Spread with DefaultProbability and RecoveryRate.

Data Types: double

Period — Coupons per year

2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of 'Period' and a NINST-by-1 vector.

Data Types: double

IssueDate — Bond issue date

datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of 'IssueDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cbondbyeqp also accepts serial date numbers as inputs, but they are not recommended.

FirstCouponDate — Irregular first coupon date

datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

Face value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector of nonnegative face values or an `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is the associated face value. The date indicates the last day that the face value is valid.

Data Types: `cell` | `double`

`CallStrike` — Call strike price for European, Bermuda, or American option
nonnegative integer | vector of nonnegative integers

Call strike price for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'CallStrike'` and one of the following values:

- For a European call option — `NINST`-by-`1` vector of nonnegative integers
- For a Bermuda call option — `NINST`-by-`NSTRIKES` matrix of strike price values, where each row is the schedule for one call option. If an option has fewer than `NSTRIKES` exercise opportunities, the end of the row is padded with `NaN`s.
- For an American call option — `NINST`-by-`1` vector of strike price values for each call option.

Data Types: `single` | `double`

`CallExDates` — Call exercise date for European, Bermuda, or American option
datetime array | string array | date character vector

Call exercise date for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'CallExDates'` and a datetime array, string array, or date character vectors for one of the following values:

- For a European option — `NINST`-by-`1` vector of date character vectors.
- For a Bermuda option — `NINST`-by-`NSTRIKES` matrix of exercise dates, where each row is the schedule for one call option. For a European option, there is only one `CallExDate` on the option expiry date.
- For an American option — `NINST`-by-`1` or `NINST`-by-`2` matrix of exercise date boundaries. For each instrument, the call option can be exercised on any tree date between or including the pair of dates on that row. If `CallExDates` is `NINST`-by-`1`, the call option can be exercised between the `ValuationDate` of the EQP stock tree and the single listed `CallExDate`.

To support existing code, `cbondbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanCall` — Call option type indicator
`0` if `AmericanCall` is `NaN` or not entered (default) | scalar | vector of positive integers `[0,1]`

Call option type, specified as the comma-separated pair consisting of `'AmericanCall'` and a `NINST`-by-`1` vector with positive integer flags with values `0` or `1`.

- For a European or Bermuda option — `AmericanCall` is `0` for each European or Bermuda option.
- For an American option — `AmericanCall` is `1` for each American option. The `AmericanCall` argument is required to invoke American exercise rules.

Data Types: `single` | `double`

`PutStrike` — Put strike values for European, Bermuda, or American option
positive integer | vector of positive integers

Put strike values for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'PutStrike'` and one of the following values:

- For a European put option — `NINST`-by-`1` vector of nonnegative integers
- For a Bermuda put option — `NINST`-by-`NSTRIKES` matrix of put strike price values, where each row is the schedule for one put option. If a put option has fewer than `NSTRIKES` exercise opportunities, the end of the row is padded with `NaN`s.
- For an American put option — `NINST`-by-`1` vector of strike price values for each put option.

Data Types: `single` | `double`

`PutExDates` — Put exercise date for European, Bermuda, or American option
datetime array | string array | date character vector

Put exercise date for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'PutExDates'` and a datetime array, string array, or date character vectors for one of the following values:

- For a European option — `NINST`-by-`1` vector of date character vectors.
- For a Bermuda option — `NINST`-by-`NSTRIKES` matrix of exercise dates, where each row is the schedule for one put option. For a European option, there is only one `PutExDate` on the option expiry date.
- For an American option — `NINST`-by-`1` or `NINST`-by-`2` matrix of exercise date boundaries. For each instrument, the put option can be exercised on any tree date between or including the pair of dates on that row. If `PutExDates` is `NINST`-by-`1`, the put option can be exercised between the `ValuationDate` of the EQP stock tree and the single listed `PutExDate`.

  To support existing code, `cbondbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanPut` — Put option type indicator
`0` if `AmericanPut` is `NaN` or not entered (default) | positive integer `[0,1]` | vector of positive integers `[0,1]`

Put option type, specified as the comma-separated pair consisting of `'AmericanPut'` and a `NINST`-by-`1` vector of positive integer flags with values `0` or `1`.

- For a European or Bermuda option — `AmericanPut` is `0` for each European or Bermuda option.

- For an American option — `AmericanPut` is `1` for each American option. The `AmericanPut` argument is required to invoke American exercise rules.

Data Types: `single` | `double`

`ConvDates` — Convertible dates
`MaturityDate` (default) | datetime array | string array | date character vector

Convertible dates, specified as the comma-separated pair consisting of `'ConvDates'` and a `NINST`-by-`1` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors. If `ConvDates` is not specified, the bond is always convertible until maturity.

To support existing code, `cbondbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

For each instrument, the bond can be converted on any tree date between or including the pair of dates on that row.

If `ConvDates` is `NINST`-by-`1`, the bond can be converted between the `ValuationDate` of the EQP stock tree and the single listed `ConvDates`.

`DefaultProbability` — Annual probability of default rate
`0` (default) | nonnegative decimal

Annual probability of default rate, specified as the comma-separated pair consisting of `'DefaultProbability'` and a `NINST`-by-`1` nonnegative decimal value.

Note To incorporate default risk into the algorithm, specify the optional input arguments `DefaultProbability` and `RecoveryRate`. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument `Spread`. Do not use `DefaultProbability` and `RecoveryRate` with `Spread`.

Data Types: `double`

`RecoveryRate` — Recovery rate
`1` (default) | nonnegative decimal

Recovery rate, specified as the comma-separated pair consisting of `'RecoveryRate'` and a `NINST`-by-`1` nonnegative decimal value.

Note To incorporate default risk into the algorithm, specify the optional input arguments `DefaultProbability` and `RecoveryRate`. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument `Spread`. Do not use `DefaultProbability` and `RecoveryRate` with `Spread`.

Data Types: `double`

Output Arguments

`Price` — Expected price at time `0`
array

Expected price at time `0`, returned as an `NINST`-by-`1` array.

`PriceTree` — Structure with vector of convertible bond prices at each node
tree structure

Structure with a vector of convertible bond prices at each node, returned as a tree structure.

`EquityTree` — Structure with vector of convertible bond equity component at each node
tree structure

Structure with a vector of convertible bond equity component at each node, returned as a tree structure.

`DebtTree` — Structure with vector of convertible bond debt component at each node
tree structure

Structure with a vector of convertible bond debt component at each node, returned as a tree structure.

More About

Callable Convertible

A convertible bond that is callable by the issuer. The issuer of the bond forces conversion, removing the advantage that conversion is at the discretion of the bondholder.

Upon call, the bondholder can either convert the bond or redeem at the call price. This option enables the issuer to control the price of the convertible bond and, if necessary, refinance the debt with a new cheaper one.

Puttable Convertible

A convertible bond with a put feature allows the bondholder to sell back the bond at a premium on a specific date.

This option protects the holder against rising interest rates by reducing the year to maturity.

Algorithms

`cbondbycrr`, `cbondbyeqp`, `cbondbyitt`, and `cbondbysttr`eturn price information in the form of a price vector and a price tree. These functions implement the risk in the form of either a credit spread or incorporating the risk of bond default. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional name-value pair argument `Spread`. To incorporate default risk into the algorithm, specify the optional name-value pair arguments `DefaultProbability` and `RecoveryRate`.

Version History

Introduced in R2015a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `cbondbyeqp` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Tsiveriotis, K., and C. Fernandes. “Valuing Convertible Bonds with Credit Risk.” *Journal of Fixed Income.* Vol. 8, 1998, pp. 95–102.

[2] Hull, J. *Options, Futures and Other Derivatives.* Fourth Edition. Prentice Hall, 2000, pp. 646–649.

See Also

`eqptree` | `cbondbycrr` | `instcbond` | `intenvset` | `stockspec` | `instadd` | `instdisp` | `eqpprice` | `crrsens` | `eqpsens`

Topics

“Convertible Bond” on page 2-4

cbondbyitt

Price convertible bonds from ITT trinomial tree

Syntax

```
Price = cbondbyitt(ITTTree,CouponRate,Settle,Maturity,ConvRatio)
[Price,PriceTree] = cbondbyitt(ITTTree,CouponRate,Settle,Maturity,ConvRatio)
[Price,PriceTree,EquityTree,DebtTree] = cbondbyitt( ___ ,Name,Value)
```

Description

`Price = cbondbyitt(ITTTree,CouponRate,Settle,Maturity,ConvRatio)` prices convertible bonds from an ITT trinomial tree using the Tsiveriotis and Fernandes method.

`[Price,PriceTree] = cbondbyitt(ITTTree,CouponRate,Settle,Maturity,ConvRatio)` prices convertible bonds from an ITT trinomial tree using the Tsiveriotis and Fernandes method.

`[Price,PriceTree,EquityTree,DebtTree] = cbondbyitt( ___ ,Name,Value)` prices convertible bonds from an ITT trinomial tree using a credit spread or incorporating the risk of bond default.

To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional name-value pair input argument `Spread`. To incorporate default risk into the algorithm, specify the optional name-value pair input arguments `DefaultProbability` and `RecoveryRate`.

Examples

Price a Convertible Bond Using an ITT Tree

Price a convertible bond using the following data for an `ITTTree` from `deriv.mat`:

```
load deriv.mat
```

Use `cbondbyitt` to price a convertible bond using an ITT trinomial tree.

```
CouponRate = 0.05;
Settle = datetime(2006,1,1);
Maturity = datetime(2008,1,1);
Period = 1;
CallStrike = 65;
CallExDates = datetime(2007,1,1);
ConvRatio = 1;
Spread = 0.015;

[Price,PriceTree,EqtTre,DbtTree] = cbondbyitt(ITTTree,CouponRate,Settle,Maturity,ConvRatio,...
'Period',Period,'Spread',Spread,'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall'
```

```
Price = 58.9170
```

```
PriceTree = struct with fields:
    FinObj: 'TrinPriceTree'
```

```
     PTree: {1x5 cell}
      tObs: [0 1 2 3 4]
      dObs: [732678 733043 733408 733773 734139]


EqtTre = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {[28.0629]  [66.3448 0 0]  [0 0 0 0 0]  [0 0 0 0 0 0 0]  [0 0 0 0 0 ... ]}
      tObs: [0 1 2 3 4]
      dObs: [732678 733043 733408 733773 734139]


DbtTree = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {[30.8540]  [0 65 65]  [105 105 105 105 105]  [0 0 0 0 0 0 0]  [0 0 ... ]}
      tObs: [0 1 2 3 4]
      dObs: [732678 733043 733408 733773 734139]
```

Price a Convertible Bond Using an ITT Tree and Incorporate Default Risk Using `DefaultProbability` and `RecoveryRate`

Price a convertible bond using the following data for an `ITTTree` from `deriv.mat`.

```
load deriv.mat
```

Use `cbondbyitt` to price a convertible bond using an ITT trinomial tree with the optional `DefaultProbability` and `RecoveryRate` arguments.

```
CouponRate = 0.05;
Settle = datetime(2006,1,1);
Maturity = datetime(2008,1,1);
Period = 1;
CallStrike = 65;
CallExDates = datetime(2007,1,1);
ConvRatio = 1;
DefaultProbability = .30;
RecoveryRate = .82;

[Price,PriceTree,EqtTre,DbtTree] = cbondbyitt(ITTTree,CouponRate,Settle,Maturity,ConvRatio,...
'Period',Period,'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall',1,...
'DefaultProbability',DefaultProbability,'RecoveryRate',RecoveryRate)

Price = 50.6487

PriceTree = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {1x5 cell}
      tObs: [0 1 2 3 4]
      dObs: [732678 733043 733408 733773 734139]


EqtTre = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {[20.7895]  [66.3448 0 0]  [0 0 0 0 0]  [0 0 0 0 0 0 0]  [0 0 0 0 0 ... ]}
      tObs: [0 1 2 3 4]
```

```
      dObs: [732678 733043 733408 733773 734139]


DbtTree = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {[29.8591]  [0 65 65]  [105 105 105 105 105]  [0 0 0 0 0 0 0]  [0 0 ... ]}
      tObs: [0 1 2 3 4]
      dObs: [732678 733043 733408 733773 734139]
```

Input Arguments

`ITTTree` — Stock tree structure

structure

Stock tree structure, specified by using `itttree`.

Data Types: `struct`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every convertible bond is set to the `ValuationDate` of the standard trinomial (STT) stock tree. The bond argument, `Settle`, is ignored.

To support existing code, `cbondbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`ConvRatio` — Number of shares convertible to one bond

nonnegative number

Number of shares convertible to one bond, specified as an `NINST`-by-`1` with a nonnegative number.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree,EquityTree,DebtTree] = cbondbyitt(ITTTree,CouponRate,Settle, Maturity, ConvRatio,'Spread',Spread,'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall',1)

Spread — Number of basis points over the reference rate
0 (default) | vector

Number of basis points over the reference rate, specified as the comma-separated pair consisting of 'Spread' and a NINST-by-1 vector.

Note To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument Spread. To incorporate default risk into the algorithm, specify the optional input arguments DefaultProbability and RecoveryRate. Do not use Spread with DefaultProbability and RecoveryRate.

Data Types: double

Period — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of 'Period' and a NINST-by-1 vector.

Data Types: double

IssueDate — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of 'IssueDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cbondbyitt also accepts serial date numbers as inputs, but they are not recommended.

FirstCouponDate — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cbondbyitt also accepts serial date numbers as inputs, but they are not recommended.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

Face value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector of nonnegative face values or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is the associated face value. The date indicates the last day that the face value is valid.

Data Types: `cell` | `double`

`CallStrike` — Call strike price for European, Bermuda, or American option
nonnegative integer | vector of nonnegative integers

Call strike price for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'CallStrike'` and one of the following values:

- For a European call option — `NINST`-by-`1` vector of nonnegative integers.
- For a Bermuda call option — `NINST`-by-`NSTRIKES` matrix of call strike price values, where each row is the schedule for one call option. If a call option has fewer than `NSTRIKES` exercise opportunities, the end of the row is padded with `NaN`s.
- For an American call option — `NINST`-by-`1` vector of strike price values for each option.

Data Types: `double`

`CallExDates` — Call exercise date for European, Bermuda, or American option
datetime array | string array | date character vector

Call exercise date for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'CallExDates'` and a datetime array, string array, or date character vectors for one of the following values:

- For a European option — `NINST`-by-`1` vector of date character vectors.
- For a Bermuda option — `NINST`-by-`NSTRIKES` matrix of exercise dates, where each row is the schedule for one option. For a European option, there is only one `CallExDate` on the option expiry date.
- For an American option — `NINST`-by-`1` or `NINST`-by-`2` matrix of exercise date boundaries. For each instrument, the call option can be exercised on any tree date between or including the pair of dates on that row. If `CallExDates` is `NINST`-by-`1`, the option can be exercised between the `ValuationDate` of the ITT stock tree and the single listed `CallExDate`.

To support existing code, `cbondbyitt` also accepts serial date numbers as inputs, but they are not recommended.

AmericanCall — Call option type indicator
0 if AmericanCall is NaN or not entered (default) | positive integer [0,1] | vector of positive integers [0,1]

Call option type, specified as the comma-separated pair consisting of 'AmericanCall' and a NINST-by-1 vector of positive integer flags with values 0 or 1.

- For a European or Bermuda option — AmericanCall is 0 for each European or Bermuda option.
- For an American option — AmericanCall is 1 for each American option. The AmericanCall argument is required to invoke American exercise rules.

Data Types: double

PutStrike — Put strike values for European, Bermuda, or American option
positive integer | vector of positive integers

Put strike values for European, Bermuda, or American option, specified as the comma-separated pair consisting of 'PutStrike' and one of the following values:

- For a European put option — NINST-by-1 vector of nonnegative integers.
- For a Bermuda put option — NINST-by-NSTRIKES matrix of strike price values where each row is the schedule for one option. If a put option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- For an American put option — NINST-by-1 vector of strike price values for each option.

Data Types: double

PutExDates — Put exercise date for European, Bermuda, or American option
datetime array | string array | date character vector

Put exercise date for European, Bermuda, or American option, specified as the comma-separated pair consisting of 'PutExDates' and a datetime array, string array, or date character vectors for one of the following values:

- For a European option — NINST-by-1 vector of date character vectors.
- For a Bermuda option — NINST-by-NSTRIKES matrix of exercise dates where each row is the schedule for one option. For a European option, there is only one PutExDate on the option expiry date.
- For an American option — NINST-by-1 or NINST-by-2 matrix of exercise date boundaries. For each instrument, the put option can be exercised on any tree date between or including the pair of dates on that row. If PutExDates is NINST-by-1, the put option can be exercised between the ValuationDate of the ITT stock tree and the single listed PutExDate.

To support existing code, cbondbyitt also accepts serial date numbers as inputs, but they are not recommended.

AmericanPut — Put option type indicator
0 if AmericanPut is NaN or not entered (default) | positive integer [0,1] | vector of positive integers [0,1]

Put option type, specified as the comma-separated pair consisting of 'PutExDates' and a NINST-by-1 vector of positive integer flags with values 0 or 1.

- For a European or Bermuda option — `AmericanPut` is `0` for each European or Bermuda option.
- For an American option — `AmericanPut` is `1` for each American option. The `AmericanPut` argument is required to invoke American exercise rules.

Data Types: `double`

`ConvDates` — Convertible dates
`MaturityDate` (default) | datetime array | string array | date character vector

Convertible dates, specified as the comma-separated pair consisting of `'ConvDates'` and a `NINST`-by-`1` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors. If `ConvDates` is not specified, the bond is always convertible until maturity.

To support existing code, `cbondbyitt` also accepts serial date numbers as inputs, but they are not recommended.

For each instrument, the bond can be converted on any tree date between or including the pair of dates on that row.

If `ConvDates` is `NINST`-by-`1`, the bond can be converted between the `ValuationDate` of the ITT stock tree and the single listed `ConvDates`.

`DefaultProbability` — Annual probability of default rate
`0` (default) | nonnegative decimal

Annual probability of default rate, specified as the comma-separated pair consisting of `'DefaultProbability'` and a `NINST`-by-`1` nonnegative decimal value.

Note To incorporate default risk into the algorithm, specify the optional input arguments `DefaultProbability` and `RecoveryRate`. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument `Spread`. Do not use `DefaultProbability` and `RecoveryRate` with `Spread`.

Data Types: `double`

`RecoveryRate` — Recovery rate
`1` (default) | nonnegative decimal

Recovery rate, specified as the comma-separated pair consisting of `'RecoveryRate'` and a `NINST`-by-`1` nonnegative decimal value.

Note To incorporate default risk into the algorithm, specify the optional input arguments `DefaultProbability` and `RecoveryRate`. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument `Spread`. Do not use `DefaultProbability` and `RecoveryRate` with `Spread`.

Data Types: `double`

Output Arguments

`Price` — Expected price at time `0`
array

Expected price at time `0`, returned as an `NINST`-by-`1` array.

`PriceTree` — Structure with vector of convertible bond prices at each node
tree structure

Structure with a vector of convertible bond prices at each node, returned as a tree structure.

`EquityTree` — Structure with vector of convertible bond equity component at each node
tree structure

Structure with a vector of convertible bond equity components at each node, returned as a tree structure.

`DebtTree` — Structure with vector of convertible bond debt component at each node
tree structure

Structure with a vector of convertible bond debt components at each node, returned as a tree structure.

More About

Callable Convertible

A convertible bond that is callable by the issuer. The issuer of the bond forces conversion, removing the advantage that conversion is at the discretion of the bondholder.

Upon call, the bondholder can either convert the bond or redeem at the call price. This option enables the issuer to control the price of the convertible bond and, if necessary, refinance the debt with a new cheaper bond.

Puttable Convertible

A convertible bond with a put feature allows the bondholder to sell back the bond at a premium on a specific date.

This option protects the holder against rising interest rates by reducing the year to maturity.

Algorithms

`cbondbycrr`, `cbondbyeqp`, `cbondbyitt`, and `cbondbysttr`eturn price information in the form of a price vector and a price tree. These functions implement the risk in the form of either a credit spread or incorporating the risk of bond default. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional name-value pair argument `Spread`. To incorporate default risk into the algorithm, specify the optional name-value pair arguments `DefaultProbability` and `RecoveryRate`.

Version History

Introduced in R2015b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `cbondbyitt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Tsiveriotis, K., and C. Fernandes. “Valuing Convertible Bonds with Credit Risk.” *Journal of Fixed Income*. Vol 8, 1998, pp. 95–102.

[2] Hull, J. *Options, Futures and Other Derivatives*. Fourth Edition. Prentice Hall, 2000, pp. 646–649.

See Also

`itttree` | `cbondbystt` | `instcbond` | `intenvset` | `stockspec` | `instadd` | `instdisp` | `ittprice` | `ittsens`

Topics

“Convertible Bond” on page 2-4

cbondbystt

Price convertible bonds from standard trinomial tree

Syntax

```
Price = cbondbystt(STTTree,CouponRate,Settle,Maturity,ConvRatio)
[Price,PriceTree] = cbondbystt(STTTree,CouponRate,Settle,Maturity,ConvRatio)
[Price,PriceTree,EquityTree,DebtTree] = cbondbystt( ___ ,Name,Value)
```

Description

`Price = cbondbystt(STTTree,CouponRate,Settle,Maturity,ConvRatio)` prices convertible bonds using a standard trinomial (STT) tree using the Tsiveriotis and Fernandes method.

`[Price,PriceTree] = cbondbystt(STTTree,CouponRate,Settle,Maturity,ConvRatio)` prices convertible bonds using a standard trinomial (STT) tree using the Tsiveriotis and Fernandes method.

`[Price,PriceTree,EquityTree,DebtTree] = cbondbystt( ___ ,Name,Value)` prices convertible bonds from a standard trinomial (STT) tree using a credit spread or incorporating the risk of bond default.

To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional name-value pair input argument `Spread`. To incorporate default risk into the algorithm, specify the optional name-value pair input arguments `DefaultProbability` and `RecoveryRate`.

Examples

Price a Convertible Bond Using a STTTree

Create a `RateSpec`.

```
StartDates = datetime(2015,1,1);
EndDates = datetime(2020,1,1);
Rates = 0.025;
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8825
            Rates: 0.0250
        EndTimes: 5
       StartTimes: 0
         EndDates: 737791
       StartDates: 735965
    ValuationDate: 735965
```

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

Create a StockSpec.

```
AssetPrice = 80;
Sigma = 0.12;
StockSpec = stockspec(Sigma,AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1200
         AssetPrice: 80
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Create a STTTree.

```
TimeSpec = stttimespec(StartDates, EndDates, 20);
STTTree = stttree(StockSpec, RateSpec, TimeSpec)
```

```
STTTree = struct with fields:
       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
         dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]
        STree: {1x21 cell}
        Probs: {1x20 cell}
```

Define the convertible bond. The convertible bond can be called starting on Jan 1, 2016 with a strike price of 95.

```
CouponRate = 0.03;
Settle = datetime(2015,1,1);
Maturity = datetime(2018,4,1);
Period = 1;
CallStrike = 95;
CallExDates = [datetime(2016,1,1) ; datetime(2018,1,1)];
ConvRatio = 1;
Spread = 0.025;
```

Price the convertible bond using the standard trinomial tree model.

```
[Price,PriceTree,EqtTre,DbtTree] = cbondbystt(STTTree,CouponRate,Settle,Maturity,ConvRatio,...
'Period',Period,'Spread',Spread,'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall'
```

```
Price = 2×1

   91.8885
   90.6328


PriceTree = struct with fields:
    FinObj: 'TrinPriceTree'
```

```
     PTree: {1x21 cell}
      tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
      dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]


EqtTre = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {1x21 cell}
      tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
      dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]


DbtTree = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {1x21 cell}
      tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
      dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]
```

Spread Effect Analysis for a Convertible Bond Using a STTTree

This example demonstrates the spread effect analysis of a 4% coupon convertible bond, callable at 110 at end of the second year, maturing in five years, with spreads of 0, 50, 100, and 150 BP.

Define the `RateSpec`.

```
StartDates = datetime(2015,4,1);
EndDates = datetime(2020,4,1);
Rates = 0.05;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('StartDates',StartDates,'EndDates',EndDates,'Rates',Rates,...
'Compounding',Compounding,'Basis',Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.7788
            Rates: 0.0500
         EndTimes: 5
       StartTimes: 0
         EndDates: 737882
       StartDates: 736055
    ValuationDate: 736055
            Basis: 1
     EndMonthRule: 1
```

Define the convertible bond data.

```
Settle = datetime(2015,4,1);
Maturity = datetime(2020,4,1);
CouponRate = 0.04;
CallStrike = 110;
CallExDates = [datetime(2017,4,1)  datetime(2020,4,1)];
ConvRatio = 1;
```

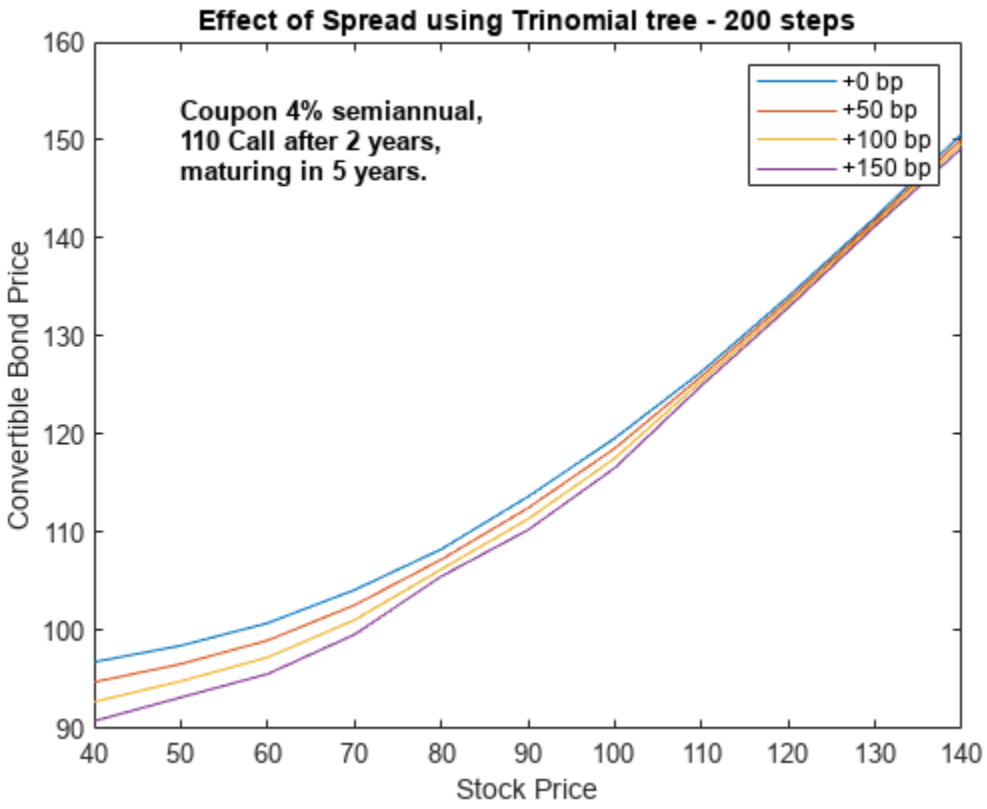
```
AmericanCall = 1;
Sigma = 0.3;
Spreads = 0:0.005:0.015;
Prices = 40:10:140;
convprice = zeros(length(Prices),length(Spreads));
```

Define the `TimeSpec` for the Standard Trinomial Tree, create an `STTTree` using `stttree`, and price the convertible bond using `cbondbystt`.

```
NumSteps = 200;
TimeSpec = stttimespec(StartDates, EndDates, NumSteps);

for PriceIdx = 1:length(Prices)
    StockSpec = stockspec(Sigma, Prices(PriceIdx));
    STTT = stttree(StockSpec, RateSpec, TimeSpec);
    convprice(PriceIdx,:) = cbondbystt(STTT,  CouponRate, Settle, Maturity, ConvRatio,...
    'Spread', Spreads(:),'CallExDates', CallExDates, 'CallStrike', CallStrike,...
    'AmericanCall', AmericanCall);
end
```

Plot the spread effect analysis for the convertible bond.

```
stock = repmat(Prices',1,length(Spreads));
plot(stock,convprice);
legend({'+0 bp'; '+50 bp'; '+100 bp'; '+150 bp'});
title ('Effect of Spread using Trinomial tree - 200 steps')
xlabel('Stock Price');
ylabel('Convertible Bond Price');
text(50, 150, ['Coupon 4% semiannual,', sprintf('\n'), ...
    '110 Call after 2 years,' sprintf('\n'), ...
    'maturing in 5 years.'],'fontweight','Bold')
```



Price a Convertible Bond Using an STT Tree and Incorporate Default Risk Using `DefaultProbability` and `RecoveryRate`

Create the interest-rate term structure `RateSpec`.

```
StartDates = datetime(2015,1,1);
EndDates = datetime(2020,1,1);
Rates = 0.025;
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8825
            Rates: 0.0250
        EndTimes: 5
       StartTimes: 0
         EndDates: 737791
       StartDates: 735965
    ValuationDate: 735965
            Basis: 1
```

```
     EndMonthRule: 1
```

Create the `StockSpec`.

```
AssetPrice = 80;
Sigma = 0.12;
StockSpec = stockspec(Sigma,AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1200
         AssetPrice: 80
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Create the STT tree for the equity.

```
TimeSpec = stttimespec(StartDates, EndDates, 20);
STTTree = stttree(StockSpec, RateSpec, TimeSpec)
```

```
STTTree = struct with fields:
       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
         dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]
        STree: {1x21 cell}
        Probs: {1x20 cell}
```

Define and price the convertible bond using the optional `DefaultProbability` and `RecoveryRate` arguments.

```
CouponRate = 0.03;
Settle = datetime(2015,1,1);
Maturity = datetime(2018,4,1);
Period = 1;
CallStrike = 95;
CallExDates = [datetime(2016,1,1) datetime(2018,1,1)];
ConvRatio = 1;
DefaultProbability = .30;
RecoveryRate = .82;

[Price,PriceTree,EqtTre,DbtTree] = cbondbystt(STTTree,CouponRate,Settle,Maturity,ConvRatio,...
'Period',Period,'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall',1,...
'DefaultProbability',DefaultProbability,'RecoveryRate',RecoveryRate)
```

```
Price = 80

PriceTree = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {1x21 cell}
      tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
      dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]
```

```
EqtTre = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {1x21 cell}
      tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
      dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]


DbtTree = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {1x21 cell}
      tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
      dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]
```

Input Arguments

STTTree — Stock tree structure for standard trinomial tree

structure

Stock tree structure for a standard trinomial tree, specified by using `sttree`.

Data Types: `struct`

CouponRate — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

Settle — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every convertible bond is set to the `ValuationDate` of the standard trinomial (STT) stock tree. The bond argument, `Settle`, is ignored.

To support existing code, `cbondbystt` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbystt` also accepts serial date numbers as inputs, but they are not recommended.

`ConvRatio` — Number of shares convertible to one bond
nonnegative number

Number of shares convertible to one bond, specified as an `NINST`-by-`1` with a nonnegative number.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree,EquityTree,DebtTree] = cbondbystt(STTTree,CouponRate,Settle,Maturity,ConvRatio,'Spread',Spread,'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall',1)`

`Spread` — Number of basis points over the reference rate
`0` (default) | vector

Number of basis points over the reference rate, specified as the comma-separated pair consisting of `'Spread'` and a `NINST`-by-`1` vector.

Note To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument `Spread`. To incorporate default risk into the algorithm, specify the optional input arguments `DefaultProbability` and `RecoveryRate`. Do not use `Spread` with `DefaultProbability` and `RecoveryRate`.

Data Types: `double`

`Period` — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbystt` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbystt` also accepts serial date numbers as inputs, but they are not recommended.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cbondbystt` also accepts serial date numbers as inputs, but they are not recommended.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

Face value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector of nonnegative face values or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is the associated face value. The date indicates the last day that the face value is valid.

Data Types: `cell` | `double`

`CallStrike` — Call strike price for European, Bermuda, or American option
nonnegative integer | vector of nonnegative integers

Call strike price for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'CallStrike'` and one of the following values:

- For a European call option — `NINST`-by-`1` vector of nonnegative integers
- For a Bermuda call option — `NINST`-by-`NSTRIKES` matrix of call strike price values, where each row is the schedule for one call option. If a call option has fewer than `NSTRIKES` exercise opportunities, the end of the row is padded with `NaN`s.
- For an American call option — `NINST`-by-`1` vector of strike price values for each option.

Data Types: `double`

`CallExDates` — Call exercise date for European, Bermuda, or American option
datetime array | string array | date character vector

Call exercise date for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'CallExDates'` and a datetime array, string array, or date character vectors for one of the following values:

- For a European option — `NINST`-by-`1` vector of date character vectors.
- For a Bermuda option — `NINST`-by-`NSTRIKES` matrix of exercise dates, where each row is the schedule for one option. For a European option, there is only one `CallExDate` on the option expiry date.
- For an American option — `NINST`-by-`1` or `NINST`-by-`2` matrix of exercise date boundaries. For each instrument, the call option can be exercised on any tree date between or including the pair of dates on that row. If `CallExDates` is `NINST`-by-`1`, the option can be exercised between the `ValuationDate` of the STT stock tree and the single listed `CallExDate`.

To support existing code, `cbondbystt` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanCall` — Call option type indicator
`0` if `AmericanCall` is `NaN` or not entered (default) | positive integer `[0,1]` | vector of positive integers `[0,1]`

Call option type, specified as the comma-separated pair consisting of `'AmericanCall'` and a `NINST`-by-`1` vector of positive integer flags with values `0` or `1`.

- For a European or Bermuda option — `AmericanCall` is `0` for each European or Bermuda option.
- For an American option — `AmericanCall` is `1` for each American option. The `AmericanCall` argument is required to invoke American exercise rules.

Data Types: `double`

`PutStrike` — Put strike values for European, Bermuda, or American option
positive integer | vector of positive integers

Put strike values for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'PutStrike'` and one of the following values:

- For a European put option — `NINST`-by-`1` vector of nonnegative integers.
- For a Bermuda put option — `NINST`-by-`NSTRIKES` matrix of strike price values where each row is the schedule for one option. If a put option has fewer than `NSTRIKES` exercise opportunities, the end of the row is padded with `NaN`s.
- For an American put option — `NINST`-by-`1` vector of strike price values for each option.

Data Types: `double`

`PutExDates` — Put exercise date for European, Bermuda, or American option
datetime array | string array | date character vector

Put exercise date for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'PutExDates'` and a datetime array, string array, or date character vectors for one of the following values:

- For a European option — `NINST`-by-`1` vector of date character vectors.
- For a Bermuda option — `NINST`-by-`NSTRIKES` matrix of exercise dates where each row is the schedule for one option. For a European option, there is only one `PutExDate` on the option expiry date.
- For an American option — `NINST`-by-`1` or `NINST`-by-`2` matrix of exercise date boundaries. For each instrument, the put option can be exercised on any tree date between or including the pair of dates on that row. If `PutExDates` is `NINST`-by-`1`, the put option can be exercised between the `ValuationDate` of the STT stock tree and the single listed `PutExDate`.

To support existing code, `cbondbystt` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanPut` — Put option type indicator
`0` if `AmericanPut` is `NaN` or not entered (default) | positive integer `[0,1]` | vector of positive integers `[0,1]`

Put option type, specified as the comma-separated pair consisting of `'AmericanPut'` and a `NINST`-by-`1` vector of positive integer flags with values `0` or `1`.

- For a European or Bermuda option — `AmericanPut` is `0` for each European or Bermuda option.
- For an American option — `AmericanPut` is `1` for each American option. The `AmericanPut` argument is required to invoke American exercise rules.

Data Types: `double`

`ConvDates` — Convertible dates
`MaturityDate` (default) | datetime array | string array | date character vector

Convertible dates, specified as the comma-separated pair consisting of `'ConvDates'` and a `NINST`-by-`1` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors. If `ConvDates` is not specified, the bond is always convertible until maturity.

To support existing code, `cbondbystt` also accepts serial date numbers as inputs, but they are not recommended.

For each instrument, the bond can be converted on any tree date between or including the pair of dates on that row.

If `ConvDates` is `NINST`-by-`1`, the bond can be converted between the `ValuationDate` of the standard trinomial (STT) stock tree and the single listed `ConvDates`.

`DefaultProbability` — Annual probability of default rate
`0` (default) | nonnegative decimal

Annual probability of default rate, specified as the comma-separated pair consisting of `'DefaultProbability'` and a `NINST`-by-`1` nonnegative decimal value.

Note To incorporate default risk into the algorithm, specify the optional input arguments `DefaultProbability` and `RecoveryRate`. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument `Spread`. Do not use `DefaultProbability` and `RecoveryRate` with `Spread`.

Data Types: `double`

`RecoveryRate` — Recovery rate
`1` (default) | nonnegative decimal

Recovery rate, specified as the comma-separated pair consisting of `'RecoveryRate'` and a `NINST`-by-`1` nonnegative decimal value.

Note To incorporate default risk into the algorithm, specify the optional input arguments `DefaultProbability` and `RecoveryRate`. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional input argument `Spread`. Do not use `DefaultProbability` and `RecoveryRate` with `Spread`.

Data Types: `double`

Output Arguments

`Price` — Expected price at time `0`
array

Expected price at time `0`, returned as an `NINST`-by-`1` array.

`PriceTree` — Structure with vector of convertible bond prices at each node
tree structure

Structure with a vector of convertible bond prices at each node, returned as a tree structure.

`EquityTree` — Structure with vector of convertible bond equity component at each node
tree structure

Structure with a vector of convertible bond equity components at each node, returned as a tree structure.

`DebtTree` — Structure with vector of convertible bond debt component at each node
tree structure

Structure with a vector of convertible bond debt components at each node, returned as a tree structure.

More About

Callable Convertible

A convertible bond that is callable by the issuer. The issuer of the bond forces conversion, removing the advantage that conversion is at the discretion of the bondholder.

Upon call, the bondholder can either convert the bond or redeem at the call price. This option enables the issuer to control the price of the convertible bond and, if necessary, refinance the debt with a new cheaper bond.

Puttable Convertible

A convertible bond with a put feature allows the bondholder to sell back the bond at a premium on a specific date.

This option protects the holder against rising interest rates by reducing the year to maturity.

Algorithms

`cbondbycrr`, `cbondbyeqp`, `cbondbyitt`, and `cbondbysttreturn` price information in the form of a price vector and a price tree. These functions implement the risk in the form of either a credit spread or incorporating the risk of bond default. To incorporate the risk in the form of credit spread (Tsiveriotis-Fernandes method), use the optional name-value pair argument `Spread`. To incorporate default risk into the algorithm, specify the optional name-value pair arguments `DefaultProbability` and `RecoveryRate`.

Version History

Introduced in R2015b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `cbondbystt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Tsiveriotis, K., and C. Fernandes. “Valuing Convertible Bonds with Credit Risk.” *Journal of Fixed Income*. Vol 8, 1998, pp. 95–102.

[2] Hull, J. *Options, Futures and Other Derivatives*. Fourth Edition. Prentice Hall, 2000, pp. 646–649.

See Also

`stttree` | `cbondbyeqp` | `cbondbycrr` | `instcbond` | `intenvset` | `stockspec` | `instadd` | `instdisp` | `sttprice` | `sttsens`

Topics

“Convertible Bond” on page 2-4

cfbybdt

Price cash flows from Black-Derman-Toy interest-rate tree

Syntax

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

Description

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

`[Price,PriceTree] = cfbybdt( ___ ,Basis,Options)` adds optional arguments.

Examples

Price a Portfolio Containing Two Cash Flow Instruments

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 = 2×1

   74.0112
   74.3671


PriceTree = struct with fields:
    FinObj: 'BDTPriceTree'
```

```
     tObs: [0 1 2 3 4]
    PTree: {1x5 cell}
```

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

Input Arguments

`BDTTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bdttree`.

Data Types: `struct`

`CFlowAmounts` — Cash flow amounts

matrix

Cash flow amounts, specified as a 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.

Data Types: `double`

`CFlowDates` — Cash flow dates

serial date number

Cash flow dates, specified as `NINST`-by-`MOSTCFS` vector using serial date numbers. Each entry contains the date of the corresponding cash flow in `CFlowAmounts`.

Data Types: `double`

`Settle` — Settlement date

serial date number | date character vector

Settlement date, specified as a vector using serial date numbers or date character vectors. The `Settle` date for every cash flow is set to the `ValuationDate` of the BDT tree. The cash flow argument, `Settle`, is ignored.

Data Types: `double` | `char`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis of the instrument, specified as a vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices at time 0
vector

Expected prices at time 0, returned as a `NINST`-by-1 vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as a MATLAB structure of trees containing vectors of instrument prices and observation times for each node. Within `PriceTree`:

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

Version History

Introduced before R2006a

See Also

`bdttree` | `bdtprice` | `cfamounts` | `instcf`

Topics

“Computing Instrument Prices” on page 2-81
“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

cfbybk

Price cash flows from Black-Karasinski interest-rate tree

Syntax

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

Description

`[Price,PriceTree] = cfbybk(BKTree,CFlowAmounts,CFlowDates,Settle)` prices cash flows from a Black-Karasinski interest-rate tree.

`[Price,PriceTree] = cfbybk( ___ ,Basis,Options)` adds optional arguments.

Examples

Price a Portfolio Containing Two Cash Flow Instruments

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)
```

```
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In cfbytrintree (line 88)
  In cfbybk (line 75)

Price =
```

```
   93.3600
   81.6218


PriceTree =

  struct with fields:

     FinObj: 'BKPriceTree'
      PTree: {[2×1 double]  [2×3 double]  [2×5 double]  [2×5 double]  [2×5 double]}
       tObs: [0 1 2 3 4]
    Connect: {[2]  [2 3 4]  [2 2 3 4 4]}
      Probs: {[3×1 double]  [3×3 double]  [3×5 double]}
```

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

Input Arguments

`BKTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bktree`.

Data Types: `struct`

`CFlowAmounts` — Cash flow amounts

matrix

Cash flow amounts, specified as a 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.

Data Types: `double`

`CFlowDates` — Cash flow dates

serial date number

Cash flow dates, specified as `NINST`-by-`MOSTCFS` vector using serial date numbers. Each entry contains the serial date number of the corresponding cash flow in `CFlowAmounts`.

Data Types: `double`

`Settle` — Settlement date

serial date number | date character vector

Settlement date, specified as a vector using serial date numbers or date character vectors. The `Settle` date for every cash flow is set to the `ValuationDate` of the BK tree. The cash flow argument, `Settle`, is ignored.

Data Types: `double` | `char`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis of the instrument, specified as a vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices at time 0
vector

Expected prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as a MATLAB structure of trees containing vectors of instrument prices and observation times for each node. Within `PriceTree`:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

Version History

Introduced before R2006a

See Also

`bktree` | `bkprice` | `cfamounts` | `instcf`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

cfbycir

Price cash flows from Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = cfbycir(CIRTree,CFlowAmounts,CFlowDates,Settle)
[Price,PriceTree] = cfbycir( ___ ,Basis)
```

Description

`[Price,PriceTree] = cfbycir(CIRTree,CFlowAmounts,CFlowDates,Settle)` prices cash flows from a Cox-Ingersoll-Ross (CIR) interest-rate tree using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = cfbycir( ___ ,Basis)` adds an optional argument for `Basis`.

Examples

Price a Portfolio Containing Two Cash Flow Instruments

Price a portfolio containing two cash flow instruments paying interest annually over the four-year period from January 1, 2017 to June 1, 2020.

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

```
load deriv.mat;
```

The valuation date (settle date) specified in the `CIRTree` is January 1, 2017 (serial date number `736696`).

```
CIRTree.RateSpec.ValuationDate
```

```
ans = 736696
```

Provide values for the other required arguments.

```
CFlowAmounts =[5 NaN 5.5 105; 5 0 6 105];
CFlowDates = [736847,NaN,737061,737212;
              737426,737577,737791,737943];
```

Compute the prices of the two cash flow instruments using `cfbycir`.

```
[Price,PriceTree] = cfbycir(CIRTree, CFlowAmounts, CFlowDates,...
CIRTree.RateSpec.ValuationDate)
```

```
Price = 2×1

  109.6845
   98.7246
```

```
PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
      PTree: {1x5 cell}
       tObs: [0 1 2 3 4]
    Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
      Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `cirtree`.

Data Types: `struct`

`CFlowAmounts` — Cash flow amounts

matrix

Cash flow amounts, specified as a 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.

Data Types: `double`

`CFlowDates` — Cash flow dates

serial date number

Cash flow dates, specified as `NINST`-by-`MOSTCFS` vector using serial date numbers. Each entry contains the serial date number of the corresponding cash flow in `CFlowAmounts`.

Data Types: `double`

`Settle` — Settlement date

serial date number | date character vector

Settlement date, specified as a vector using serial date numbers or date character vectors. The `Settle` date for every cash flow is set to the `ValuationDate` of the CIR tree. The cash flow argument `Settle` is ignored.

Data Types: `double` | `char`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis of the instrument, specified as a vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)

- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

`Price` — Expected prices at time 0

vector

Expected prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as a MATLAB structure of trees containing vectors of instrument prices and observation times for each node. Within `PriceTree`:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

Version History

Introduced in R2018a

References

[1] Cox, J., Ingersoll, J.,and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir` | `instcf`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

cfbyhjm

Price cash flows from Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree] = cfbyhjm(HJMTree,CFlowAmounts,CFlowDates,Settle)
[Price,PriceTree] = cfbyhjm( ___ ,Basis,Options)
```

Description

`[Price,PriceTree] = cfbyhjm(HJMTree,CFlowAmounts,CFlowDates,Settle)` prices cash flows from a Heath-Jarrow-Morton interest-rate tree.

`[Price,PriceTree] = cfbyhjm( ___ ,Basis,Options)` adds optional arguments.

Examples

Price a Portfolio Containing Two Cash Flow Instruments

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 interest-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`).

```
HJMTree.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 = 2×1

   96.7805
   97.2188


PriceTree = struct with fields:
    FinObj: 'HJMPriceTree'
```

```
     tObs: [0 1 2 3 4]
    PBush: {1x5 cell}
```

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

Input Arguments

`HJMTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hjmtree`.

Data Types: `struct`

`CFlowAmounts` — Cash flow amounts

matrix

Cash flow amounts, specified as a 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.

Data Types: `double`

`CFlowDates` — Cash flow dates

serial date number

Cash flow dates, specified as `NINST`-by-`MOSTCFS` vector using serial date numbers. Each entry contains the serial date number of the corresponding cash flow in `CFlowAmounts`.

Data Types: `double`

`Settle` — Settlement date

serial date number | date character vector

Settlement date, specified as a vector using serial date numbers or date character vectors. The `Settle` date for every cash flow is set to the `ValuationDate` of the HJM tree. The cash flow argument, `Settle`, is ignored.

Data Types: `double` | `char`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis of the instrument, specified as a vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices at time 0
vector

Expected prices at time 0, returned as a `NINST`-by-1 vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as a MATLAB structure of trees containing vectors of instrument prices and observation times for each node. Within `PriceTree`:

- `PriceTree.tObs` contains the observation times.
- `PriceTree.PBush` contains the clean prices.

Version History

Introduced before R2006a

See Also

`cfamounts` | `hjmprice` | `hjmtree` | `instcf`

Topics
“Computing Instrument Prices” on page 2-81
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

cfbyhw

Price cash flows from Hull-White interest-rate tree

Syntax

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

Description

`[Price,PriceTree] = cfbyhw(HWTree,CFlowAmounts,CFlowDates,Settle)` prices cash flows from a Hull-White interest-rate tree.

`[Price,PriceTree] = cfbyhw( ___ ,Basis,Options)` adds optional arguments.

Examples

Price a Portfolio Containing Two Cash Flow Instruments

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)
```

```
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In cfbytrintree (line 88)
  In cfbyhw (line 75)

Price =
```

```
   93.3789
   81.7651


PriceTree =

  struct with fields:

     FinObj: 'HWPriceTree'
      PTree: {[2×1 double]  [2×3 double]  [2×5 double]  [2×5 double]  [2×5 double]}
       tObs: [0 1 2 3 4]
    Connect: {[2]  [2 3 4]  [2 2 3 4 4]}
      Probs: {[3×1 double]  [3×3 double]  [3×5 double]}
```

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

Input Arguments

`HWTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hwtree`.

Data Types: `struct`

`CFlowAmounts` — Cash flow amounts

matrix

Cash flow amounts, specified as a 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.

Data Types: `double`

`CFlowDates` — Cash flow dates

serial date number

Cash flow dates, specified as `NINST`-by-`MOSTCFS` vector using serial date numbers. Each entry contains the serial date number of the corresponding cash flow in `CFlowAmounts`.

Data Types: `double`

`Settle` — Settlement date

serial date number | date character vector

Settlement date, specified as a vector using serial date numbers or date character vectors. The `Settle` date for every cash flow is set to the `ValuationDate` of the HW tree. The cash flow argument, `Settle`, is ignored.

Data Types: `double` | `char`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis of the instrument, specified as a vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices at time 0
vector

Expected prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as a MATLAB structure of trees containing vectors of instrument prices and observation times for each node. Within `PriceTree`:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

Version History

Introduced before R2006a

See Also

`cfamounts` | `hwtree` | `hwprice` | `instcf`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

cfbyzero

Price cash flows from set of zero curves

Syntax

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

Description

`Price = cfbyzero(RateSpec,CFlowAmounts,CFlowDates,Settle)` prices cash flows from a set of zero curves.

`Price = cfbyzero( ___ ,Basis)` adds an optional argument.

Examples

Compute the Price and Sensitivity From the Interest-Rate Term Structure

This example shows how to 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 = 2×1

   96.7804
   97.2187
```

Input Arguments

RateSpec — Annualized zero rate term structure

structure

Annualized zero rate term structure, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

CFlowAmounts — Cash flow amounts

matrix

Cash flow amounts, specified as a Number of instruments (NINST) by maximum number of cash flows (MOSTCFS) matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than MOSTCFS cash flows, the end of the row is padded with NaNs.

Data Types: double

CFlowDates — Cash flow dates
serial date number

Cash flow dates, specified as NINST-by-MOSTCFS vector using serial date numbers. Each entry contains the serial date number of the corresponding cash flow in CFlowAmounts.

Data Types: double

Settle — Settlement date on which cash flows are priced
serial date number | date character vector

Settlement date on which the cash flows are priced, specified using a NINST-by-1 vector with serial date numbers or date character vectors of the same value which represent the settlement date for each cash flow. Settle must be earlier than Maturity.

Data Types: double | char

Basis — Day-count basis of instrument
0 (actual/actual) (default) | integer from 0 to 13

(Optional) Day-count basis of the instrument, specified as a vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: double

Output Arguments

Price — Cash flow prices
matrix

Cash flow prices, returned as a `NINST`-by-`NUMCURVES` matrix where each column arises from one of the zero curves.

Version History

Introduced before R2006a

See Also

`bondbyzero` | `fixedbyzero` | `floatbyzero` | `swapbyzero`

Topics

“Pricing Using Interest-Rate Term Structure” on page 2-61
“Understanding the Interest-Rate Term Structure” on page 2-48
“Supported Interest-Rate Instrument Functions” on page 2-3

chooserbybls

Price European simple chooser options using Black-Scholes model

Syntax

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

Description

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

Examples

Price European Simple Chooser Options Using the Black-Scholes Model

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 = datetime(2007,1,1);
Maturity = datetime(2007,12,2);
ChooseDate = datetime(2007,8,31);
RiskFreeRate = 0.1;
Sigma = 0.20;
Yield = 0.05
```

```
Yield = 0.0500
```

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 = 9.2781
```

Input Arguments

RateSpec — Annualized zero rate term structure
structure

Annualized zero rate term structure, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset
structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

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

Data Types: `struct`

`Settle` — Settlement or trade dates
datetime array | string array | date character vector

Settlement or trade dates, specified using an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `chooserbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `chooserbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Strike` — Option strike price value
nonnegative integer

Option strike price value, specified with a `NINST`-by-`1` vector of nonnegative integers.

Data Types: `double`

`ChooseDate` — Chooser dates
datetime array | string array | date character vector

Choose dates, specified with a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `chooserbybls` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`Price` — Expected prices
vector

Expected prices, returned as an `NINST`-by-`1` vector.

Version History

Introduced in R2008b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `chooserbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Rubinstein, Mark. "Options for the Undecided." *Risk*. Vol 4, 1991.

See Also

`blsprice` | `intenvset`

Topics
“Pricing Using the Black-Scholes Model” on page 3-82
“Chooser Option” on page 3-23
“Supported Equity Derivative Functions” on page 3-19

cirprice

Instrument prices from Cox-Ingersoll-Ross interest-rate model

Syntax

```
Price = cirprice(CIRTree,InstSet)
[Price,PriceTree] = cirprice(CIRTree,InstSet)
```

Description

`Price = cirprice(CIRTree,InstSet)` computes prices for instruments using a Cox-Ingersoll-Ross (CIR) interest rate tree created with `cirtree`. The CIR tree uses a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`cirprice` handles the following instrument type values: `'Bond'`, `'CashFlow'`,`'OptBond'`, `'Fixed'`, `'Float'`, `'Cap'`, `'Floor'`, `'Swap'`,`'Swaption'`, `'RangeFloat'`, `'OptFloat'`, `'OptEmFloat'`.

`[Price,PriceTree] = cirprice(CIRTree,InstSet)` returns the optional output for `PriceTree`.

Examples

Price Bonds Using a CIR Interest-Rate Tree

Define two bond instruments.

```
CouponRate= [0.035;0.04];
Settle= 'Jan-1-2017';
Maturity = 'Jan-1-2019';
Period = 1;
InstSet = instbond(CouponRate, Settle, Maturity, Period)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Bond'}
     FieldName: {{11x1 cell}}
    FieldClass: {{11x1 cell}}
     FieldData: {{11x1 cell}}
```

```
instdisp(InstSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.035      01-Jan-2017    01-Jan-2019    1      0     1            NaN       NaN
2     Bond 0.04       01-Jan-2017    01-Jan-2019    1      0     1            NaN       NaN
```

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = {'Jan-1-2017'; 'Jan-1-2018'; 'Jan-1-2019'; 'Jan-1-2020'; 'Jan-1-2021'};
ValuationDate = 'Jan-1-2017';
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',End
```

Create a CIR tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = '01-Jan-2017';
Maturity = '01-Jan-2019';
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 0.5000 1 1.5000]
        dObs: [736696 736878 737061 737243]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Price the bonds.

```
Price = cirprice(CIRT,InstSet)
```

```
Price = 2×1

   98.6793
   99.6228
```

Input Arguments

CIRTree — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `cirtree`.

Data Types: `struct`

InstSet — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

Output Arguments

`Price` — Expected floating-rate note prices at time 0

vector

Expected floating-rate note prices at time 0, returned as a `NINST`-by-1 vector. The prices are computed by backward dynamic programming on the interest-rate tree. If an instrument cannot be priced, a `NaN` is returned in that entry.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as 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.

Version History

Introduced in R2018a

References

[1] Cox, J., Ingersoll, J.,and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`cirsens` | `bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir`

Topics

“Computing Instrument Prices” on page 2-81
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

cirsens

Instrument sensitivities and prices from Cox-Ingersoll-Ross interest-rate model

Syntax

```
[Delta,Gamma,Vega,Price] = cirsens(CIRTree,InstSet)
```

Description

`[Delta,Gamma,Vega,Price] = cirsens(CIRTree,InstSet)` computes dollar sensitivities and prices for instruments using a Cox-Ingersoll-Ross (CIR) interest rate tree created with `cirtree`. The CIR tree uses a CIR++ model with the Nawalka-Beliaeva (NB) approach.

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

`cirsens` handles the following instrument type values: `'Bond'`, `'CashFlow'`,`'OptBond'`, `'Fixed'`, `'Float'`, `'Cap'`, `'Floor'`, `'Swap'`,`'Swaption'`, `'RangeFloat'`, `'OptFloat'`, `'OptEmFloat'`.

Examples

Compute Instrument Sensitivities Using a CIR Interest-Rate Tree

Define and set up two bond instruments. Compute `Delta` and `Gamma` for the bond instruments contained in the instrument set.

```
CouponRate= [0.035;0.04];
Settle= 'Jan-1-2017';
Maturity = 'Jan-1-2019';
Period = 1;
InstSet = instbond(CouponRate, Settle, Maturity, Period)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Bond'}
     FieldName: {{11x1 cell}}
    FieldClass: {{11x1 cell}}
     FieldData: {{11x1 cell}}
```

```
instdisp(InstSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.035      01-Jan-2017    01-Jan-2019    1      0     1            NaN       NaN
2     Bond 0.04       01-Jan-2017    01-Jan-2019    1      0     1            NaN       NaN
```

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = {'Jan-1-2017'; 'Jan-1-2018'; 'Jan-1-2019'; 'Jan-1-2020'; 'Jan-1-2021'};
ValuationDate = 'Jan-1-2017';
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',End
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = '01-Jan-2017';
Maturity = '01-Jan-2019';
CIRTimeSpec = cirtimespec(Settle, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 0.5000 1 1.5000]
        dObs: [736696 736878 737061 737243]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Calculate the `Delta` and `Gamma` sensitivities for the two bonds.

```
[Delta, Gamma] = cirsens(CIRT,InstSet)
```

```
Delta = 2×1

 -186.1885
 -187.5390


Gamma = 2×1

  532.8675
  536.3132
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `cirtree`.

Data Types: `struct`

`InstSet` — Instrument variable
structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

Output Arguments

`Delta` — Rate of change of instruments prices with respect to changes in interest rate
vector

Rate of change of instruments prices with respect to changes in the interest rate, returned as a `NINST`-by-`1` vector of deltas. `Delta` is computed by finite differences in calls to `cirtree`.

Note `Delta` is calculated based on yield shifts of 100 basis points.

`Gamma` — Rate of change of instruments deltas with respect to changes in interest rate
vector

Rate of change of instruments deltas with respect to changes in the interest rate, returned as a `NINST`-by-`1` vector of gammas. `Gamma` is computed by finite differences in calls to `cirtree`.

Note `Gamma` is calculated based on yield shifts of 100 basis points.

`Vega` — Rate of change of instruments prices with respect to changes in volatility
vector

Rate of change of instruments prices with respect to changes in the volatility, returned as a `NINST`-by-`1` vector of vegas. Volatility is Sigma (t,T) of the interest rate. `Vega` is computed by finite differences in calls to `cirtree`. For information on the volatility process, see `cirvolspec`.

Note `Vega` is calculated based on 1% shift in the volatility process.

`Price` — Price of each instrument
vector

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

Version History

Introduced in R2018a

References

[1] Cox, J., Ingersoll, J.,and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica.* Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice.* Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance.* CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling.* Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies.* Vol 3. 1990, pp. 393–430.

See Also

`cirprice` | `bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir`

Topics

“Computing Instrument Prices” on page 2-81
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

classfin

Create financial structure or return financial structure class name

Syntax

```
Obj = classfin(ClassName)
Obj = classfin(Struct,ClassName)
ClassName = classfin(Obj)
```

Description

`Obj = classfin(ClassName)` create a financial structure of class `ClassName`.

`Obj = classfin(Struct,ClassName)` create a converted MATLAB financial structure of class `ClassName`.

`ClassName = classfin(Obj)` returns a character vector containing a financial structure's class name.

Examples

Create a Financial Structure

This example shows how to create a financial structure `HJMTimeSpec` and complete its fields. (Typically, the function `hjmtimespec` is used to create `HJMTimeSpec` structures).

```
TimeSpec = classfin('HJMTimeSpec');
TimeSpec.ValuationDate = datetime(1999,12,10);
TimeSpec.Maturity = datetime(2002,12,10);
TimeSpec.Compounding = 2;
TimeSpec.Basis = 0;
TimeSpec.EndMonthRule = 1;
TimeSpec
```

```
TimeSpec = struct with fields:
           FinObj: 'HJMTimeSpec'
    ValuationDate: 10-Dec-1999
         Maturity: 10-Dec-2002
      Compounding: 2
            Basis: 0
     EndMonthRule: 1
```

Convert an Existing MATLAB Structure into a Financial Structure

This example shows how to convert an existing MATLAB structure into a financial structure.

```
TSpec.ValuationDate = datetime(1999,12,10);
TSpec.Maturity = datetime(2002,12,10);
```

```
TSpec.Compounding = 2;
TSpec.Basis = 0;
TSpec.EndMonthRule = 0;
TimeSpec = classfin(TSpec, 'HJMTimeSpec')
```

```
TimeSpec = struct with fields:
    ValuationDate: 10-Dec-1999
         Maturity: 10-Dec-2002
      Compounding: 2
            Basis: 0
     EndMonthRule: 0
           FinObj: 'HJMTimeSpec'
```

Return a Character Vector Containing a Financial Structure's Class Name

This example shows how to obtain a character vector containing a financial structure's class name.

```
load deriv.mat
ClassName = classfin(HJMTree)
```

```
ClassName =
'HJMFwdTree'
```

Input Arguments

`ClassName` — Name of a financial structure class

character vector

Name of a financial structure class, specified by a character vector.

Data Types: `char`

`Struct` — Structure for conversion

structure

Structure for conversion, specified as a MATLAB structure

Data Types: `struct`

`Obj` — Name of a financial structure

object

Name of a financial structure, specified with an instance of an object.

Data Types: `object`

Output Arguments

`Obj` — Name of a financial structure

object

Name of a financial structure, returned as an instance of an object.

`ClassName` — Name of a financial structure class
character vector

Name of a financial structure class, specified by a character vector.

Version History

Introduced before R2006a

See Also

`isafin`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15
“Hedging” on page 4-2

cirtimespec

Specify time structure for Cox-Ingersoll-Ross tree

Syntax

```
TimeSpec = cirtimespec(ValuationDate,Maturity,NumPeriods)
TimeSpec = cirtimespec( ___ ,Name,Value)
```

Description

`TimeSpec = cirtimespec(ValuationDate,Maturity,NumPeriods)` creates a time spec for a Cox-Ingersoll-Ross (CIR) tree.

`TimeSpec = cirtimespec( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Set the Number of Levels and Node Times for a CIR Tree

Set the number of levels and node times for an CIR tree by specifying a four-period tree with time steps of 1 year.

```
ValuationDate = datetime(2017,1,1);
Maturity = datetime(2021,1,1);
NumPeriods = 4
```

```
NumPeriods = 4
```

```
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods)
```

```
CIRTimeSpec = struct with fields:
           FinObj: 'CIRTimeSpec'
    ValuationDate: 736696
         Maturity: 738157
       NumPeriods: 4
      Compounding: 1
            Basis: 0
     EndMonthRule: 1
             tObs: [0 1 2 3 4]
             dObs: [736696 737061 737426 737791 738157]
```

Input Arguments

`ValuationDate` — Date marking the pricing date and first observation tree

datetime scalar | string scalar | date character vector

Date marking the pricing date and first observation in the tree, specified as a scalar datetime, string, or date character vector

To support existing code, `cirtimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Date marking the depth of tree
datetime scalar | string scalar | date character vector

Date marking the depth of the tree, specified as a scalar datetime, string, or date character vector, .

To support existing code, `cirtimespec` also accepts serial date numbers as inputs, but they are not recommended.

`NumPeriods` — Determines how many time steps are in tree
nonnegative integer

Determines how many time steps are in tree, specified as a scalar using a nonnegative integer value.

Data Types: `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TimeSpec = cirtimespec(Valuationdate,Maturity,NumPeriods,'Basis',3)`

`Compounding` — Frequency at which the rates are compounded when annualized
`1` (default) | integer with value of `1`, `2`, `3`, `4`, `6`, or `12`

Frequency at which the rates are compounded when annualized, specified as the comma-separated pair consisting of `'Compounding'` and a scalar value:

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a scalar.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`TimeSpec` — Time layout for CIR tree
structure

Time layout for the `CIRTree`, returned as a structure.

Version History

Introduced in R2018a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `cirtimespec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cirtree`

cirvolspec

Specify Cox-Ingersoll-Ross interest-rate volatility process

Syntax

```
VolSpec = cirvolspec(Sigma,Alpha,Theta)
```

Description

`VolSpec = cirvolspec(Sigma,Alpha,Theta)` creates a Cox-Ingersoll-Ross (CIR) `VolSpec`.

Examples

Create a Structure Specifying the Volatility for `cirtree`

Create a Cox-Ingersoll-Ross volatility specification (`CIRVolSpec`) using the following data.

```
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
CIRVolSpec = cirvolspec(Sigma,Alpha,Theta)
```

```
CIRVolSpec = struct with fields:
    FinObj: 'CIRVolSpec'
     Sigma: 0.1000
     Alpha: 0.0300
     Theta: 0.0200
```

Input Arguments

`Sigma` — Volatility

numeric

Volatility, specified as a scalar using a numeric value.

Data Types: `double`

`Alpha` — Mean reversion speed

numeric

Mean reversion speed, specified as a scalar using a numeric value.

Data Types: `double`

`Theta` — Mean reversion level or long-term mean of short rate

numeric

Mean reversion level or long-term mean of the short rate, specified as a scalar using a numeric value.

Data Types: `double`

Output Arguments

`VolSpec` — Volatility model for CIR tree

structure

Volatility model for the `CIRTree`, returned as a structure.

Version History

Introduced in R2018a

See Also

`cirtree`

cirtree

Build a Cox-Ingersoll-Ross interest-rate tree

Syntax

```
CIRTree = cirvolspec(VolSpec,RateSpec,TimeSpec)
```

Description

`CIRTree = cirvolspec(VolSpec,RateSpec,TimeSpec)` builds a Cox-Ingersoll-Ross (CIR) interest-rate tree. The CIR tree uses a CIR++ model with the Nawalka-Beliaeva (NB) approach.

Examples

Create a CIR Tree

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = {'Jan-1-2017'; 'Jan-1-2018'; 'Jan-1-2019'; 'Jan-1-2020'; 'Jan-1-2021'};
ValuationDate = 'Jan-1-2017';
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',End
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = '01-Jan-2017';
Maturity = '01-Jan-2021';
CIRTimeSpec = cirtimespec(Settle, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [736696 737061 737426 737791]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Input Arguments

`VolSpec` — Volatility process specification
structure

Volatility process specification, specified using the `VolSpec` output obtained from `cirvolspec`.

Data Types: `struct`

`RateSpec` — Interest-rate specification for initial risk-free rate curve
structure

Interest-rate specification for initial risk-free rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`TimeSpec` — Time tree layout specification
structure

Time tree layout specification, specified using the `TimeSpec` output obtained from `cirtimespec`.

Data Types: `struct`

Output Arguments

`CIRTree` — Time and interest-rate information of a recombining tree
structure

Time and interest-rate information of a recombining tree, returned as a structure.

Version History

Introduced in R2018a

See Also

`cirtimespec` | `cirvolspec`

Topics

“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194

compoundbycrr

Price compound option from Cox-Ross-Rubinstein binomial tree

Syntax

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

Description

`[Price,PriceTree] = compoundbycrr(CRRTree,UOptSpec,UStrike,USettle, UExerciseDates,UAmericanOpt,COptSpec,CStrike,CSettle,CExerciseDates)` prices compound options from a Cox-Ross-Rubinstein binomial tree.

`[Price,PriceTree] = compoundbycrr( ___ ,CAmericanOpt)` adds an optional argument for `CAmericanOpt`.

Examples

Price a Compound Option Using a CRR Binomial Tree

This example shows how to price a compound option using a CRR binomial tree by loading 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

UOptSpec = 'Call';
UStrike = 130;
USettle = datetime(2003,1,1);
UExerciseDates = datetime(2006,1,1);
UAmericanOpt = 1;
COptSpec = 'Put';
CStrike = 5;
CSettle = datetime(2003,1,1);
CExerciseDates = datetime(2005,1,1);

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

```
Price = 2.8482
```

Input Arguments

`CRRTree` — Stock tree structure

structure

Stock tree structure, specified by using `crrtree`.

Data Types: `struct`

`UOptSpec` — Definition of underlying option
character vector with value `'call'` or `'put'`

Definition of underlying option, specified as `'call'` or `'put'` using a character vector.

Data Types: `char`

`UStrike` — Underlying option strike price value
nonnegative integer

Underlying option strike price value, specified with a nonnegative integer using a 1-by-1 vector.

Data Types: `double`

`USettle` — Underlying option settlement date or trade date
datetime array | string array | date character vector

Underlying option settlement date or trade date, specified as a 1-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `compoundbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`UExerciseDates` — Underlying option exercise date
datetime array | string array | date character vector

Underlying option exercise date, specified as a datetime array, string array, or date character vectors:

- For a European option, use a1-by-1 vector of the underlying exercise date. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a 1-by-2 vector of the underlying 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 `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `compoundbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`UAmericanOpt` — Underlying option type
`0` European (default) | scalar with values `0` or `1`

Underlying option type, specified as `NINST`-by-1 positive integer scalar flags with values:

- `0` — European
- `1` — American

If `UAmericanOpt` is a `NaN` or is unspecified, the option is a European option.

Data Types: `double`

`COptSpec` — Definition of compound option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of compound option, specified as `'call'` or `'put'` using a character vector or a cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`CStrike` — Compound option strike price values
nonnegative integers

Compound option strike price values for a European and American option, specified with a nonnegative integer using a `NINST`-by-`1` matrix. Each row is the schedule for one option.

Data Types: `double`

`CSettle` — Compound option settlement date or trade date
datetime array | string array | date character vector

Compound option settlement date or trade date, specified as a `1`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `compoundbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`CExerciseDates` — Compound option exercise dates
datetime array | string array | date character vector

Compound option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a`NINST`-by-`1` matrix of the compound exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of the compound 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 `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `compoundbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`CAmericanOpt` — Compound option type
`0` European (default) | scalar with values `0` or `1`

(Optional) Compound option type, specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

If `CAmericanOpt` is a `NaN` or is unspecified, the option is a European option.

Data Types: `double`

Output Arguments

`Price` — Expected prices for compound options at time `0`

vector

Expected prices for compound options at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Structure with vector of compound option prices at each node

tree structure

Structure with a vector of compound option prices at each node, returned as a tree structure.

`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.

More About

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. For more information, see “Compound Option” on page 3-23.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `compoundbycrr` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Rubinstein, Mark. “Double Trouble.” *Risk*. Vol. 5, 1991, p. 73.

See Also

`crrtree` | `instcompound`

Topics

“Computing Prices Using CRR” on page 3-65
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Compound Option” on page 3-23
“Supported Equity Derivative Functions” on page 3-19

compoundbyeqp

Price compound option from Equal Probabilities binomial tree

Syntax

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

Description

`[Price,PriceTree] = compoundbyeqp(EQPTree,UOptSpec,UStrike,USettle, UExerciseDates,UAmericanOpt,COptSpec,CStrike,CSettle,CExerciseDates)` prices compound options from a Equal Probabilities binomial tree.

`[Price,PriceTree] = compoundbyeqp( ___ ,CAmericanOpt)` adds an optional argument for `CAmericanOpt`.

Examples

Price a Compound Option Using an EQP Equity Tree

This example shows how to price a compound option using a EQP equity tree by loading 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
UOptSpec = 'Call';
UStrike = 130;
USettle = datetime(2003,1,1);
UExerciseDates = datetime(2006,1,1);
UAmericanOpt = 1;
COptSpec = 'Put';
CStrike = 5;
CSettle = datetime(2003,1,1);
CExerciseDates = datetime(2005,1,1);

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

```
Price = 3.3931
```

Input Arguments

EQPTree — Stock tree structure

structure

Stock tree structure, specified by using `eqptree`.

Data Types: `struct`

`UOptSpec` — Definition of underlying option
character vector with value `'call'` or `'put'`

Definition of underlying option, specified as `'call'` or `'put'` using a character vector.

Data Types: `char`

`UStrike` — Underlying option strike price value
nonnegative integer

Underlying option strike price value, specified with a nonnegative integer using a 1-by-1 vector.

Data Types: `double`

`USettle` — Underlying option settlement date or trade date
datetime array | string array | date character vector

Underlying option settlement date or trade date, specified as a 1-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `compoundbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`UExerciseDates` — Underlying option exercise date
datetime array | string array | date character vector

Underlying option exercise date, specified as a datetime array, string array, or date character vectors:

- For a European option, use a1-by-1 vector of the underlying exercise date. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a 1-by-2 vector of the underlying 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 `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `compoundbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`UAmericanOpt` — Underlying option type
`0` European (default) | scalar with values `0` or `1`

Underlying option type, specified as `NINST`-by-1 positive integer scalar flags with values:

- `0` — European
- `1` — American

If `UAmericanOpt` is a `NaN` or is unspecified, the option is a European option.

Data Types: `double`

`COptSpec` — Definition of compound option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of compound option, specified as `'call'` or `'put'` using a character vector or a cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`CStrike` — Compound option strike price values
nonnegative integers

Compound option strike price values for a European and American option, specified with a nonnegative integer using a `NINST`-by-`1` matrix. Each row is the schedule for one option.

Data Types: `double`

`CSettle` — Compound option settlement date or trade date
datetime array | string array | date character vector

Compound option settlement date or trade date, specified as a `1`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `compoundbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`CExerciseDates` — Compound option exercise dates
datetime array | string array | date character vector

Compound option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a`NINST`-by-`1` matrix of the compound exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of the compound 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 `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `compoundbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`CAmericanOpt` — Compound option type
`0` European (default) | scalar with values `0` or `1`

(Optional) Compound option type, specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

If `CAmericanOpt` is a `NaN` or is unspecified, the option is a European option.

Data Types: `double`

Output Arguments

`Price` — Expected prices for compound options at time `0`

vector

Expected prices for compound options at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Structure with vector of compound option prices at each node

tree structure

Structure with a vector of compound option prices at each node, returned as a tree structure.

`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.

More About

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. For more information, see “Compound Option” on page 3-23.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `compoundbyeqp` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Rubinstein, Mark. “Double Trouble.” *Risk*. Vol. 5, 1991, p. 73.

See Also

`eqptree` | `instcompound`

Topics

“Computing Prices Using CRR” on page 3-65
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Compound Option” on page 3-23
“Supported Equity Derivative Functions” on page 3-19

compoundbyitt

Price compound option from implied trinomial tree (ITT)

Syntax

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

Description

`[Price,PriceTree] = compoundbyitt(ITTTree,UOptSpec,UStrike,USettle, UExerciseDates,UAmericanOpt,COptSpec,CStrike,CSettle,CExerciseDates)` prices compound options from a Equal Probabilities binomial tree.

`[Price,PriceTree] = compoundbyitt( ___ ,CAmericanOpt)` adds an optional argument for `CAmericanOpt`.

Examples

Price a Compound Option Using an ITT Tree

This example shows how to price a compound option using a ITT tree by loading the file `deriv.mat`, which provides `ITTTree`. The `ITTTree` structure contains the stock specification and time information needed to price the option.

```
load deriv.mat

UOptSpec = 'Call';
UStrike = 99;
USettle = datetime(2006,1,1);
UExerciseDates = datetime(2010,1,1);
UAmericanOpt = 1;
COptSpec = 'Put';
CStrike = 5;
CSettle = datetime(2006,1,1);
CExerciseDates = datetime(2010,1,1);

Price = compoundbyitt(ITTTree, UOptSpec, UStrike, USettle, ...
UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, ...
CExerciseDates)

Price = 2.7271
```

Input Arguments

`ITTTree` — Stock tree structure

structure

Stock tree structure, specified by using `itttree`.

Data Types: `struct`

`UOptSpec` — Definition of underlying option
character vector with value `'call'` or `'put'`

Definition of underlying option, specified as `'call'` or `'put'` using a character vector.

Data Types: `char`

`UStrike` — Underlying option strike price value
nonnegative integer

Underlying option strike price value, specified with a nonnegative integer using a 1-by-1 vector.

Data Types: `double`

`USettle` — Underlying option settlement date or trade date
datetime array | string array | date character vector

Underlying option settlement date or trade date, specified as a 1-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `compoundbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`UExerciseDates` — Underlying option exercise date
datetime array | string array | date character vector

Underlying option exercise date, specified as a datetime array, string array, or date character vectors:

- For a European option, use a1-by-1 vector of the underlying exercise date. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a 1-by-2 vector of the underlying 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 `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `compoundbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`UAmericanOpt` — Underlying option type
`0` European (default) | scalar with values `0` or `1`

Underlying option type, specified as `NINST`-by-1 positive integer scalar flags with values:

- `0` — European
- `1` — American

If `UAmericanOpt` is a `NaN` or is unspecified, the option is a European option.

Data Types: `double`

`COptSpec` — Definition of compound option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of compound option, specified as `'call'` or `'put'` using a character vector or a cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`CStrike` — Compound option strike price values
nonnegative integers

Compound option strike price values for a European and American option, specified with a nonnegative integer using a `NINST`-by-`1` matrix. Each row is the schedule for one option.

Data Types: `double`

`CSettle` — Compound option settlement date or trade date
datetime array | string array | date character vector

Compound option settlement date or trade date, specified as a `1`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `compoundbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`CExerciseDates` — Compound option exercise dates
datetime array | string array | date character vector

Compound option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a`NINST`-by-`1` matrix of the compound exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of the compound 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 `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `compoundbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`CAmericanOpt` — Compound option type
`0` European (default) | scalar with values `0` or `1`

(Optional) Compound option type, specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

If `CAmericanOpt` is a `NaN` or is unspecified, the option is a European option.

Data Types: `double`

Output Arguments

`Price` — Expected prices for compound options at time `0`

vector

Expected prices for compound options at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Structure with vector of compound option prices at each node

tree structure

Structure with a vector of compound option prices at each node, returned as a tree structure.

`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.

More About

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. For more information, see “Compound Option” on page 3-23.

Version History

Introduced in R2007a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `compoundbyitt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Rubinstein, Mark. “Double Trouble.” *Risk*. Vol. 5, 1991, p. 73.

See Also

`itttree` | `instcompound`

Topics

“Computing Prices Using CRR” on page 3-65
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Compound Option” on page 3-23
“Supported Equity Derivative Functions” on page 3-19

compoundbystt

Price compound options using standard trinomial tree

Syntax

```
[Price,PriceTree] = compoundbystt(STTTree,UOptSpec,UStrike,USettle,
UExerciseDates,UAmericanOpt,COptSpec,CStrike,CSettle,CExerciseDates)
[Price,PriceTree] = compoundbystt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = compoundbystt(STTTree,UOptSpec,UStrike,USettle, UExerciseDates,UAmericanOpt,COptSpec,CStrike,CSettle,CExerciseDates)` prices compound options using a standard trinomial (STT) tree.

`[Price,PriceTree] = compoundbystt( ___ ,Name,Value)` adds an optional name-value pair argument for `CAmericanOpt`.

Examples

Price a Compound Option Using the Standard Trinomial Tree Model

Create a `RateSpec`.

```
StartDates = datetime(2009,1,1);
EndDates = datetime(2013,1,1);
Rates = 0.035;
Basis = 1;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8694
            Rates: 0.0350
         EndTimes: 4
       StartTimes: 0
         EndDates: 735235
       StartDates: 733774
    ValuationDate: 733774
            Basis: 1
     EndMonthRule: 1
```

Create a `StockSpec`.

```
AssetPrice = 85;
Sigma = 0.15;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1500
         AssetPrice: 85
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Create an STTTree.

```
NumPeriods = 4;
TimeSpec = stttimespec(StartDates, EndDates, 4);
STTTree = stttree(StockSpec, RateSpec, TimeSpec)
```

```
STTTree = struct with fields:
       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 1 2 3 4]
         dObs: [733774 734139 734504 734869 735235]
        STree: {1x5 cell}
        Probs: {[3x1 double]  [3x3 double]  [3x5 double]  [3x7 double]}
```

Define the compound option and compute the price.

```
USettle = datetime(2009,1,1);
UExerciseDates = datetime(2012,1,1);
UOptSpec =  'call';
UStrike = 95;
UAmericanOpt = 1;
CSettle = datetime(2009,1,1);
CExerciseDates = datetime(2011,1,1);
COptSpec = 'put';
CStrike = 5;
CAmericanOpt = 1;

Price= compoundbystt(STTTree, UOptSpec, UStrike, USettle, UExerciseDates,...
UAmericanOpt, COptSpec, CStrike, CSettle,CExerciseDates, CAmericanOpt)
```

```
Price = 1.7090
```

Input Arguments

STTTree — Stock tree structure for standard trinomial tree

structure

Stock tree structure for standard trinomial tree, specified by using `stttree`.

Data Types: `struct`

UOptSpec — Definition of underlying option

character vector with value `'call'` or `'put'`

Definition of underlying option, specified as `'call'` or `'put'` using a character vector.

Data Types: `char`

`UStrike` — Underlying option strike price value
nonnegative integer

Underlying option strike price value, specified with a nonnegative integer using a 1-by-1 vector.

Data Types: `double`

`USettle` — Underlying option settlement date or trade date
datetime array | string array | date character vector

Underlying option settlement date or trade date, specified as a 1-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `compoundbystt` also accepts serial date numbers as inputs, but they are not recommended.

`UExerciseDates` — Underlying option exercise date
datetime array | string array | date character vector

Underlying option exercise date, specified as a datetime array, string array, or date character vectors:

- For a European option, use a1-by-1 vector of the underlying exercise date. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a 1-by-2 vector of the underlying 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 `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `compoundbystt` also accepts serial date numbers as inputs, but they are not recommended.

`UAmericanOpt` — Underlying option type
`0` European (default) | scalar with values `0` or `1`

Underlying option type, specified as `NINST`-by-1 positive integer scalar flags with values:

- `0` — European
- `1` — American

If `UAmericanOpt` is a `NaN` or is unspecified, the option is a European option.

Data Types: `single` | `double`

`COptSpec` — Definition of compound option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of compound option, specified as `'call'` or `'put'` using a character vector or a cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`CStrike` — Compound option strike price values
nonnegative integers

Compound option strike price values for a European and American option, specified with a nonnegative integer using a `NINST`-by-`1` matrix. Each row is the schedule for one option.

Data Types: `double`

`CSettle` — Compound option settlement date or trade date
datetime array | string array | date character vector

Compound option settlement date or trade date, specified as a `1`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `compoundbystt` also accepts serial date numbers as inputs, but they are not recommended.

`CExerciseDates` — Compound option exercise dates
datetime array | string array | date character vector

Compound option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a`NINST`-by-`1` matrix of the compound exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of the compound 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 `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `compoundbystt` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = compoundbystt(STTTree,UOptSpec,UStrike,USettle,UExerciseDates,UAmericanOpt,COptSpec,CStrike,CSettle,CExerciseDates,'CAmericanOpt',1)`

`CAmericanOpt` — Compound option type
`0` European (default) | scalar with values `[0,1]`

Compound option type, specified as the comma-separated pair consisting of `'CAmericanOpt'` and a `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

Data Types: `single` | `double`

Output Arguments

`Price` — Expected prices for compound options at time `0`

vector

Expected prices for compound options at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Structure with vector of compound option prices at each node

tree structure

Structure with a vector of compound option prices at each node, returned as a tree structure.

`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.

More About

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. For more information, see “Compound Option” on page 3-23.

Version History

Introduced in R2015b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `compoundbystt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`stttimespec` | `stttree` | `sttprice` | `sttsens` | `instcompound`

Topics

“Compound Option” on page 3-23
“Supported Equity Derivative Functions” on page 3-19

crrprice

Instrument prices from Cox-Ross-Rubinstein tree

Syntax

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

Description

`[Price,PriceTree] = crrprice(CRRTree,InstSet)` computes stock option prices using a CRR binomial tree created with `crrtree`. All instruments contained in a financial instrument variable, `InstSet`, are priced.

`crrprice` handles instrument types: `'Asian'`, `'Barrier'`, `'Compound'`, `'CBond'`, `'Lookback'`, `'OptStock'`. See `instadd` to construct defined types.

`[Price,PriceTree] = crrprice( ___ ,Options)` adds an optional input argument for `Options`.

Examples

Price Barrier and Lookback Options in the Instrument Set

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 Barrier Rebate
1     Barrier call    105    01-Jan-2003    01-Jan-2006    1           ui          102     0

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
```

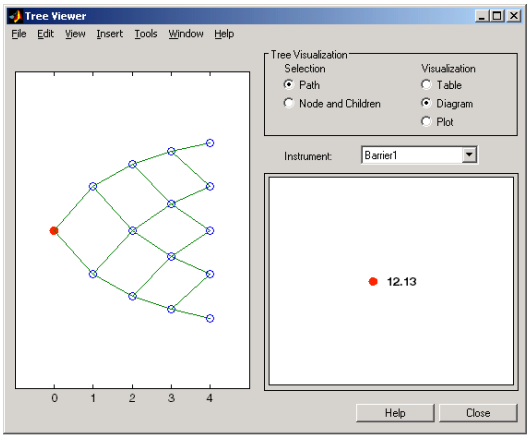
Price the barrier and lookback options.

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

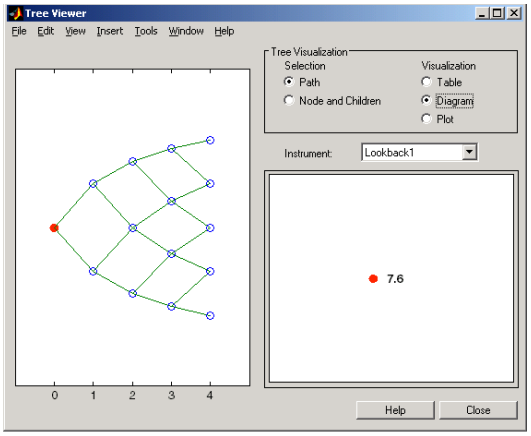
```
Price = 3×1

   12.1272
    7.6015
   11.7772
```

```
PriceTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {1x5 cell}
      tObs: [0 1 2 3 4]
      dObs: [731582 731947 732313 732678 733043]
```

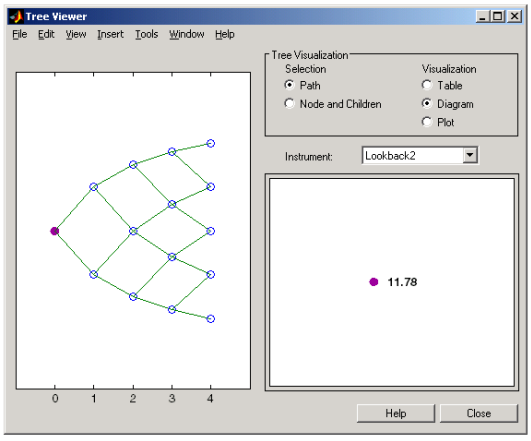
You can use `treeviewer` to see the prices of these three instruments along the price tree.



Barrier1



Lookback1



Lookback2

Input Arguments

CRRTree — Stock price tree structure

structure

Stock price tree structure, specified by using `crrtree`.

Data Types: `struct`

`InstSet` — Instrument variable
structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Price for each instrument
vector

Price for each instrument, returned as a `NINST`-by-`1` vector. 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.

Related single-type pricing functions are:

- `asianbycrr`: Price an Asian option from a CRR tree.
- `barrierbycrr`: Price a barrier option from a CRR tree.
- `cbondbycrr`: Price convertible bonds 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.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.tObs` contains the observation times.
- `PriceTree.dObs` contains the observation dates.

Version History

Introduced before R2006a

See Also

`crrsens` | `crrtree` | `instadd` | `instcbond` | `cbondbycrr`

Topics

“Computing Prices Using CRR” on page 3-65
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

crrsens

Instrument prices and sensitivities from Cox-Ross-Rubinstein tree

Syntax

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

Description

`[Delta,Gamma,Vega,Price] = crrsens(CRRTree,InstSet)` computes instrument sensitivities and prices for instruments using a binomial tree created with the `crrtree` function. All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide by the respective instrument price.

`crrsens` handles instrument types: `'Asian'`, `'Barrier'`, `'Compound'`, `'CBond'`, `'Lookback'`, `'OptStock'`. See `instadd` for information on instrument types.

`[Delta,Gamma,Vega,Price] = crrsens(___,Options)` adds an optional input argument for `Options`.

Examples

Compute Sensitivities for Barrier and Lookback Instruments Using a crrtree

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 Barrier Rebate
1     Barrier call    105    01-Jan-2003    01-Jan-2006    1           ui          102     0

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
```

Obtain the `Delta` and `Gamma` for the barrier and lookback options contained in the instrument set.

```
[Delta, Gamma] = crrsens(CRRTree, CRRSubSet)
```

```
Delta = 3×1

    0.6885
    0.6049
```

```
    0.8187


Gamma = 3×1

    0.0310
   -0.0000
         0
```

Input Arguments

`CRRTree` — Stock tree structure

structure

Stock tree structure, specified by using `crrtree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Delta` — Rate of change of instruments prices with respect to changes in the stock price

vector

Rate of change of instruments prices with respect to changes in the stock price, returned as a `NINST`-by-`1` vector of deltas.

For path-dependent options (`'Lookback'` and `'Asian'`), `Delta` and `Gamma` are computed by finite differences in calls to `crrprice`. For the rest of the options (`'OptStock'`, `'Barrier'`, `'CBond'`, and `'Compound'`), `Delta` and `Gamma` are computed from the `CRRTree` and the corresponding option price tree.

`Gamma` — Rate of change of instruments deltas with respect to changes in stock price

vector

Rate of change of instruments deltas with respect to changes in the stock price, returned as a `NINST`-by-`1` vector of gammas.

For path-dependent options (`'Lookback'` and `'Asian'`), `Delta` and `Gamma` are computed by finite differences in calls to `crrprice`. For the rest of the options (`'OptStock'`, `'Barrier'`, `'CBond'`,

and 'Compound'), Delta and Gamma are computed from the CRRTree and the corresponding option price tree.

Vega — Rate of change of instruments prices with respect to changes in volatility of the stock
vector

Rate of change of instruments prices with respect to changes in the volatility of the stock, returned as a NINST-by-1 vector of vegas. Vega is computed by finite differences in calls to crrtree.

Price — Price of each instrument
vector

Price of each instrument, returned as a NINST-by-1 vector. 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.

Version History

Introduced before R2006a

References

[1] Chriss, Neil. *Black-Scholes and Beyond: Option Pricing Models.* McGraw-Hill, 1996, pp 308-312.

See Also

crrprice | crrtree | cbondbycrr | instcbond

Topics
“Computing Prices Using CRR” on page 3-65
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

crrtimespec

Specify time structure for Cox-Ross-Rubinstein tree

Syntax

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

Description

`TimeSpec = crrtimespec(ValuationDate,Maturity,NumPeriods)` sets the number of levels and node times for a CRR binomial tree (`crrtree`).

Examples

Set the Number of Levels and Node Times for a CRR Binomial Tree

This example shows how to specify a four-period CRR tree with time steps of 1 year.

```
ValuationDate = datetime(2002,7,1);
Maturity = datetime(2006,7,1);
TimeSpec = crrtimespec(ValuationDate, Maturity, 4)
```

```
TimeSpec = struct with fields:
           FinObj: 'BinTimeSpec'
    ValuationDate: 731398
         Maturity: 732859
       NumPeriods: 4
            Basis: 0
     EndMonthRule: 1
             tObs: [0 1 2 3 4]
             dObs: [731398 731763 732128 732493 732859]
```

Input Arguments

`ValuationDate` — Pricing date and first observation in the tree

datetime scalar | string scalar | date character vector

Pricing date and first observation in the `crrtree`, specified as a scalar datetime, string, or date character vector.

To support existing code, `crrtimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Date marking the depth of the CRR stock tree

datetime scalar | string scalar | date character vector

Date marking the depth of the `crrtree` binomial tree, specified as scalar datetime, string, or date character vector.

To support existing code, `crrtimespec` also accepts serial date numbers as inputs, but they are not recommended.

`NumPeriods` — Number of time steps in the CRR stock tree
integer

Number of time steps in the `crrtree` binomial tree, specified as scalar integer value.

Data Types: `double`

Output Arguments

`TimeSpec` — Specification for the time layout for `crrtree`
structure

Specification for the time layout for `crrtree`, returned as a structure.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `crrtimespec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`crrtree` | `stockspec`

Topics
“Building Equity Binary Trees” on page 3-3
“Examining Equity Trees” on page 3-14
“Understanding Equity Trees” on page 3-2
“Differences Between CRR and EQP Tree Structures” on page 3-17
“Supported Equity Derivative Functions” on page 3-19

crrtree

Build Cox-Ross-Rubinstein stock tree

Syntax

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

Description

`CRRTree = crrtree(StockSpec,RateSpec,TimeSpec)` builds a Cox-Ross-Rubinstein stock tree.

Examples

Create a CRR Tree

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);

RateSpec = intenvset('Rates', 0.05, 'StartDates',...
'01-Jan-2003', 'EndDates', '31-Dec-2003','Compounding',-1);

ValuationDate = '1-Jan-2003';
Maturity = '31-Dec-2003';
TimeSpec = crrtimespec(ValuationDate, Maturity, 4);

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

```
CRRTree = struct with fields:
       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 731673 731764 731855 731946]
        STree: {1x5 cell}
      UpProbs: [0.5378 0.5378 0.5378 0.5378]
```

Input Arguments

StockSpec — Stock specification

structure

Stock specification, specified by the `StockSpec` obtained from `stockspec`. See `stockspec` for information on creating a stock specification.

Data Types: `struct`

RateSpec — Interest-rate specification for initial risk-free rate curve

structure

Interest-rate specification for initial risk-free rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

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 is not a standard CRR tree.

Data Types: `struct`

TimeSpec — Tree time layout specification

structure

Tree time layout specification, specified by the `TimeSpec` obtained from `crrtimespec`. The `TimeSpec` defines the observation dates of the CRR binomial tree. See `crrtimespec` for information on the tree structure.

Data Types: `struct`

Output Arguments

CRRTree — CRR binomial tree

structure

CRR binomial tree, returned as a structure specifying the time layout for the tree.

Version History

Introduced before R2006a

See Also

`crrtimespec` | `intenvset` | `stockspec`

Topics

“Building Equity Binary Trees” on page 3-3
“Examining Equity Trees” on page 3-14
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Understanding Equity Trees” on page 3-2

“Differences Between CRR and EQP Tree Structures” on page 3-17
“Supported Equity Derivative Functions” on page 3-19

cvtree

Convert inverse-discount tree to interest-rate tree

Syntax

```
RateTree = cvtree(Tree)
```

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

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

```
load deriv.mat;
HWTree
```

```
HWTree = struct with fields:
      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}
```

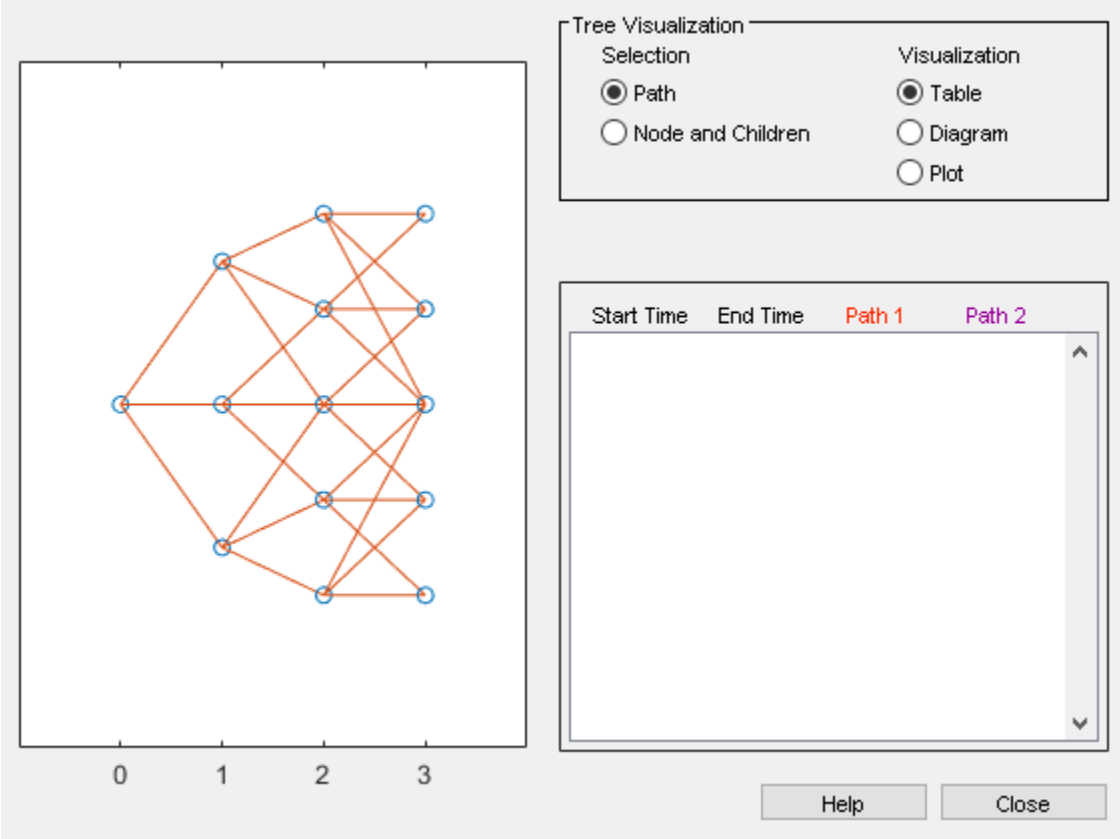
```
ans = 1.0279
```

```
HWTree.FwdTree{2}
```

```
ans = 1×3

    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 = struct with fields:
      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}
```

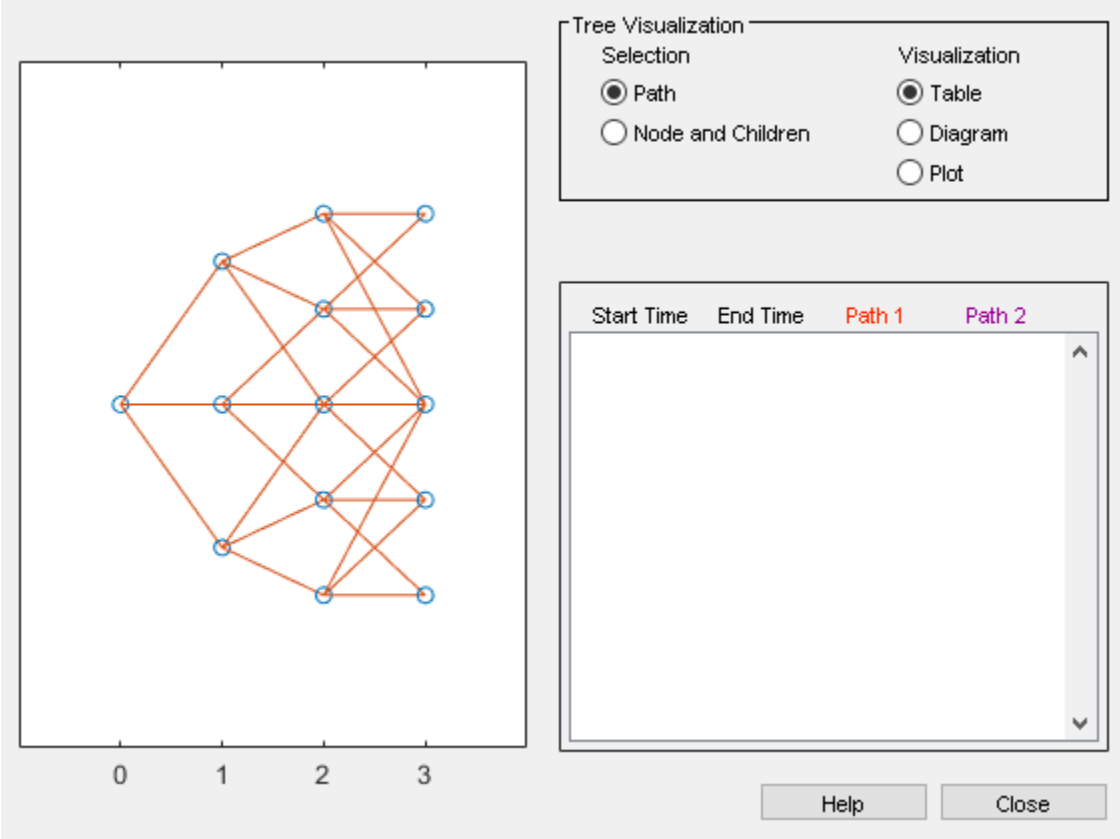
```
ans = 0.0275
```

```
RTree.RateTree{2}
```

```
ans = 1×3

    0.0514    0.0349    0.0185
```

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

```
treeviewer(RTree)
```



Input Arguments

`Tree` — Tree structure

HJM Tree | BDT Tree | HW Tree | BK Tree | CIR Tree

Tree structure, specified by Heath-Jarrow-Morton, Black-Derman-Toy, Hull-White, Black-Karasinski, or Cox-Ingersoll-Ross tree structure that uses inverse-discount notation for forward rates.

Data Types: `struct`

Output Arguments

`RateTree` — Tree structure using rate notation for forward rates

structure

Tree structure using rate notation for forward rates, returned as a tree structure.

Version History

Introduced before R2006a

See Also

`disc2rate` | `rate2disc`

Topics

“Graphical Representation of Trees” on page 2-219

date2time

Time and frequency from dates

Syntax

```
[Times,F] = date2time(Settle,Maturity)
[Times,F] = date2time( ___ ,Compounding,Basis,EndMonthRule)
```

Description

`[Times,F] = date2time(Settle,Maturity)` computes time factors appropriate to compounded rate quotes beyond the settlement date.

`[Times,F] = date2time( ___ ,Compounding,Basis,EndMonthRule)` add additional optional arguments.

Examples

Time and Frequency from Dates

This example shows how to compute time and frequency from dates.

```
Settle = datetime(2002,9,1);
Maturity = [datetime(2005,8,31) ; datetime(2006,2,28) ; datetime(2006,6,15) ; datetime(2006,12,31
Compounding = 2;
Basis = 0;
EndMonthRule = 1;
Times = date2time(Settle, Maturity, Compounding, Basis, EndMonthRule)
```

```
Times = 4×1

    5.9945
    6.9945
    7.5738
    8.6576
```

Input Arguments

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `date2time` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity dates

datetime array | string array | date character vector

Maturity dates, specified as a scalar or an N-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `date2time` also accepts serial date numbers as inputs, but they are not recommended.

`Compounding` — Rate at which the input zero rates were compounded when annualized
2 (default) | integer with value of 1, 2, 3, 4, 6, 12, 365, or -1

(Optional) Rate at which the input zero rates were compounded when annualized, specified as a scalar integer value.

- If `Compounding` = 1, 2, 3, 4, 6, 12:

  `Disc = (1 + Z/F)^(-T)`, where F is the compounding frequency, Z is the zero rate, and T is the time in periodic units; for example, `T = F` is one year.
- If `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.
- If `Compounding` = −1:

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

Data Types: `double`

`Basis` — Day-count basis
0 (actual/actual) (default) | integer from 0 to 13

(Optional) Day-count basis, specified as a scalar or an N-by-1 vector using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

EndMonthRule — End-of-month rule flag
1 (in effect) (default) | nonnegative integer [0,1]

(Optional) End-of-month rule flag, specified as a scalar or an N-by-1 vector of end-of-month rules.

- 0 = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- 1 = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: logical

Output Arguments

Times — Time factors appropriate to compounded rate quotes between Settle and Maturity dates
vector

Time factors appropriate to compounded rate quotes between Settle and Maturity dates, returned as an N-by-1 vector.

F — Frequency
numeric

Frequency, returned as a scalar of related compounding frequencies.

Note To obtain accurate results from this function, the Basis and Maturity arguments must be consistent. If the Maturity 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.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although date2time supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cftimes` | `disc2rate` | `rate2disc` | `time2date`

Topics

“Modeling the Interest-Rate Term Structure” on page 2-57
“Interest-Rate Term Conversions” on page 2-53
“Interest Rates Versus Discount Factors” on page 2-48
“Graphical Representation of Trees” on page 2-219
“Understanding the Interest-Rate Term Structure” on page 2-48

datedisp

Display date entries

Syntax

```
datedisp(NumMat,DateForm)
CharMat = datedisp(NumMat,DateForm)
```

Description

`datedisp(NumMat,DateForm)` displays the matrix with the serial dates formatted as date character vectors, 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 = datedisp(NumMat,DateForm)` displays the output matrix `CharMat`.

Examples

Display Date Entries

This example shows how to display dates for serial date numbers.

```
NumMat = [ 730730, 0.03, 1200, 730100;
           730731, 0.05, 1000, NaN]
```

```
NumMat = 2×4
10^5 ×

    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
```

Input Arguments

`NumMat` — Numeric matrix to display

numeric

Numeric matrix to display, specified as numeric values for serial date numbers.

Data Types: `double`

`DateForm` — date format

character vector

(Optional) Date format, specified as a character vector. See `datestr` for available and default format flags.

Data Types: `char`

Output Arguments

`CharMat` — Character array representing the matrix

array

Character array representing the matrix, returned as an array. If no output variable is assigned, `datedisp` prints the array to the display.

Tips

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

Version History

Introduced before R2006a

See Also

`datenum` | `datestr`

Topics

“Modeling the Interest-Rate Term Structure” on page 2-57
“Interest-Rate Term Conversions” on page 2-53
“Interest Rates Versus Discount Factors” on page 2-48
“Graphical Representation of Trees” on page 2-219
“Understanding the Interest-Rate Term Structure” on page 2-48

derivget

Get derivatives pricing options

Syntax

```
Value = derivget(Options,ParameterName)
```

Description

`Value = derivget(Options,ParameterName)` extracts the value of the specified `ParameterName` argument values from the derivative `Options` structure.

Examples

Get Derivatives Pricing Options

Enable the display of additional diagnostic information that appears when executing pricing functions

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

```
Options = struct with fields:
      Diagnostics: 'on'
         Warnings: 'on'
        ConstRate: 'on'
    BarrierMethod: 'unenhanced'
```

Use `derivget` to extract the value of `Diagnostics` from the `Options` structure.

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

```
Value =
'on'
```

Use `derivget` to extract the value of `ConstRate`.

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

```
Value =
'on'
```

If the value of `'ConstRate'` is not previously set with `derivset`, the answer represents the default setting for `'ConstRate'`.

Find the value of `'BarrierMethod'` in this Options structure.

```
derivget(Options ,'BarrierMethod')
```

```
ans =
'unenhanced'
```

Input Arguments

`Options` — Existing options specification

structure

Existing options specification, specified as a structure obtained from a previous call to `derivset`.

Data Types: `struct`

`ParameterName` — Parameter name to be accessed in `Options` structure

character vector with value of `'Diagnostics'`, `'Warnings'`, `'ConstRate'`, or `'BarrierMethod'`

Parameter name to be accessed in `Options` structure, specified as a character vector for one of the following:

- `'Diagnostics'` — Print diagnostic information with a returned value of `'on'` or `'off'`. This option applies only for HJM, BDT, HW and BK pricing.
- `'Warnings'` — Display warnings with a returned value of `'on'` or `'off'`. This option applies only for HJM, BDT, HW and BK pricing.
- `'ConstRate'` — Assume constant rates between tree nodes with a returned value of `'on'` or `'off'`. This option applies only for HJM, BDT, HW and BK pricing.
- `BarrierMethod` — Method for pricing Barrier option. The returned values are either `'unenhanced'` that uses no correction calculation or `'interp'` that uses an enhanced valuation interpolating between nodes on barrier boundaries.

Data Types: `char`

Output Arguments

`Value` — Value

character vector

Value, returned as a character vector depending on the specified `ParameterName`.

Version History

Introduced before R2006a

See Also

`barrierbycrr` | `barrierbyeqp` | `derivset`

Topics

“Computing Instrument Prices” on page 2-81
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2

derivset

Set or modify derivatives pricing options

Syntax

```
Options = derivset(Options,Name,Value)
Options = derivset(Name,Value)
Options = derivset(OldOptions,NewOptions)
Options = derivset
derivset
```

Description

`Options = derivset(Options,Name,Value)` modifies an existing derivatives pricing options structure `Options` by changing the specified name-value pair argument values.

`Options = derivset(Name,Value)` creates a derivatives pricing options structure `Options` using the specified name-value pair argument values. Any unspecified name-value arguments are set to default values for when the `Options` output is passed to the derivatives function.

`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` where all the parameters are set to the default values.

`derivset` with no input or output arguments displays all parameter names and information about their possible values.

Examples

Set or Modify Derivatives Pricing Options

Enable the display of additional diagnostic information that appears when executing pricing functions

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

```
Options = struct with fields:
      Diagnostics: 'on'
         Warnings: 'on'
        ConstRate: 'on'
    BarrierMethod: 'unenhanced'
```

Change the `ConstRate` parameter in the existing `Options` structure so that the assumption of constant rates between tree nodes no longer applies.

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

```
Options = struct with fields:
      Diagnostics: 'on'
         Warnings: 'on'
        ConstRate: 'off'
    BarrierMethod: 'unenhanced'
```

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 ]
```

Input Arguments

`Options` — Existing options specification

structure

Existing options specification, specified as a structure obtained from a previous call to `derivset`.

Data Types: `struct`

`OldOptions` — Existing options specification

structure

Existing options specification, specified as a structure obtained from a previous call to `derivset`.

Data Types: `struct`

`NewOptions` — New options specification

structure

New options specification, specified as a structure.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Options = derivset(Options, 'ConstRate', 'off')`

`Diagnostics` — Print diagnostic information

character vectors with value `'on'` or `'off'`

Print diagnostic information, specified as the comma-separated pair consisting of `'Diagnostics'` and a character vector. This option applies only for HJM, BDT, HW and BK pricing.

Data Types: `char`

`Warnings` — Display warnings
character vectors with value `'on'` or `'off'`

Display warnings, specified as the comma-separated pair consisting of `'Warnings'` and a character vector. This option applies only for HJM, BDT, HW and BK pricing.

Data Types: `char`

`ConstRate` — Assume constant rates between tree nodes
character vectors with value `'on'` or `'off'`

Assume constant rates between tree nodes, specified as the comma-separated pair consisting of `'ConstRate'` and a character vector. This option applies only for HJM, BDT, HW and BK pricing.

Data Types: `char`

`BarrierMethod` — Method for pricing Barrier option
`'unenhanced'` (default) | character vector with value of `'unenhanced'` or `'interp'`

Method for pricing Barrier option, specified as the comma-separated pair consisting of `'BarrierMethod'` and a character vector. Specifying `'unenhanced'` uses no correction calculation. Specifying `'interp'` uses an enhanced valuation interpolating between nodes on barrier boundaries.

Data Types: `char`

Output Arguments

`Options` — Options specification
structure

Options specification, returned as a structure encapsulating the properties of a derivatives option.

Version History

Introduced before R2006a

See Also

`barrierbycrr` | `barrierbyeqp` | `derivget`

Topics
“Computing Instrument Prices” on page 2-81
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2

disc2rate

Interest rates from cash flow discounting factors

Syntax

```
Rates = disc2rate(Compounding,Disc,EndTimes,StartTimes)
[Rates,EndTimes,StartTimes] = disc2rate(Compounding,Disc,EndTimes,StartTimes,
ValuationDate,Basis,EndMonthRule)
```

Description

`Rates = disc2rate(Compounding,Disc,EndTimes,StartTimes)` computes interest rates from discount factors where interval points are input as times in periodic units.

`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.

The output `Rates` is an `NPOINTS`-by-`NCURVES` column vector of yields in decimal form over the `NPOINTS` time intervals.

`[Rates,EndTimes,StartTimes] = disc2rate(Compounding,Disc,EndTimes,StartTimes,
ValuationDate,Basis,EndMonthRule)` computes interest rates from discount factors where `ValuationDate` is passed and interval points are input as dates.

You can specify the investment intervals either with input times or with input dates. Entering `ValuationDate` invokes the date interpretation; omitting `ValuationDate` invokes the default time interpretations.

Examples

Compute Interest Rates from Cash Flow Discounting Factors

This example shows the two uses of `disc2rate`.

Interval Points Are Input as Times in Periodic Units

Compute rates from a zero curve at 6 months, 12 months, and 24 months, given the discount factors for these periods. The times to the cash flows are 1, 2, and 4. `disc2rate` assumes that the valuation date corresponds to time = 0.

```
Compounding = 2;
Disc = [0.9756; 0.9426; 0.8799];
EndTimes   = [1; 2; 4];
Rates = disc2rate(Compounding, Disc, EndTimes)
```

```
Rates = 3×1

    0.0500
    0.0600
```

```
    0.0650
```

Interval Points Are Input as Dates

Compute rates from a zero curve at 6 months, 12 months, and 24 months, given the discount factors for these periods. Use dates to specify the ending time horizon.

```
Compounding = 2;
Disc = [0.9756; 0.9426; 0.8799];
EndDates = ['10/15/97'; '04/15/98'; '04/15/99'];
ValuationDate = '4/15/97';
Rates = disc2rate(Compounding, Disc, EndDates, [], ValuationDate)
```

```
Rates = 3×1

    0.0500
    0.0600
    0.0650
```

Input Arguments

`Compounding` — Compounding rate

integer with value of `0`, `1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding rate for which the input zero rates are compounded when annualized, specified as one of the following scalar integers. Compounding determines the formula for the discount factors (`Disc`):

- If `Compounding` = `0` for simple interest:
  - `Disc = 1/(1 + Z * T)`, where `T` is time in years and simple interest assumes annual times `F = 1`.
- If `Compounding` = `1`, `2`, `3`, `4`, `6`, `12`:
  - `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units, for example, `T = F` is one year.
- If `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.
- If `Compounding` = `-1`:
  - `Disc = exp(-T*Z)`, where `T` is time in years.

Data Types: `double`

`Disc` — Discounts

matrix

Discounts, specified as a 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.

Data Types: `double`

EndTimes — End times
serial date number | date character vector | numeric

End times, specified as a scalar or an NPOINTS-by-1 column vector using serial date numbers or date character vectors, or times in periodic units ending the interval to discount over. When EndTimes is not a date, the value for EndTimes is T computed from SIA semi-annual time factors, Tsemi, by the formula T = Tsemi/2 * F, where F is the compounding frequency. F is set to 1 for continuous compounding.

Note When ValuationDate is not passed, EndTimes is interpreted as times. If Compounding = 365 (daily), EndTimes is measured in days.

Data Types: double | char

StartTimes — Start times
serial date number | date character vector | numeric

Start times, specified a scalar or an NPOINTS-by-1 column vector using serial date numbers or date character vectors, or times in periodic units starting the interval to discount over. StartDates must be earlier than EndDates. When StarTimes is not a date, the value for StartTimes is T computed from SIA semi-annual time factors, Tsemi, by the formula T = Tsemi/2 * F, where F is the compounding frequency. F is set to 1 for continuous compounding.

Note When ValuationDate is not passed, StartTimes is interpreted as times. If Compounding = 365 (daily), StartTimes is measured in days.

Data Types: double | char

ValuationDate — Observation date of the investment horizons entered in StartTimes and EndTimes
serial date number | date character vector

Observation date of the investment horizons entered in StartTimes and EndTimes, specified as scalar serial date number or date character vector.

Note You can specify the investment intervals either with input times or with input dates. Entering ValuationDate invokes the date interpretation; omitting ValuationDate invokes the default time interpretations.

Data Types: double | char

Basis — Day-count basis
0 (actual/actual) (default) | integer from 0 to 13

Day-count basis of the instrument when using dates for StartTimes and EndTimes, specified as a scalar or an NINST-by-1 vector of integers..

- 0 = actual/actual
- 1 = 30/360 (SIA)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag when using dates for `StartTimes` and `EndTimes`, specified as a scalar or an `NINST`-by-`1` vector of nonnegative integers. 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

Output Arguments

`Rates` — Rates
vector

Rates, returned as an `NPOINTS`-by-`NCURVES` column vector of in decimal form over `NPOINTS` time intervals.

`EndTimes` — Times ending the interval to discount over
vector

Times ending the interval to discount over, returned as an `NPOINTS`-by-`1` column vector, measured in periodic units.

`StartTimes` — Times starting the interval to discount over
vector

Times starting the interval to discount over, returned as an `NPOINTS`-by-`1` column vector, measured in periodic units.

Version History

Introduced before R2006a

See Also

`rate2disc` | `ratetimes`

Topics

“Modeling the Interest-Rate Term Structure” on page 2-57
“Interest-Rate Term Conversions” on page 2-53
“Interest Rates Versus Discount Factors” on page 2-48
“Understanding the Interest-Rate Term Structure” on page 2-48

eqpprice

Instrument prices from Equal Probabilities binomial tree

Syntax

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

Description

`[Price,PriceTree] = eqpprice(EQPTree,InstSet)` computes stock option prices using an EQP binomial tree created with `eqptree`. All instruments contained in a financial instrument variable, `InstSet`, are priced.

`eqpprice` handles instrument types: `'Asian'`, `'Barrier'`, `'Compound'`, `'CBond'`, `'Lookback'`, `'OptStock'`. See `instadd` to construct defined types.

`[Price,PriceTree] = eqpprice( ___ ,Options)` adds an optional input argument for `Options`.

Examples

Price the Put Options Contained in the Instrument Set

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')
```

```
EQPSubSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {5x1 cell}
     FieldName: {5x1 cell}
    FieldClass: {5x1 cell}
     FieldData: {5x1 cell}
```

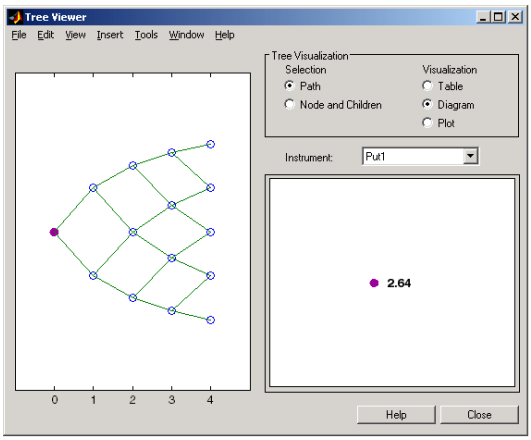
```
instdisp(EQPSubSet)
```

```
Index Type     OptSpec Strike Settle         ExerciseDates  AmericanOpt Name  Quantity
1     OptStock put     105    01-Jan-2003    01-Jan-2006    0           Put1   5

Index Type  OptSpec Strike Settle         ExerciseDates  AmericanOpt AvgType    AvgPrice AvgDate
2     Asian put     110    01-Jan-2003    01-Jan-2006    0           arithmetic NaN      NaN
3     Asian put     110    01-Jan-2003    01-Jan-2007    0           arithmetic NaN      NaN
```

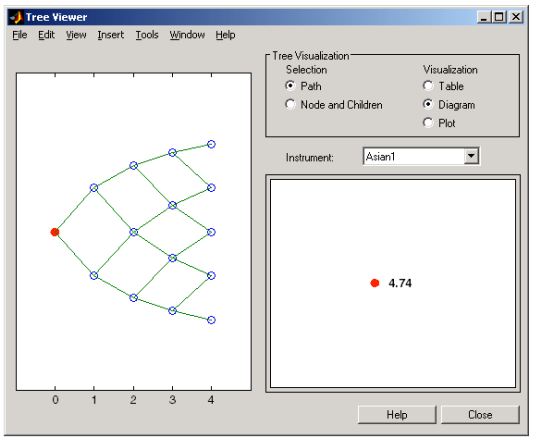
Price the put options.

```
[Price, PriceTree] = eqpprice(EQPTree, EQPSubSet)

Price = 3×1

    2.6375
    4.7444
    3.9178


PriceTree = struct with fields:
    FinObj: 'BinPriceTree'
     PTree: {1x5 cell}
      tObs: [0 1 2 3 4]
      dObs: [731582 731947 732313 732678 733043]
```

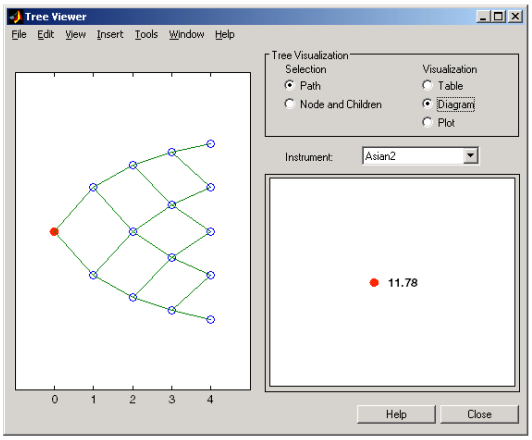
You can use `treeviewer` to see the prices of these three instruments along the price tree.



Put1



Asian11



Asian2

Input Arguments

`EQPTree` — Stock price tree structure

structure

Stock price tree structure, specified by using `eqptree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

(Optional) Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Price for each instrument

vector

Price for each instrument, returned as a `NINST`-by-1 vector. 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.

Related single-type pricing functions are:

- `asianbyeqp`: Price an Asian option from an EQP tree.
- `barrierbyeqp`: Price a barrier option from an EQP tree.
- `cbondbyeqp`: Price convertible bonds 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.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as 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.tObs` contains the observation times.

- `PriceTree.dObs` contains the observation dates.

Version History

Introduced before R2006a

See Also

`eqpsens` | `eqptimespec` | `eqptree` | `instadd` | `instcbond` | `cbondbyeqp`

Topics

“Computing Prices Using EQP” on page 3-66
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

eqpsens

Instrument prices and sensitivities from Equal Probabilities binomial tree

Syntax

```
[Delta,Gamma,Vega,Price] = eqpsens(EQPTree,InstSet)
[Delta,Gamma,Vega,Price] = eqpsens(___,Options)
```

Description

`[Delta,Gamma,Vega,Price] = eqpsens(EQPTree,InstSet)` computes instrument sensitivities and prices for instruments using a binomial tree created with the `eqptree` function. All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide by the respective instrument price.

`eqpsens` handles instrument types: `'Asian'`, `'Barrier'`, `'Compound'`, `'CBond'`, `'Lookback'`, and `'OptStock'`. See `instadd` for information on instrument types.

`[Delta,Gamma,Vega,Price] = eqpsens(___,Options)` adds an optional input argument for `Options`.

Examples

Compute Sensitivities for Instruments Using an eqptree

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')
```

```
EQPSubSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {5x1 cell}
     FieldName: {5x1 cell}
    FieldClass: {5x1 cell}
     FieldData: {5x1 cell}
```

```
instdisp(EQPSubSet)
```

```
Index Type     OptSpec Strike Settle         ExerciseDates  AmericanOpt Name  Quantity
1     OptStock put     105    01-Jan-2003    01-Jan-2006    0           Put1   5

Index Type  OptSpec Strike Settle         ExerciseDates  AmericanOpt AvgType    AvgPrice AvgDate
2     Asian put     110    01-Jan-2003    01-Jan-2006    0           arithmetic NaN      NaN
```

```
3     Asian put     110    01-Jan-2003    01-Jan-2007    0           arithmetic NaN      NaN
```

Obtain the `Delta` and `Gamma` for the put options contained in the instrument set.

```
[Delta, Gamma] = eqpsens(EQPTree, EQPSubSet)
```

```
Delta = 3×1

   -0.2336
   -0.5443
   -0.4516


Gamma = 3×1

    0.0218
    0.0000
    0.0000
```

Input Arguments

`EQPTree` — Stock tree structure

structure

Stock tree structure, specified by using `eqptree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Delta` — Rate of change of instruments prices with respect to changes in the stock price

vector

Rate of change of instruments prices with respect to changes in the stock price, returned as a `NINST`-by-`1` vector of deltas.

For path-dependent options (`'Lookback'` and `'Asian'`), `Delta` and `Gamma` are computed by finite differences in calls to `eqpprice`. For the rest of the options (`'OptStock'`, `'Barrier'`, `'CBond'`,

and 'Compound'), Delta and Gamma are computed from the EQPTree and the corresponding option price tree.

Gamma — Rate of change of instruments deltas with respect to changes in stock price
vector

Rate of change of instruments deltas with respect to changes in the stock price, returned as a NINST-by-1 vector of gammas.

For path-dependent options ('Lookback' and 'Asian'), Delta and Gamma are computed by finite differences in calls to eqpprice. For the rest of the options ('OptStock', 'Barrier', 'CBond', and 'Compound'), Delta and Gamma are computed from the EQPTree and the corresponding option price tree.

Vega — Rate of change of instruments prices with respect to changes in volatility of the stock
vector

Rate of change of instruments prices with respect to changes in the volatility of the stock, returned as a NINST-by-1 vector of vegas. Vega is computed by finite differences in calls to eqptree.

Price — Price of each instrument
vector

Price of each instrument, returned as a NINST-by-1 vector. 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.

Version History

Introduced before R2006a

References

[1] Chriss, Neil. *Black-Scholes and Beyond: Option Pricing Models.* McGraw-Hill, 1996, pp 308-312.

See Also

eqpprice | eqptree | cbondbyeqp | instcbond

Topics

“Computing Prices Using EQP” on page 3-66
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

eqptimespec

Specify time structure for Equal Probabilities binomial tree

Syntax

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

Description

`TimeSpec = eqptimespec(ValuationDate,Maturity,NumPeriods)` sets the number of levels and node times for an equal probabilities tree (`eqptree`).

Examples

Set the Number of Levels and Node Times for an EQP Tree

This example shows how to set the number of levels and node times for an EQP tree by specifying a four-period tree with time steps of 1 year.

```
ValuationDate = datetime(2002,7,1);
Maturity = datetime(2006,7,1);
TimeSpec = eqptimespec(ValuationDate, Maturity, 4)
```

```
TimeSpec = struct with fields:
           FinObj: 'BinTimeSpec'
    ValuationDate: 731398
         Maturity: 732859
       NumPeriods: 4
            Basis: 0
     EndMonthRule: 1
             tObs: [0 1 2 3 4]
             dObs: [731398 731763 732128 732493 732859]
```

Input Arguments

`ValuationDate` — Pricing date and first observation in the tree

datetime scalar | string scalar | date character vector

Pricing date and first observation in the `eqptree`, specified as a scalar datetime, string, or date character vector.

To support existing code, `eqptimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Date marking the depth of the EQP stock tree

datetime scalar | string scalar | date character vector

Date marking the depth of the `eqptree` binomial tree, specified as scalar datetime, string, or date character vector.

To support existing code, `eqptimespec` also accepts serial date numbers as inputs, but they are not recommended.

`NumPeriods` — Number of time steps in the EQP tree
integer

Number of time steps in the `eqptree` binomial tree, specified as scalar integer value.

Data Types: `double`

Output Arguments

`TimeSpec` — Specification for the time layout for `eqptree`
structure

Specification for the time layout for `eqptree`, returned as a structure.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `eqptimespec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`eqptree` | `stockspec`

Topics
“Computing Prices Using EQP” on page 3-66
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

eqptree

Build Equal Probabilities stock tree

Syntax

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

Description

`EQPTree = eqptree(StockSpec,RateSpec,TimeSpec)` builds an Equal Probabilities stock tree.

Examples

Create an EQP Tree

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

```
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);

RateSpec = intenvset('Rates', 0.05, 'StartDates',...
'01-Jan-2003', 'EndDates', '31-Dec-2003');

ValuationDate = '1-Jan-2003';
Maturity = '31-Dec-2003';
TimeSpec = eqptimespec(ValuationDate, Maturity, 4);

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

```
Warning: RateSpec was not created with continuous compounding. Compounding will
be set to continuous while leaving discount factors unaltered. This will result
in the recalculation of the interest rates.

EQPTree =

  struct with fields:

       FinObj: 'BinStockTree'
       Method: 'EQP'
    StockSpec: [1×1 struct]
     TimeSpec: [1×1 struct]
     RateSpec: [1×1 struct]
```

```
         tObs: [0 0.2493 0.4986 0.7479 0.9972]
         dObs: [731582 731673 731764 731855 731946]
        STree: {1×5 cell}
      UpProbs: [0.5000 0.5000 0.5000 0.5000]]
```

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

Input Arguments

`StockSpec` — Stock specification

structure

Stock specification, specified by the `StockSpec` obtained from `stockspec`. See `stockspec` for information on creating a stock specification.

Data Types: `struct`

`RateSpec` — Interest-rate specification for initial risk-free rate curve

structure

Interest-rate specification for initial risk-free rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

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 is not a standard equal probabilities tree.

Data Types: `struct`

`TimeSpec` — Tree time layout specification

structure

Tree time layout specification, specified by the `TimeSpec` obtained from `eqptimespec`. The `TimeSpec` defines the observation dates of the EQP stock tree. See `eqptimespec` for information on the tree structure.

Data Types: `struct`

Output Arguments

`EQPTree` — EQP stock tree

structure

EQP stock tree, returned as a structure specifying the time layout for the tree.

Version History

Introduced before R2006a

See Also

`eqptimespec` | `intenvset` | `stockspec`

Topics

“Computing Prices Using EQP” on page 3-66
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

fixedbybdt

Price fixed-rate note from Black-Derman-Toy interest-rate tree

Syntax

```
[Price,PriceTree] = fixedbybdt(BDTTree,CouponRate,Settle,Maturity)
[Price,PriceTree] = fixedbybdt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = fixedbybdt(BDTTree,CouponRate,Settle,Maturity)` prices a fixed-rate note from a Black-Derman-Toy interest-rate tree.

Note Alternatively, you can use the `FixedBond` object to price fixed-rate bond instruments. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`[Price,PriceTree] = fixedbybdt( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a 10% Fixed-Rate Note Using a BDT Interest-Rate Tree

This example shows how to price a 10% fixed-rate note using a BDT interest-rate tree by loading 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

CouponRate = 0.10;
Settle = datetime(2000,1,1);
Maturity = datetime(2004,1,1);
FixedReset = 1;

Price = fixedbybdt(BDTTree, CouponRate, Settle, Maturity, FixedReset)
```

```
Price = 92.9974
```

Input Arguments

BDTTree — Interest-rate structure

structure

Interest-rate tree structure, created by `bdttree`

Data Types: `struct`

`CouponRate` — Coupon annual rate
decimal

Coupon annual rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `fixedbybdt` also accepts serial date numbers as inputs, but they are not recommended.

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.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `fixedbybdt` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = fixedbybdt(BDTTree,CouponRate,Settle,Maturity,'FixedReset',4)`

`FixedReset` — Frequency of payments per year
`1` (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of `'FixedReset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

Output Arguments

`Price` — Expected fixed-rate note prices at time 0
vector

Expected fixed-rate note prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.

More About

Fixed-Rate Note

A fixed-rate note is a long-term debt security with a preset interest rate and maturity, by which the interest must be paid.

The principal may or may not be paid at maturity. In Financial Instruments Toolbox, the principal is always paid at maturity. For more information, see “Fixed-Rate Note” on page 2-9.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `fixedbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdttree` | `bondbybdt` | `capbybdt` | `cfbybdt` | `floatbybdt` | `floorbybdt` | `swapbybdt` | `FixedBond`

Topics

“Computing Instrument Prices” on page 2-81
“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond Portfolio Optimization Using Portfolio Object”
“Fixed-Rate Note” on page 2-9
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

fixedbybk

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

Syntax

```
[Price,PriceTree] = fixedbybk(BKTree,CouponRate,Settle,Maturity)
[Price,PriceTree] = fixedbybk(___,Name,Value)
```

Description

`[Price,PriceTree] = fixedbybk(BKTree,CouponRate,Settle,Maturity)` prices a fixed-rate note from a Black-Karasinski interest-rate tree.

Note Alternatively, you can use the `FixedBond` object to price fixed-rate bond instruments. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`[Price,PriceTree] = fixedbybk(___,Name,Value)` adds additional name-value pair arguments.

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 = datetime(2004,1,1);
Maturity = datetime(2006,1,1);
```

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

```
Price = fixedbybk(BKTree, CouponRate, Settle, Maturity)
```

```
Price = 103.5126
```

Input Arguments

BKTree — Interest-rate structure

structure

Interest-rate tree structure, created by `bktree`

Data Types: `struct`

`CouponRate` — Coupon annual rate
decimal

Coupon annual rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `fixedbybk` also accepts serial date numbers as inputs, but they are not recommended.

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.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each fixed-rate note.

To support existing code, `fixedbybk` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = fixedbybk(BKTree,CouponRate,Settle,Maturity,'FixedReset',4)`

`FixedReset` — Frequency of payments per year
`1` (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of `'FixedReset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

- 0 = actual/actual

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

AdjustCashFlowsBasis — Flag to adjust cash flows based on actual period day count
false (default) | value of 0 (false) or 1 (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of 'AdjustCashFlowsBasis' and a NINST-by-1 vector of logicals with values of 0 (false) or 1 (true).

Data Types: logical

Holidays — Holidays used in computing business days
if not specified, the default is to use holidays.m (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of 'Holidays' and MATLAB dates using a NHolidays-by-1 vector.

Data Types: datetime

BusinessDayConvention — Business day conventions
actual (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of 'BusinessDayConvention' and a character vector or a N-by-1 cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- actual — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- follow — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- modifiedfollow — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- previous — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- modifiedprevious — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: char | cell

Output Arguments

Price — Expected fixed-rate note prices at time 0
vector

Expected fixed-rate note prices at time 0, returned as a NINST-by-1 vector.

PriceTree — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

Fixed-Rate Note

A fixed-rate note is a long-term debt security with a preset interest rate and maturity, by which the interest must be paid.

The principal may or may not be paid at maturity. In Financial Instruments Toolbox, the principal is always paid at maturity. For more information, see “Fixed-Rate Note” on page 2-9.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `fixedbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bktree` | `bondbybk` | `capbybk` | `cfbybk` | `floatbybk` | `floorbybk` | `swapbybk` | `FixedBond`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond Portfolio Optimization Using Portfolio Object”
“Fixed-Rate Note” on page 2-9
“Understanding Interest-Rate Tree Models” on page 2-66

“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

fixedbycir

Price fixed rate note from Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = fixedbycir(CIRTree,CouponRate,Settle,Maturity)
[Price,PriceTree] = fixedbycir( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = fixedbycir(CIRTree,CouponRate,Settle,Maturity)` prices a fixed-rate note from a Cox-Ingersoll-Ross (CIR) interest-rate tree using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = fixedbycir( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Fixed-Rate Note Using a CIR Interest-Rate Tree

Define the `CouponRate` for a fixed-rate note.

```
CouponRate = 0.03;
```

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = datetime(2017,1,1);
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',EndD
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = datetime(2017,1,1);
Maturity = datetime(2021,1,1);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
```

```
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [736696 737061 737426 737791]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Price the 3% fixed-rate note.

```
[Price,PriceTree] = fixedbycir(CIRT,CouponRate,Settle,Maturity)
```

```
Price = 92.1422
```

```
PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
       tObs: [0 1 2 3 4]
       dObs: [736696 737061 737426 737791 738157]
      PTree: {1x5 cell}
     AITree: {[0]  [0 0 0]  [0 0 0 0 0]  [0 0 0 0 0 0 0]  [0 0 0 0 0 0 0]}
    Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Input Arguments

`CIRTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `cirtree`

Data Types: `struct`

`CouponRate` — Coupon annual rate

decimal

Coupon annual rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `fixedbycir` also accepts serial date numbers as inputs, but they are not recommended.

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

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each fixed-rate note.

To support existing code, `fixedbycir` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = fixedbycir(CIRTree,CouponRate,Settle,Maturity,'FixedReset',4)`

`FixedReset` — Frequency of payments per year

`1` (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of `'FixedReset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day count basis

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules

`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` cell array of character vectors of business day conventions. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.

- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

Output Arguments

`Price` — Expected fixed-rate note prices at time 0

vector

Expected fixed-rate note prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as 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.tObs` contains the observation times.
- `PriceTree.dObs` contains the observation dates.
- `PriceTree.PTree` contains the clean prices.
- `PriceTree.AITree` contains the accrued interest.

More About

Fixed-Rate Note

A fixed-rate note is a long-term debt security with a preset interest rate and maturity, by which the interest must be paid.

The principal may or may not be paid at maturity. In Financial Instruments Toolbox, the principal is always paid at maturity. For more information, see “Fixed-Rate Note” on page 2-9.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `fixedbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J.,and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir` | `instfixed`

Topics

“Computing Instrument Prices” on page 2-81
“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Fixed-Rate Note” on page 2-9
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

fixedbyhjm

Price fixed-rate note from Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree] = fixedbyhjm(HJMTree,CouponRate,Settle,Maturity)
[Price,PriceTree] = fixedbyhjm( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = fixedbyhjm(HJMTree,CouponRate,Settle,Maturity)` prices a fixed-rate note from a Heath-Jarrow-Morton interest-rate tree.

`[Price,PriceTree] = fixedbyhjm( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a 4% Fixed-Rate Note Using an HJM Forward-Rate Tree

This example shows how to price a 4% fixed-rate note using an HJM forward-rate tree by loading 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

CouponRate = 0.04;
Settle = datetime(2000,1,1);
Maturity = datetime(2003,1,1);

Price = fixedbyhjm(HJMTree, CouponRate, Settle, Maturity)
```

```
Price = 98.7159
```

Input Arguments

`HJMTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `hjmtree`

Data Types: `struct`

`CouponRate` — Coupon annual rate

decimal

Coupon annual rate, specified as a `NINST`-by-1 vector.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `fixedbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

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.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each fixed-rate note.

To support existing code, `fixedbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = fixedbyhjm(HJMTree,CouponRate,Settle,Maturity,'FixedReset',4)`

`FixedReset` — Frequency of payments per year
`1` (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of `'FixedReset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

Output Arguments

`Price` — Expected fixed-rate note prices at time 0
vector

Expected fixed-rate note prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.PBush` contains the clean prices.
- `PriceTree.AITree` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.

More About

Fixed-Rate Note

A fixed-rate note is a long-term debt security with a preset interest rate and maturity, by which the interest must be paid.

The principal may or may not be paid at maturity. In Financial Instruments Toolbox, the principal is always paid at maturity. For more information, see “Fixed-Rate Note” on page 2-9.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `fixedbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bondbyhjm` | `capbyhjm` | `cfbyhjm` | `floatbyhjm` | `floorbyhjm` | `hjmtree` | `swapbyhjm`

Topics

“Computing Instrument Prices” on page 2-81
“Fixed-Rate Note” on page 2-9
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

fixedbyhw

Price fixed-rate note from Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = fixedbyhw(HWTree,CouponRate,Settle,Maturity)
[Price,PriceTree] = fixedbyhw(___,Name,Value)
```

Description

`[Price,PriceTree] = fixedbyhw(HWTree,CouponRate,Settle,Maturity)` prices a fixed-rate note from a Hull-White interest-rate tree.

Note Alternatively, you can use the `FixedBond` object to price fixed-rate bond instruments. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`[Price,PriceTree] = fixedbyhw(___,Name,Value)` adds additional name-value pair arguments.

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 = datetime(2004,1,1);
Maturity = datetime(2006,1,1);
```

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

```
Price = fixedbyhw(HWTree, CouponRate, Settle, Maturity)
```

```
Price = 103.5126
```

Input Arguments

HWTree — Interest-rate structure

structure

Interest-rate tree structure, created by `hwtree`

Data Types: `struct`

`CouponRate` — Coupon annual rate
decimal

Coupon annual rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `fixedbyhw` also accepts serial date numbers as inputs, but they are not recommended.

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.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each fixed-rate note.

To support existing code, `fixedbyhw` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = fixedbyhw(HWTree,CouponRate,Settle,Maturity,'FixedReset',4)`

`FixedReset` — Frequency of payments per year
`1` (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of `'FixedReset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

- 0 = actual/actual

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

Output Arguments

`Price` — Expected fixed-rate note prices at time 0
vector

Expected fixed-rate note prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

Fixed-Rate Note

A fixed-rate note is a long-term debt security with a preset interest rate and maturity, by which the interest must be paid.

The principal may or may not be paid at maturity. In Financial Instruments Toolbox, the principal is always paid at maturity. For more information, see “Fixed-Rate Note” on page 2-9.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `fixedbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bondbyhw` | `capbyhw` | `cfbyhw` | `floatbyhw` | `floorbyhw` | `hwtree` | `swapbyhw` | `FixedBond`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond Portfolio Optimization Using Portfolio Object”
“Fixed-Rate Note” on page 2-9

“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

fixedbyzero

Price fixed-rate note from set of zero curves

Syntax

```
[Price,DirtyPrice,CFlowAmounts,CFlowDates] = fixedbyzero(RateSpec,CouponRate,
Settle,Maturity)
[Price,DirtyPrice,CFlowAmounts,CFlowDates] = fixedbyzero( ___ ,Name,Value)
```

Description

`[Price,DirtyPrice,CFlowAmounts,CFlowDates] = fixedbyzero(RateSpec,CouponRate, Settle,Maturity)` prices a fixed-rate note from a set of zero curves.

Note Alternatively, you can use the `FixedBond` object to price fixed-rate bond instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,DirtyPrice,CFlowAmounts,CFlowDates] = fixedbyzero( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a 4% Fixed-Rate Note Using a Set of Zero Curves

This example shows how to price a 4% fixed-rate note using a set of zero curves by loading the file `deriv.mat`, which provides `ZeroRateSpec`, the interest-rate term structure needed to price the note.

```
load deriv.mat

CouponRate = 0.04;
Settle = datetime(2000,1,1);
Maturity = datetime(2003,1,1);

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

```
Price = 98.7159
```

Pricing a Fixed-Fixed Cross Currency Swap

Assume that a financial institution has an existing swap with three years left to maturity where they are receiving 5% per year in yen and paying 8% per year in USD. The reset frequency for the swap is annual, the principals for the two legs are 1200 million yen and $10 million USD, and both term structures are flat.

```
Settle = datetime(2015,8,15);
Maturity = datetime(2018,8,15);
Reset = 1;

r_d = .09;
r_f = .04;

FixedRate_d = .08;
FixedRate_f = .05;

Principal_d = 10000000;
Principal_f = 1200000000;

S0 = 1/110;
```

Construct term structures.

```
RateSpec_d = intenvset('StartDate',Settle,'EndDate',Maturity,'Rates',r_d,'Compounding',-1);
RateSpec_f = intenvset('StartDate',Settle,'EndDate',Maturity,'Rates',r_f,'Compounding',-1);
```

Use fixedbyzero:

```
B_d = fixedbyzero(RateSpec_d,FixedRate_d,Settle,Maturity,'Principal',Principal_d,'Reset',Reset);
B_f = fixedbyzero(RateSpec_f,FixedRate_f,Settle,Maturity,'Principal',Principal_f,'Reset',Reset);
```

Compute swap price. Based on Hull (see References), a cross currency swap can be valued with the following formula `V_swap = S0*B_f − B_d`.

```
V_swap = S0*B_f - B_d
```

```
V_swap = 1.5430e+06
```

Input Arguments

`RateSpec` — Annualized zero rate term structure

structure

Annualized zero rate term structure, specified using `intenvset` to create a `RateSpec`.

Data Types: `struct`

`CouponRate` — Annual rate

decimal

Annual rate, specified as `NINST`-by-`1` decimal annual rate or a `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array and the first column is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `fixedbyzero` also accepts serial date numbers as inputs, but they are not recommended.

`Settle` must be earlier than `Maturity`.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each fixed-rate note.

To support existing code, `fixedbyzero` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,DirtyPrice,CFlowAmounts,CFlowDates] = fixedbyzero(RateSpec,CouponRate,Settle,Maturity,'Principal',Principal)`

`FixedReset` — Frequency of payments per year
`1` (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of `'FixedReset'` and a `NINST`-by-1 vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-1 vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)

- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

Output Arguments

`Price` — Fixed-rate note prices
matrix

Floating-rate note prices, returned as a (`NINST`) by number of curves (`NUMCURVES`) matrix. Each column arises from one of the zero curves.

`DirtyPrice` — Dirty bond price
matrix

Dirty bond price (clean + accrued interest), returned as a `NINST`- by-`NUMCURVES` matrix. Each column arises from one of the zero curves.

`CFlowAmounts` — Cash flow amounts
matrix

Cash flow amounts, returned as a `NINST`- by-`NUMCFS` matrix of cash flows for each bond.

`CFlowDates` — Cash flow dates
matrix

Cash flow dates, returned as a `NINST`- by-`NUMCFS` matrix of payment dates for each bond.

More About

Fixed-Rate Note

A fixed-rate note is a long-term debt security with a preset interest rate and maturity, by which the interest must be paid.

The principal may or may not be paid at maturity. In Financial Instruments Toolbox, the principal is always paid at maturity. For more information, see “Fixed-Rate Note” on page 2-9.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `fixedbyzero` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. *Options, Futures, and Other Derivatives.* Prentice-Hall, 2011.

See Also

`bondbyzero` | `cfbyzero` | `floatbyzero` | `swapbyzero` | `FixedBond`

Topics

“Pricing Using Interest-Rate Term Structure” on page 2-61
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond Portfolio Optimization Using Portfolio Object”
“Fixed-Rate Note” on page 2-9
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

floatbybdt

Price floating-rate note from Black-Derman-Toy interest-rate tree

Syntax

```
[Price,PriceTree] = floatbybdt(BDTTree,Spread,Settle,Maturity)
[Price,PriceTree] = floatbybdt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = floatbybdt(BDTTree,Spread,Settle,Maturity)` prices a floating-rate note from a Black-Derman-Toy interest-rate tree.

`floatbybdt` computes prices of vanilla floating-rate notes, amortizing floating-rate notes, capped floating-rate notes, floored floating-rate notes and collared floating-rate notes.

Note Alternatively, you can use the `FloatBond` object to price floating-rate bond instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = floatbybdt( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Floating-Rate Note Using a BDT Tree

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;
```

Define the floating-rate note using the required arguments. Other arguments use defaults.

```
Spread = 20;
Settle = datetime(2000,1,1);
Maturity = datetime(2003,1,1);
```

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

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

```
Price = 100.4865
```

Price an Amortizing Floating-Rate Note

Price an amortizing floating-rate note using the `Principal` input argument to define the amortization schedule.

Create the `RateSpec`.

```
Rates = [0.03583; 0.042147; 0.047345; 0.052707; 0.054302];
ValuationDate = datetime(2011,11,15);
StartDates = ValuationDate;
EndDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate,'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
          EndTimes: [5x1 double]
        StartTimes: [5x1 double]
          EndDates: [5x1 double]
        StartDates: 734822
    ValuationDate: 734822
            Basis: 0
     EndMonthRule: 1
```

Create the floating-rate instrument using the following data:

```
Settle = datetime(2011,11,15);
Maturity = datetime(2015,11,15);
Spread = 15;
```

Define the floating-rate note amortizing schedule.

```
Principal ={{datetime(2012,11,15) 100;datetime(2013,11,15) 70;datetime(2014,11,15) 40;datetime(20
```

Build the BDT tree and assume volatility is 10%.

```
MatDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
BDTTimeSpec = bdttimespec(ValuationDate, MatDates);
Volatility = 0.10;
BDTVolSpec = bdtvolspec(ValuationDate, MatDates, Volatility*ones(1,length(MatDates))');
BDTT = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Compute the price of the amortizing floating-rate note.

```
Price = floatbybdt(BDTT, Spread, Settle, Maturity, 'Principal', Principal)
```

```
Price = 100.3059
```

Price a Collar with a Floating-Rate Note

Price a collar with a floating-rate note using the `CapRate` and `FloorRate` input argument to define the collar pricing.

Create the `RateSpec`.

```
Rates = [0.0287; 0.03024; 0.03345; 0.03861; 0.04033];
ValuationDate = datetime(2012,4,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,4,1) ; datetime(2014,4,1) ; datetime(2015,4,1) ; datetime(2016,4,1) ; d
Compounding = 1;
```

Create the `RateSpec`.

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

Build the BDT tree and assume volatility is 5%.

```
MatDates = [datetime(2013,4,1) ; datetime(2014,4,1) ; datetime(2015,4,1) ; datetime(2016,4,1) ; d
BDTTimeSpec = bdttimespec(ValuationDate, MatDates);
Volatility = 0.05;
BDTVolSpec = bdtvolspec(ValuationDate, MatDates, Volatility*ones(1,length(MatDates))');
BDTT = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Create the floating rate note instrument.

```
Settle = datetime(2012,4,1);
Maturity = datetime(2016,4,1);
Spread = 10;
Principal = 100;
```

Compute the price of a collared floating-rate note.

```
CapStrike = {{datetime(2013,4,1) 0.03;datetime(2015,4,1) 0.055}};
FloorStrike = {{datetime(2013,4,1) 0.025;datetime(2015,4,1) 0.04}};

Price = floatbybdt(BDTT, Spread, Settle, Maturity, 'CapRate',...
CapStrike, 'FloorRate', FloorStrike)
```

```
Price = 101.2414
```

Pricing a Floating-Rate Note When the Reset Dates Are Not Tree Level Dates

When using `floatbybdt` to price floating-rate notes, there are cases where the dates specified in the BDT tree `TimeSpec` are not aligned with the cash flow dates.

Price floating-rate notes using the following data:

```
ValuationDate = datetime(2013,9,1);
Rates = [0.0235; 0.0239; 0.0311; 0.0323];
EndDates = [datetime(2014,9,1);datetime(2015,9,1);datetime(2016,9,1);datetime(2017,9,1)];
```

Create the `RateSpec`.

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

Build the BDT tree.

```
VolCurve = [.10; .11; .11; .12];

BDTVolatilitySpec = bdtvolspec(RateSpec.ValuationDate, EndDates,...
                               VolCurve);

BDTTimeSpec = bdttimespec(RateSpec.ValuationDate, EndDates, 1);

BDTT = bdttree(BDTVolatilitySpec, RateSpec, BDTTimeSpec);
```

Compute the price of the floating-rate note using the following data:

```
Spread = 5;
Settle = datetime(2013,9,1);
Maturity = datetime(2013,12,1);
Reset = 2;

Price = floatbybdt(BDTT, Spread, Settle, Maturity, 'FloatReset', Reset)

Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In floatengbybdt at 204
  In floatbybdt at 123
Error using floatengbybdt (line 299)
Instrument '1 ' has cash flow dates that span across tree nodes.

Error in floatbybdt (line 123)
[Price, PriceTree, CFTree, TLPpal] = floatengbybdt(BDTTree, Spread, Settle, Maturity, OArgs{:});
```

This error indicates that it is not possible to determine the applicable rate used to calculate the payoff at the reset dates, given that the applicable rate needed cannot be calculated (the information was lost due to the recombination of the tree nodes). Note, if the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates. The simplest solution is to place the tree levels at the cash flow dates of the instrument, which is done by specifying `BDTTimeSpec`. It is also acceptable to have reset dates between tree levels, as long as there are reset dates on the tree levels.

To recover from this error, build a tree that lines up with the instrument.

```
Basis = intenvget(RateSpec, 'Basis');
EOM = intenvget(RateSpec, 'EndMonthRule');
resetDates = cfdates(ValuationDate, Maturity, Reset ,Basis, EOM);
BDTTimeSpec = bdttimespec(RateSpec.ValuationDate,resetDates,Reset);
BDTT = bdttree(BDTVolatilitySpec, RateSpec, BDTTimeSpec);

Price = floatbybdt(BDTT, Spread, RateSpec.ValuationDate, ...
                   Maturity, 'FloatReset', Reset)

Price =

  100.1087
```

Input Arguments

`BDTTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `bdttree`

Data Types: `struct`

Spread — Number of basis points over the reference rate
vector

Number of basis points over the reference rate, specified as a NINST-by-1 vector.

Data Types: double

Settle — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, floatbybdt also accepts serial date numbers as inputs, but they are not recommended.

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.

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each floating-rate note.

To support existing code, floatbybdt also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree] = floatbybdt(BDTTree,Spread,Settle,Maturity,'Basis',3)

FloatReset — Frequency of payments per year
1 (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of 'FloatReset' and a NINST-by-1 vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates.

Data Types: double

Basis — Day count basis
0 (actual/actual) (default) | integer from 0 to 13

Day count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer `[0, 1]` using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.

- 1 = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: logical

AdjustCashFlowsBasis — Flag to adjust cash flows based on actual period day count
false (default) | value of 0 (false) or 1 (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of 'AdjustCashFlowsBasis' and a NINST-by-1 vector of logicals with values of 0 (false) or 1 (true).

Data Types: logical

Holidays — Holidays used in computing business days
if not specified, the default is to use holidays.m (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of 'Holidays' and MATLAB dates using a NHolidays-by-1 vector.

Data Types: datetime

BusinessDayConvention — Business day conventions
actual (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of 'BusinessDayConvention' and a character vector or a N-by-1 cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- actual — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- follow — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- modifiedfollow — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- previous — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- modifiedprevious — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: char | cell

CapRate — Annual cap rate
decimal

Annual cap rate, specified as the comma-separated pair consisting of 'CapRate' and a NINST-by-1 decimal annual rate or NINST-by-1 cell array, where each element is a NumDates-by-2 cell array, and the cell array first column is dates, and the second column is associated cap rates. The date indicates the last day that the cap rate is valid.

Data Types: double | cell

`FloorRate` — Annual floor rate
decimal

Annual floor rate, specified as the comma-separated pair consisting of `'FloorRate'` and a `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array, and the cell array first column is dates, and the second column is associated floor rates. The date indicates the last day that the floor rate is valid.

Data Types: `double` | `cell`

Output Arguments

`Price` — Expected floating-rate note prices at time 0
vector

Expected floating-rate note prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.

More About

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.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `floatbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdttree` | `bondbybdt` | `capbybdt` | `cfbybdt` | `fixedbybdt` | `floorbybdt` | `swapbybdt` | `FloatBond`

Topics

“Computing Instrument Prices” on page 2-81
“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Compute LIBOR Fallback” on page 2-192
“Floating-Rate Note” on page 2-10
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

floatbybk

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

Syntax

```
[Price,PriceTree] = floatbybk(BKTree,Spread,Settle,Maturity)
[Price,PriceTree] = floatbybk(___,Name,Value)
```

Description

`[Price,PriceTree] = floatbybk(BKTree,Spread,Settle,Maturity)` prices a floating-rate note from a Black-Karasinski interest-rate tree.

`floatbybk` computes prices of vanilla floating-rate notes, amortizing floating-rate notes, capped floating-rate notes, floored floating-rate notes and collared floating-rate notes.

Note Alternatively, you can use the `FloatBond` object to price floating-rate bond instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = floatbybk(___,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Floating-Rate Note Using a Black-Karasinski Tree

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;
```

Define the floating-rate note using the required arguments. Other arguments use defaults.

```
Spread = 20;
Settle = datetime(2005,1,1);
Maturity = datetime(2006,1,1);
```

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

```
Price = floatbybk(BKTree, Spread, Settle, Maturity)
```

```
Warning: Floating range notes are valued at Tree ValuationDate rather than Settle.
```

```
Price = 100.3825
```

Price an Amortizing Floating-Rate Note

Price an amortizing floating-rate note using the `Principal` input argument to define the amortization schedule.

Create the `RateSpec`.

```
Rates = [0.03583; 0.042147; 0.047345; 0.052707; 0.054302];
ValuationDate = datetime(2011,11,15);
StartDates = ValuationDate;
EndDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate,'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: [5x1 double]
       StartDates: 734822
    ValuationDate: 734822
            Basis: 0
     EndMonthRule: 1
```

Create the floating-rate instrument using the following data:

```
Settle = datetime(2011,11,15);
Maturity = datetime(2015,11,15);
Spread = 15;
```

Define the floating-rate note amortizing schedule.

```
Principal ={{datetime(2012,11,15) 100;datetime(2013,11,15) 70;datetime(2014,11,15) 40;datetime(20
```

Build the BK tree and assume the volatility is 10%.

```
VolDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
VolCurve = 0.1;
AlphaDates = datetime(2017,11,15);
AlphaCurve = 0.1;

BKVolSpec = bkvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RateSpec.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RateSpec, BKTimeSpec);
```

Compute the price of the amortizing floating-rate note.

```
Price = floatbybk(BKT, Spread, Settle, Maturity, 'Principal', Principal)
```

```
Price = 100.3059
```

Price a Collar with a Floating-Rate Note

Price a collar with a floating-rate note using the `CapRate` and `FloorRate` input argument to define the collar pricing.

Price a portfolio of collared floating-rate notes using the following data:

```
Rates = [0.0287; 0.03024; 0.03345; 0.03861; 0.04033];
ValuationDate = datetime(2012,4,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,4,1) ; datetime(2014,4,1) ; datetime(2015,4,1) ; datetime(2016,4,1) ; d
Compounding = 1;
```

Create the `RateSpec`.

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

Build the BK tree and assume the volatility to be 5%.

```
VolDates = [datetime(2013,4,1) ; datetime(2014,4,1) ; datetime(2015,4,1) ; datetime(2016,4,1) ; d
VolCurve = 0.05;
AlphaDates = datetime(2018,11,15);
AlphaCurve = 0.1;

BKVolSpec = bkvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RateSpec.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RateSpec, BKTimeSpec);
```

Create the floating-rate note instrument.

```
Settle = datetime(2012,4,1);
Maturity = datetime(2016,4,1);
Spread = [15;10];
Principal = 100;
```

Compute the price of the two vanilla floaters.

```
Price = floatbybk(BKT, Spread, Settle, Maturity)
```

```
Price = 2×1

  100.5519
  100.3680
```

Compute the price of the collared floating-rate notes.

```
CapStrike = {{datetime(2013,4,1) 0.045;datetime(2014,4,1) 0.05;...
datetime(2015,4,1) 0.06}; 0.06};

FloorStrike = {{datetime(2013,4,1) 0.035; datetime(2014,4,1) 0.04;...
datetime(2015,4,1) 0.05}; 0.03};
PriceCollared = floatbybk(BKT, Spread, Settle, Maturity,...
'CapRate', CapStrike,'FloorRate', FloorStrike)
```

```
PriceCollared = 2×1
```

```
  102.8537
  100.4918
```

Pricing a Floating-Rate Note When the Reset Dates Are Not Tree Level Dates

When using `floatbybk` to price floating-rate notes, there are cases where the dates specified in the BK tree Time Specs are not aligned with the cash flow dates.

Price floating-rate notes using the following data:

```
ValuationDate      = datetime(2013,9,13);
ForwardRatesVector = [ 0.0001; 0.0001; 0.0010; 0.0015];
EndDatesVector     = [datetime(2013,12,13);datetime(2014,3,14);datetime(2014,6,13);datetime(2014,9,13)];
```

Create the `RateSpec`.

```
RateSpec = intenvset('ValuationDate',ValuationDate,'StartDates',...
ValuationDate,'EndDates',EndDatesVector,'Rates',ForwardRatesVector,'Compounding', 1);
```

Build the BK tree.

```
Volcurve         = 0.1;
Alpha            = 0.01;
BKVolatilitySpec = bkvolspec(RateSpec.ValuationDate, ...
                   EndDatesVector, Volcurve,...
                   EndDatesVector, Alpha);

BKTimeSpec = bktimespec(RateSpec.ValuationDate, EndDatesVector, 1);

BKT = bktree(BKVolatilitySpec, RateSpec, BKTimeSpec);
```

Create the floating-rate note instrument using the following data;

```
Spread      = 0;
Maturity    = datetime(2014,6,13);
reset = 4;
```

Compute the price of the floating-rate note.

```
Price = floatbybk(BKT, Spread, RateSpec.ValuationDate,...
Maturity, 'FloatReset', reset)

Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In floatengbytrintree at 214
  In floatbybk at 136
Error using floatengbytrintree (line 319)
Instrument '1 ' has cash flow dates that span across tree nodes.

Error in floatbybk (line 136)
[Price, PriceTree, CFTree] = floatengbytrintree(BKTree, Spread, Settle, Maturity, OArgs{:});
```

This error indicates that it is not possible to determine the applicable rate used to calculate the payoff at the reset dates, given that the applicable rate needed cannot be calculated (the information was lost due to the recombination of the tree nodes). Note, if the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates. The simplest solution is to place the tree levels at the cash flow dates of the instrument, which is done by specifying `BKTimeSpec`. It is also acceptable to have reset dates between tree levels, as long as there are reset dates on the tree levels.

To recover from this error, build a tree that lines up with the instrument.

```
Basis = intenvget(RateSpec, 'Basis');
EOM = intenvget(RateSpec, 'EndMonthRule');
resetDates = cfdates(ValuationDate, Maturity,reset,Basis,EOM);
BKTimeSpec = bktimespec(RateSpec.ValuationDate,resetDates,reset);
BKT        = bktree(BKVolatilitySpec, RateSpec, BKTimeSpec);

Price      = floatbybk(BKT, Spread, RateSpec.ValuationDate, ...
             Maturity, 'FloatReset', reset)
```

```
Price =

  100.0004
```

Input Arguments

`BKTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `bktree`

Data Types: `struct`

`Spread` — Number of basis points over the reference rate

vector

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floatbybk` also accepts serial date numbers as inputs, but they are not recommended.

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.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each floating-rate note.

To support existing code, `floatbybk` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = floatbybk(BKTree,Spread,Settle,Maturity,'Basis',3)`

FloatReset — Frequency of payments per year
1 (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of 'FloatReset' and a NINST-by-1 vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates.

Data Types: double

Basis — Day count basis
0 (actual/actual) (default) | integer from 0 to 13

Day count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of 'Basis' and a NINST-by-1 vector.

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

For more information, see “Basis” on page 2-228.

Data Types: double

Principal — Notional principal amounts or principal value schedules
100 (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of 'Principal' and a vector or cell array.

Principal accepts a NINST-by-1 vector or NINST-by-1 cell array, where each element of the cell array is a NumDates-by-2 cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.

- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`CapRate` — Annual cap rate
decimal

Annual cap rate, specified as the comma-separated pair consisting of `'CapRate'` and a `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array, and the cell array first column is dates, and the second column is associated cap rates. The date indicates the last day that the cap rate is valid.

Data Types: `double` | `cell`

`FloorRate` — Annual floor rate
decimal

Annual floor rate, specified as the comma-separated pair consisting of `'FloorRate'` and a `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array, and the cell array first column is dates, and the second column is associated floor rates. The date indicates the last day that the floor rate is valid.

Data Types: `double` | `cell`

Output Arguments

`Price` — Expected floating-rate note prices at time 0
vector

Expected floating-rate note prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

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.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floatbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bktree` | `bondbybk` | `capbybk` | `cfbybk` | `fixedbybk` | `floorbybk` | `swapbybk` | `FloatBond`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Compute LIBOR Fallback” on page 2-192
“Floating-Rate Note” on page 2-10
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

floatbycir

Price floating-rate note from Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = floatbycir(CIRTree,Spread,Settle,Maturity)
[Price,PriceTree] = floatbycir( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = floatbycir(CIRTree,Spread,Settle,Maturity)` prices a floating-rate note from a Cox-Ingersoll-Ross (CIR) interest-rate tree.

`floatbycir` computes prices of vanilla floating-rate notes, amortizing floating-rate notes, capped floating-rate notes, floored floating-rate notes, and collared floating-rate notes using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = floatbycir( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Floating-Rate Note Using a CIR Interest-Rate Tree

Define a `Spread` of 20-basis points for a floating-rate note.

```
Spread = 20;
```

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = datetime(2017,1,1);
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',End
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = datetime(2017,1,1);
Maturity = datetime(2021,1,1);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
```

```
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [736696 737061 737426 737791]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Price the 20-basis point floating-rate note.

```
[Price,PriceTree] = floatbycir(CIRT,Spread,Settle,Maturity)
```

```
Price = 100.7143
```

```
PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
      PTree: {1x5 cell}
     AITree: {[0]  [0 0 0]  [0 0 0 0 0]  [0 0 0 0 0 0 0]  [0 0 0 0 0 0 0]}
       tObs: [0 1 2 3 4]
    Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
      Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree structure, created by `cirtree`

Data Types: `struct`

`Spread` — Number of basis points over the reference rate

vector

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floatbycir` also accepts serial date numbers as inputs, but they are not recommended.

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

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each floating-rate note.

To support existing code, floatbycir also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree] = floatbycir(CIRTree,Spread,Settle,Maturity,'Basis',3)

FloatReset — Frequency of payments per year

1 (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of 'FloatReset' and a NINST-by-1 vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates.

Data Types: double

Basis — Day count basis

0 (actual/actual) (default) | integer from 0 to 13

Day count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of 'Basis' and a NINST-by-1 vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)

- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array, and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` cell array of character vectors of business day conventions. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`CapRate` — Annual cap rate
decimal

Annual cap rate, specified as the comma-separated pair consisting of `'CapRate'` and a `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array, and the cell array first column is dates, and the second column is associated cap rates. The date indicates the last day that the cap rate is valid.

Data Types: `double` | `cell`

`FloorRate` — Annual floor rate
decimal

Annual floor rate, specified as the comma-separated pair consisting of `'FloorRate'` and a `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array.

For the `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array, where the cell array first column is dates, and the second column is associated floor rates. The date indicates the last day that the floor rate is valid.

Data Types: `double` | `cell`

Output Arguments

`Price` — Expected floating-rate note prices at time 0
vector

Expected floating-rate note prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

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.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floatbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J.,and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir` | `instfloat`

Topics

“Computing Instrument Prices” on page 2-81
“Floating-Rate Note” on page 2-10
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

floatbyhjm

Price floating-rate note from Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree] = floatbyhjm(HJMTree,Spread,Settle,Maturity)
[Price,PriceTree] = floatbyhjm( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = floatbyhjm(HJMTree,Spread,Settle,Maturity)` prices a floating-rate note from a Heath-Jarrow-Morton interest-rate tree.

`floatbyhjm` computes prices of vanilla floating-rate notes, amortizing floating-rate notes, capped floating-rate notes, floored floating-rate notes and collared floating-rate notes.

`[Price,PriceTree] = floatbyhjm( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Floating-Rate Note Using an HJM Tree

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 interest-rate information needed to price the note.

```
load deriv.mat;
```

Define the floating-rate note using the required arguments. Other arguments use defaults.

```
Spread = 20;
Settle = datetime(2000,1,1);
Maturity = datetime(2003,1,1);
```

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

```
Price = floatbyhjm(HJMTree, Spread, Settle, Maturity)
```

```
Price = 100.5529
```

Price an Amortizing Floating-Rate Note

Price an amortizing floating-rate note using the `Principal` input argument to define the amortization schedule.

Create the `RateSpec`.

```
Rates = [0.03583; 0.042147; 0.047345; 0.052707; 0.054302];
ValuationDate = datetime(2011,11,15);
StartDates = ValuationDate;
EndDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate,'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
          EndTimes: [5x1 double]
        StartTimes: [5x1 double]
          EndDates: [5x1 double]
        StartDates: 734822
    ValuationDate: 734822
            Basis: 0
     EndMonthRule: 1
```

Create the floating-rate instrument using the following data:

```
Settle = datetime(2011,11,15);
Maturity = datetime(2015,11,15);
Spread = 15;
```

Define the floating-rate note amortizing schedule.

```
Principal ={{datetime(2012,11,15) 100;datetime(2013,11,15) 70;datetime(2014,11,15) 40;datetime(20
```

Build the HJM tree using the following data:

```
MatDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
HJMTimeSpec = hjmtimespec(RateSpec.ValuationDate, MatDates);
Volatility = [.10; .08; .06; .04];
CurveTerm = [ 1; 2; 3; 4];
HJMVolSpec = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
HJMT = hjmtree(HJMVolSpec,RateSpec,HJMTimeSpec);
```

Compute the price of the amortizing floating-rate note.

```
Price = floatbyhjm(HJMT, Spread, Settle, Maturity, 'Principal', Principal)
```

```
Price = 100.3059
```

Price a Collar with a Floating-Rate Note

Price a collar with a floating-rate note using the `CapRate` and `FloorRate` input argument to define the collar pricing.

Price a portfolio of collared floating-rate notes using the following data:

```
Rates = [0.0287; 0.03024; 0.03345; 0.03861; 0.04033];
ValuationDate = datetime(2012,4,1);
```

```
StartDates = ValuationDate;
EndDates = [datetime(2013,4,1) ; datetime(2014,4,1) ; datetime(2015,4,1) ; datetime(2016,4,1) ; d
Compounding = 1;
```

Create the `RateSpec`.

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

Build the HJM tree with the following data:

```
MatDates = [datetime(2013,4,1) ; datetime(2014,4,1) ; datetime(2015,4,1) ; datetime(2016,4,1) ; d
HJMTimeSpec = hjmtimespec(RateSpec.ValuationDate, MatDates);
Volatility = [.10; .08; .06; .04];
CurveTerm = [ 1; 2; 3; 4];
HJMVolSpec = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
HJMT = hjmtree(HJMVolSpec,RateSpec,HJMTimeSpec);
```

Create the floating-rate note instrument.

```
Settle = datetime(2012,4,1);
Maturity = datetime(2016,4,1);
Spread = 10;
Principal = 100;
```

Compute the price of two capped collared floating-rate notes.

```
CapStrike = [0.04;0.055];
PriceCapped = floatbyhjm(HJMT, Spread, Settle, Maturity,...
'CapRate', CapStrike)
```

```
PriceCapped = 2×1

   98.9986
  100.2051
```

Compute the price of two collared floating-rate notes.

```
FloorStrike = [0.035;0.040];
PriceCollared = floatbyhjm(HJMT, Spread, Settle, Maturity,...
'CapRate', CapStrike, 'FloorRate', FloorStrike)
```

```
PriceCollared = 2×1

   99.9246
  102.2321
```

Input Arguments

`HJMTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `hjmtree`

Data Types: `struct`

Spread — Number of basis points over the reference rate
vector

Number of basis points over the reference rate, specified as a NINST-by-1 vector.

Data Types: double

Settle — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, floatbyhjm also accepts serial date numbers as inputs, but they are not recommended.

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.

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each floating-rate note.

To support existing code, floatbyhjm also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree] = floatbyhjm(HJMTree,Spread,Settle,Maturity,'Basis',3)

FloatReset — Frequency of payments per year
1 (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of 'FloatReset' and a NINST-by-1 vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates.

Data Types: double

Basis — Day count basis
0 (actual/actual) (default) | integer from 0 to 13

Day count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer `[0, 1]` using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.

- 1 = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: logical

AdjustCashFlowsBasis — Flag to adjust cash flows based on actual period day count
false (default) | value of 0 (false) or 1 (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of 'AdjustCashFlowsBasis' and a NINST-by-1 vector of logicals with values of 0 (false) or 1 (true).

Data Types: logical

Holidays — Holidays used in computing business days
if not specified, the default is to use holidays.m (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of 'Holidays' and MATLAB dates using a NHolidays-by-1 vector.

Data Types: datetime

BusinessDayConvention — Business day conventions
actual (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of 'BusinessDayConvention' and a character vector or a N-by-1 cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- actual — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- follow — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- modifiedfollow — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- previous — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- modifiedprevious — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: char | cell

CapRate — Annual cap rate
decimal

Annual cap rate, specified as the comma-separated pair consisting of 'CapRate' and a NINST-by-1 decimal annual rate or NINST-by-1 cell array, where each element is a NumDates-by-2 cell array, and the cell array first column is dates, and the second column is associated cap rates. The date indicates the last day that the cap rate is valid.

Data Types: double | cell

`FloorRate` — Annual floor rate
decimal

Annual floor rate, specified as the comma-separated pair consisting of `'FloorRate'` and a `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array, and the cell array first column is dates, and the second column is associated floor rates. The date indicates the last day that the floor rate is valid.

Data Types: `double` | `cell`

Output Arguments

`Price` — Expected floating-rate note prices at time 0
vector

Expected floating-rate note prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.PBush` contains the clean prices.
- `PriceTree.AIBush` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.

More About

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.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `floatbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bondbyhjm` | `capbyhjm` | `cfbyhjm` | `fixedbyhjm` | `floorbyhjm` | `hjmtree` | `swapbyhjm`

Topics

“Computing Instrument Prices” on page 2-81
“Floating-Rate Note” on page 2-10
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

floatbyhw

Price floating-rate note from Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = floatbyhw(HWTree,Spread,Settle,Maturity)
[Price,PriceTree] = floatbyhw( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = floatbyhw(HWTree,Spread,Settle,Maturity)` prices a floating-rate note from a Hull-White interest-rate tree.

`floatbyhw` computes prices of vanilla floating-rate notes, amortizing floating-rate notes, capped floating-rate notes, floored floating-rate notes and collared floating-rate notes.

Note Alternatively, you can use the `FloatBond` object to price floating-rate bond instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = floatbyhw( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Floating-Rate Note Using a Hull-White Tree

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;
```

Define the floating-rate note using the required arguments. Other arguments use defaults.

```
Spread = 20;
Settle = datetime(2004,1,1);
Maturity = datetime(2007,1,1);
```

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

```
Price = floatbyhw(HWTree, Spread, Settle, Maturity)
```

```
Price = 100.5618
```

Price an Amortizing Floating-Rate Note

Price an amortizing floating-rate note using the `Principal` input argument to define the amortization schedule.

Create the `RateSpec`.

```
Rates = [0.03583; 0.042147; 0.047345; 0.052707; 0.054302];
ValuationDate = datetime(2011,11,15);
StartDates = ValuationDate;
EndDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate,'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: [5x1 double]
       StartDates: 734822
    ValuationDate: 734822
            Basis: 0
     EndMonthRule: 1
```

Create the floating-rate instrument using the following data:

```
Settle = datetime(2011,11,15);
Maturity = datetime(2015,11,15);
Spread = 15;
```

Define the floating-rate note amortizing schedule.

```
Principal ={{datetime(2012,11,15) 100;datetime(2013,11,15) 70;datetime(2014,11,15) 40;datetime(20
```

Build the HW tree and assume the volatility is 10%.

```
VolDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
VolCurve = 0.1;
AlphaDates = datetime(2017,11,15);
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RateSpec, HWTimeSpec);
```

Compute the price of the amortizing floating-rate note.

```
Price = floatbyhw(HWT, Spread, Settle, Maturity, 'Principal', Principal)
```

```
Price = 100.3059
```

Price a Collar with a Floating-Rate Note

Price a collar with a floating-rate note using the `CapRate` and `FloorRate` input argument to define the collar pricing.

Price two collared floating-rate notes using the following data:

```
Rates = [0.0287; 0.03024; 0.03345; 0.03861; 0.04033];
ValuationDate = datetime(2012,4,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,4,1) ; datetime(2014,4,1) ; datetime(2015,4,1) ; datetime(2016,4,1) ; d
Compounding = 1;
```

Create the `RateSpec`.

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

Build the HW tree and assume the volatility to be 5%.

```
VolDates = [datetime(2013,4,1) ; datetime(2014,4,1) ; datetime(2015,4,1) ; datetime(2016,4,1) ; d
VolCurve = 0.05;
AlphaDates = datetime(2018,11,15);
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RateSpec, HWTimeSpec);
```

Create the floating-rate note instrument.

```
Settle = datetime(2012,4,1);
Maturity = datetime(2016,4,1);
Spread = 10;
Principal = 100;
```

Compute the price of a vanilla floater.

```
Price = floatbyhw(HWT, Spread, Settle, Maturity)
```

```
Price = 100.3680
```

Compute the price of the collared floating-rate notes.

```
CapStrike = {{datetime(2014,4,1) 0.045;datetime(2015,4,1) 0.05;...
 datetime(2016,4,1) 0.06}; 0.06};

FloorStrike = {{datetime(2014,4,1) 0.035;datetime(2015,4,1)  0.04;...
 datetime(2016,4,1) 0.05}; 0.03};
PriceCollared = floatbyhw(HWT, Spread, Settle, Maturity,....
'CapRate', CapStrike,'FloorRate', FloorStrike)
```

```
PriceCollared = 2×1

  102.0458
  100.9299
```

Pricing a Floating-Rate Note When the Reset Dates Are Not Tree Level Dates

When using `floatbyhw` to price floating-rate notes, there are cases where the dates specified in the HW tree `TimeSpec` are not aligned with the cash flow dates.

Price floating-rate notes using the following data:

```
ValuationDate = datetime(2013,9,1);
Rates = [0.0001; 0.0001; 0.0010; 0.0015];
EndDates = [datetime(2013,12,1);datetime(2014,3,1);datetime(2014,6,1);datetime(2014,9,1)];
```

Create the `RateSpec`.

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

Build the HW tree.

```
Volcurve = 0.1;
Alpha = 0.01;
HWVolatilitySpec = hwvolspec(RateSpec.ValuationDate, ...
                             EndDates, Volcurve,...
                             EndDates, Alpha);

HWTimeSpec = hwtimespec(RateSpec.ValuationDate, EndDates, 1);

HWT = hwtree(HWVolatilitySpec, RateSpec, HWTimeSpec);
```

Compute the price of the floating-rate note using the following data.

```
Spread = 10;
Settle = datetime(2013,9,1);
Maturity = datetime(2014,6,1);
Reset = 2;

Price = floatbyhw(HWT, Spread, Settle, Maturity, 'FloatReset', Reset)

Error using floatengbytrintree (line 318)
Instrument '1 ' has cash flow dates that span across tree nodes.

Error in floatbyhw (line 136)
        [Price, PriceTree, CFTree] = floatengbytrintree(HWTree, Spread, Settle, Maturity, OArgs{:});
```

This error indicates that it is not possible to determine the applicable rate used to calculate the payoff at the reset dates, given that the applicable rate needed cannot be calculated (the information was lost due to the recombination of the tree nodes). Note, if the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates. The simplest solution is to place the tree levels at the cash flow dates of the instrument, which is done by specifying `HWTimeSpec`. It is also acceptable to have reset dates between tree levels, as long as there are reset dates on the tree levels.

To recover from this error, build a tree that lines up with the instrument.

```
Basis = intenvget(RateSpec, 'Basis');
EOM = intenvget(RateSpec, 'EndMonthRule');
resetDates = cfdates(ValuationDate, Maturity, Reset, Basis, EOM);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate,resetDates, Reset);
HWT = hwtree(HWVolatilitySpec, RateSpec, HWTimeSpec);

Price = floatbyhw(HWT, Spread, RateSpec.ValuationDate, ...
                  Maturity, 'FloatReset', Reset)
```

```
Price =

  100.0748
```

Input Arguments

`HWTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `hwtree`

Data Types: `struct`

`Spread` — Number of basis points over the reference rate

vector

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floatbyhw` also accepts serial date numbers as inputs, but they are not recommended.

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.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each floating-rate note.

To support existing code, `floatbyhw` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = floatbyhw(HWTree,Spread,Settle,Maturity,'Basis',3)`

`FloatReset` — Frequency of payments per year

1 (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of `'FloatReset'` and a `NINST`-by-`1` vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMOnthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`CapRate` — Annual cap rate
decimal

Annual cap rate, specified as the comma-separated pair consisting of `'CapRate'` and a `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array, and the cell array first column is dates, and the second column is associated cap rates. The date indicates the last day that the cap rate is valid.

Data Types: `double` | `cell`

`FloorRate` — Annual floor rate
decimal

Annual floor rate, specified as the comma-separated pair consisting of `'FloorRate'` and a `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array, and the cell array first column is dates, and the second column is associated floor rates. The date indicates the last day that the floor rate is valid.

Data Types: `double` | `cell`

Output Arguments

`Price` — Expected floating-rate note prices at time 0
vector

Expected floating-rate note prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

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.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floatbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bondbyhw` | `capbyhw` | `cfbyhw` | `fixedbyhw` | `floorbyhw` | `hwtree` | `swapbyhw` | `FloatBond`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Compute LIBOR Fallback” on page 2-192
“Floating-Rate Note” on page 2-10
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

floatbyzero

Price floating-rate note from set of zero curves

Syntax

```
[Price,DirtyPrice,CFlowAmounts,CFlowDates] = floatbyzero(RateSpec,Spread,
Settle,Maturity)
[Price,DirtyPrice,CFlowAmounts,CFlowDates] = floatbyzero( ___ ,Name,Value)
```

Description

`[Price,DirtyPrice,CFlowAmounts,CFlowDates] = floatbyzero(RateSpec,Spread, Settle,Maturity)` prices a floating-rate note from a set of zero curves.

`floatbyzero` computes prices of vanilla floating-rate notes and amortizing floating-rate notes.

Note Alternatively, you can use the `FloatBond` object to price floating-rate bond instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,DirtyPrice,CFlowAmounts,CFlowDates] = floatbyzero( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Floating-Rate Note Using a Set of Zero Curves

Price a 20-basis point floating-rate note using a set of zero curves.

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

```
load deriv.mat;
```

Define the floating-rate note using the required arguments. Other arguments use defaults.

```
Spread = 20;
Settle = datetime(2000,1,1);
Maturity = datetime(2003,1,1);
```

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

```
Price = floatbyzero(ZeroRateSpec, Spread, Settle, Maturity)
```

```
Price = 100.5529
```

Price an Amortizing Floating-Rate Note

Price an amortizing floating-rate note using the `Principal` input argument to define the amortization schedule.

Create the `RateSpec`.

```
Rates = [0.03583; 0.042147; 0.047345; 0.052707; 0.054302];
ValuationDate = datetime(2011,11,15);
StartDates = ValuationDate;
EndDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate,'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
          EndTimes: [5x1 double]
        StartTimes: [5x1 double]
          EndDates: [5x1 double]
        StartDates: 734822
    ValuationDate: 734822
            Basis: 0
     EndMonthRule: 1
```

Create the floating-rate instrument using the following data:

```
Settle = datetime(2011,11,15);
Maturity = datetime(2015,11,15);
Spread = 15;
```

Define the floating-rate note amortizing schedule.

```
Principal ={{datetime(2012,11,15) 100;datetime(2013,11,15) 70;datetime(2014,11,15) 40;datetime(20
```

Compute the price of the amortizing floating-rate note.

```
Price  = floatbyzero(RateSpec, Spread, Settle, Maturity, 'Principal', Principal)
```

```
Price = 100.3059
```

Specify the Rate at the Instrument's Starting Date When It Cannot Be Obtained from the RateSpec

If `Settle` is not on a reset date of a floating-rate note, `floatbyzero` attempts to obtain the latest floating rate before `Settle` from `RateSpec` or the `LatestFloatingRate` parameter. When the reset date for this rate is out of the range of `RateSpec` (and `LatestFloatingRate` is not specified), `floatbyzero` fails to obtain the rate for that date and generates an error. This example shows how to use the `LatestFloatingRate` input parameter to avoid the error.

Create the error condition when a floating-rate instrument's `StartDate` cannot be determined from the `RateSpec`.

```
load deriv.mat;

Spread = 20;
Settle = datetime(2000,1,1);
Maturity = datetime(2003,12,1);

Price = floatbyzero(ZeroRateSpec, Spread, Settle, Maturity)

Error using floatbyzero (line 256)
The rate at the instrument starting date cannot be obtained from RateSpec.
 Its reset date (01-Dec-1999) is out of the range of dates contained in RateSpec.
 This rate is required to calculate cash flows at the instrument starting date.
 Consider specifying this rate with the 'LatestFloatingRate' input parameter.
```

Here, the reset date for the rate at `Settle` was 01-Dec-1999, which was earlier than the valuation date of `ZeroRateSpec` (01-Jan-2000). This error can be avoided by specifying the rate at the instrument's starting date using the `LatestFloatingRate` name-value pair argument.

Define `LatestFloatingRate` and calculate the floating-rate price.

```
Price = floatbyzero(ZeroRateSpec, Spread, Settle, Maturity, 'LatestFloatingRate', 0.03)

Price =

  100.0285
```

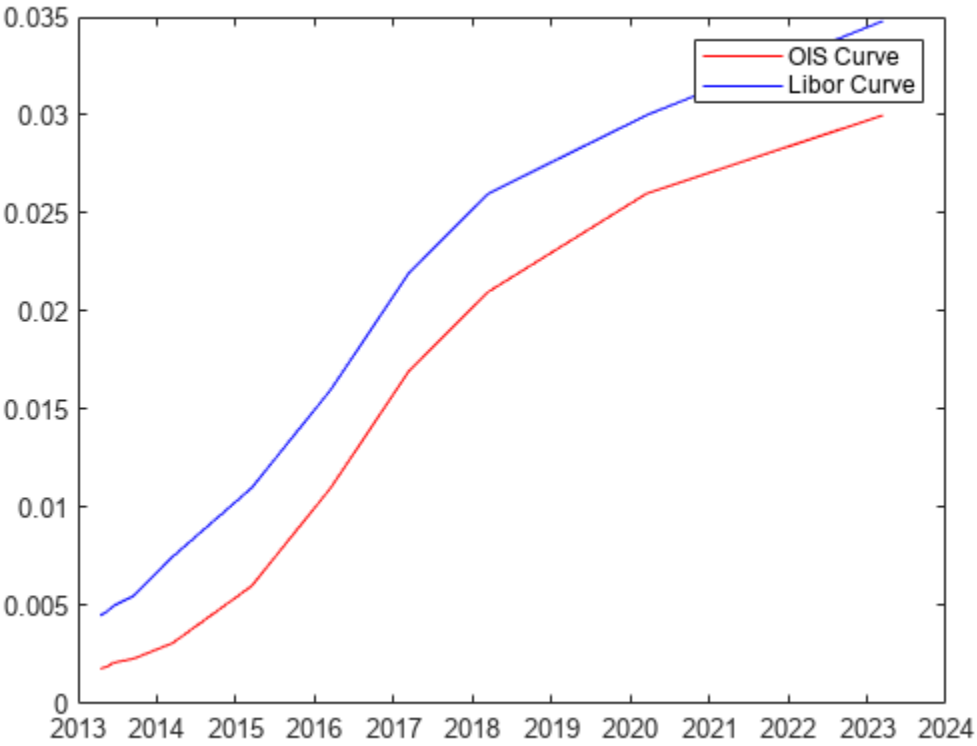
Price a Floating-Rate Note Using a Different Curve to Generate Floating Cash Flows

Define the OIS and Libor rates.

```
Settle = datetime(2013,3,15);
CurveDates = daysadd(Settle,360*[1/12 2/12 3/12 6/12 1 2 3 4 5 7 10],1);
OISRates = [.0018 .0019 .0021 .0023 .0031 .006  .011 .017 .021 .026 .03]';
LiborRates = [.0045 .0047 .005 .0055 .0075 .011 .016 .022 .026 .030 .0348]';
```

Plot the dual curves.

```
figure,plot(CurveDates,OISRates,'r');hold on;plot(CurveDates,LiborRates,'b')
datetick
legend({'OIS Curve', 'Libor Curve'})
```



Create an associated `RateSpec` for the OIS and Libor curves.

```
OISCurve = intenvset('Rates',OISRates,'StartDate',Settle,'EndDates',CurveDates);
LiborCurve = intenvset('Rates',LiborRates,'StartDate',Settle,'EndDates',CurveDates);
```

Define the floating-rate note.

```
Maturity = datetime(2018,3,15);
```

Compute the price for the floating-rate note. The `LiborCurve` term structure will be used to generate the floating cash flows of the floater instrument. The `OISCurve` term structure will be used for discounting the cash flows.

```
Price = floatbyzero(OISCurve,0,Settle,Maturity,'ProjectionCurve',LiborCurve)
```

```
Price = 102.4214
```

Some instruments require using different interest-rate curves for generating the floating cash flows and discounting. This is when the `ProjectionCurve` parameter is useful. When you provide both `RateSpec` and `ProjectionCurve`, `floatbyzero` uses the `RateSpec` for the purpose of discounting and it uses the `ProjectionCurve` for generating the floating cash flows.

Input Arguments

`RateSpec` — Annualized zero rate term structure
structure

Annualized zero rate term structure, specified using `intenvset` to create a `RateSpec`.

Data Types: `struct`

`Spread` — Number of basis points over the reference rate
vector

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floatbyzero` also accepts serial date numbers as inputs, but they are not recommended.

`Settle` must be earlier than `Maturity`.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each floating-rate note.

To support existing code, `floatbyzero` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,DirtyPrice,CFlowAmounts,CFlowDates] = floatbyzero(RateSpec,Spread,Settle,Maturity,'Principal',Principal)`

`FloatReset` — Frequency of payments per year
`1` (default) | vector

Frequency of payments per year, specified as the comma-separated pair consisting of `'FloatReset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector.

- 0 = actual/actual

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`LatestFloatingRate` — Rate for the next floating payment
if not specified, the floating rate at the previous reset date is computed from `RateSpec` (default) | numeric

Rate for the next floating payment set at the last reset date, specified as the comma-separated pair consisting of `'LatestFloatingRate'` and a `NINST`-by-`1`.

Data Types: `double`

`ProjectionCurve` — Rate curve used in generating future forward rates
if not specified, then `RateSpec` is used both for discounting cash flows and projecting future forward rates (default) | structure

The rate curve to be used in generating the future forward rates, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a structure created using `intenvset`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `struct`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` vector of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

Output Arguments

`Price` — Floating-rate note prices

matrix

Floating-rate note prices, returned as a (`NINST`) by number of curves (`NUMCURVES`) matrix. Each column arises from one of the zero curves.

`DirtyPrice` — Dirty note price

matrix

Dirty note price (clean + accrued interest), returned as a `NINST`- by-`NUMCURVES` matrix. Each column arises from one of the zero curves.

`CFlowAmounts` — Cash flow amounts

matrix

Cash flow amounts, returned as a `NINST`- by-`NUMCFS` matrix of cash flows for each note. If there is more than one curve specified in the `RateSpec` input, then the first `NCURVES` rows correspond to the first note, the second `NCURVES` rows correspond to the second note, and so on.

`CFlowDates` — Cash flow dates

matrix

Cash flow dates, returned as a `NINST`- by-`NUMCFS` matrix of payment dates for each note.

More About

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.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floatbyzero` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bondbyzero` | `cfbyzero` | `fixedbyzero` | `swapbyzero` | `intenvset` | `FloatBond`

Topics

“Pricing Using Interest-Rate Term Structure” on page 2-61
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Compute LIBOR Fallback” on page 2-192
“Floating-Rate Note” on page 2-10
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

floorbybdt

Price floor instrument from Black-Derman-Toy interest-rate tree

Syntax

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

Description

`[Price,PriceTree] = floorbybdt(BDTTree,Strike,Settle,Maturity)` computes the price of a floor instrument from a Black-Derman-Toy interest-rate tree. `floorbybdt` computes prices of vanilla floors and amortizing floors.

Note Alternatively, you can use the `Floor` object to price floor instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = floorbybdt( ___ ,FloorReset,Basis,Principal,Options)` adds optional arguments.

Examples

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 = datetime(2000,1,1);
Maturity = datetime(2004,1,1);
```

Use `floorbybdt` to compute the price of the floor instrument.

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

```
Price = 0.2428
```

Price a 10% Floor Instrument Using a Newly Created BDT Interest-Rate Tree

First set the required arguments for the three needed specifications.

```
Compounding = 1;
ValuationDate = datetime(2000,1,1);
StartDate = ValuationDate;
EndDates = [datetime(2001,1,1) ; datetime(2002,1,1) ; datetime(2003,1,1) ; datetime(2004,1,1) ;da
Rates = [.1; .11; .12; .125; .13];
Volatility = [.2; .19; .18; .17; .16];
```

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);
```

Create the BDT tree from the specifications.

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

```
BDTTree = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 4]
        dObs: [730486 730852 731217 731582 731947]
        TFwd: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [4]}
      CFlowT: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [5]}
     FwdTree: {1x5 cell}
```

Set the floor arguments. Remaining arguments will use defaults.

```
FloorStrike = 0.10;
Settlement = ValuationDate;
Maturity = datetime(2002,1,1);
FloorReset = 1;
```

Use `floorbybdt` to find the price of the floor instrument.

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

```
Price = 0.0863
```

Compute the Price of an Amortizing Floor Using the BDT Model

Define the `RateSpec`.

```
Rates = [0.03583; 0.042147; 0.047345; 0.052707; 0.054302];
ValuationDate = datetime(2011,11,15);
StartDates = ValuationDate;
EndDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,1
Compounding = 1;
```

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: [5x1 double]
       StartDates: 734822
    ValuationDate: 734822
            Basis: 0
     EndMonthRule: 1
```

Define the floor instrument.

```
Settle = datetime(2011,11,15);
Maturity = datetime(2015,11,15);
Strike = 0.039;
Reset = 1;
Principal ={{datetime(2012,11,15) 100;datetime(2013,11,15) 70;datetime(2014,11,15) 40;datetime(20
```

Build the BDT Tree.

```
BDTTimeSpec = bdttimespec(ValuationDate, EndDates);
Volatility = 0.10;
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Volatility*ones(1,length(EndDates))');
BDTTree = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec)
```

```
BDTTree = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 4]
        dObs: [734822 735188 735553 735918 736283]
        TFwd: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [4]}
      CFlowT: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [5]}
     FwdTree: {1x5 cell}
```

Price the amortizing floor.

```
Basis = 0;
Price = floorbybdt(BDTTree, Strike, Settle, Maturity, Reset, Basis, Principal)
```

```
Price = 0.3060
```

Input Arguments

BDTTree — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bdttree`.

Data Types: `struct`

`Strike` — Rate at which floor is exercised
decimal

Rate at which the floor is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for floor
datetime array | string array | date character vector

Settlement date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every floor is set to the `ValuationDate` of the BDT tree. The floor argument `Settle` is ignored.

To support existing code, `floorbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for floor
datetime array | string array | date character vector

Maturity date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`FloorReset` — Reset frequency payment per year
`1` (default) | numeric

(Optional) Reset frequency payment per year, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis representing the basis used when annualizing the input forward rate, specified as a `NINST`-by-`1` vector of integers.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing floor.

Data Types: `double` | `cell`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected price of floor at time 0
vector

Expected price of the floor at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure with values of floor at each node
vector

Tree structure with values of the floor at each node, returned as a MATLAB structure of trees containing vectors of instrument prices and a vector of observation times for each node:

- `PriceTree.PTree` contains floor prices.
- `PriceTree.tObs` contains the observation times.

More About

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:

$\max(FloorRate - CurrentRate, 0)$

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floorbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdttree` | `capbybdt` | `cfbybdt` | `swapbybdt` | `floorbynormal` | `Floor`

Topics

“Computing Instrument Prices” on page 2-81
“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Floor” on page 2-12
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

floorbybk

Price floor instrument from Black-Karasinski interest-rate tree

Syntax

```
[Price,PriceTree] = floorbybk(BKTree,Strike,Settle,Maturity)
[Price,PriceTree] = floorbybk( ___ ,Reset,Basis,Principal,Options)
```

Description

`[Price,PriceTree] = floorbybk(BKTree,Strike,Settle,Maturity)` computes the price of a floor instrument from a Black-Karasinski interest-rate tree. `floorbybk` computes prices of vanilla floors and amortizing floors.

Note Alternatively, you can use the `Floor` object to price floor instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = floorbybk( ___ ,Reset,Basis,Principal,Options)` adds optional arguments.

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 = datetime(2004,1,1);
Maturity = datetime(2007,1,1);
```

Use `floorbybk` to compute the price of the floor instrument.

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

```
Price = 0.2061
```

Compute the Price of an Amortizing and Vanilla Floors Using the BK Model

Load `deriv.mat` to specify the `BKTree` and then define the floor instrument.

```
load deriv.mat;
Settle = datetime(2004,1,1);
Maturity = datetime(2008,1,1);
Strike = 0.045;
Reset = 1;
Principal ={{datetime(2005,1,1) 100;datetime(2006,1,1) 60;datetime(2007,1,1) 30;datetime(2008,1,1
            100};
```

Price the amortizing and vanilla floors.

```
Basis = 1;
Price = floorbybk(BKTree, Strike, Settle, Maturity, Reset, Basis, Principal)
```

```
Price = 2×1

    2.2000
    2.5564
```

Input Arguments

BKTree — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bktree`.

Data Types: `struct`

Strike — Rate at which cap is exercised

decimal

Rate at which cap is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

Settle — Settlement date for floor

datetime array | string array | date character vector

Settlement date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every floor is set to the `ValuationDate` of the BK tree. The floor argument `Settle` is ignored.

To support existing code, `floorbybk` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date for floor

datetime array | string array | date character vector

Maturity date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorbybk` also accepts serial date numbers as inputs, but they are not recommended.

Reset — Reset frequency payment per year

1 (default) | numeric

(Optional) Reset frequency payment per year, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis representing the basis used when annualizing the input forward rate, specified as a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing floor.

Data Types: `double` | `cell`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected price of floor at time 0
vector

Expected price of the floor at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure with values of floor at each node
vector

Tree structure with values of the floor at each node, returned as a MATLAB structure of trees containing vectors of instrument prices and a vector of observation times for each node:

- `PriceTree.PTree` contains floor prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

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:

$\max(FloorRate - CurrentRate, 0)$

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floorbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bktree` | `capbybk` | `cfbybk` | `swapbybk` | `floorbynormal` | `Floor`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Floor” on page 2-12
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

floorbycir

Price floor instrument from Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = floorbycir(CIRTree,Strike,Settle,Maturity)
[Price,PriceTree] = floorbycir(___,Name,Value)
```

Description

`[Price,PriceTree] = floorbycir(CIRTree,Strike,Settle,Maturity)` computes the price of a floor instrument from a Cox-Ingersoll-Ross (CIR) interest-rate tree. `floorbycir` computes prices of vanilla floors and amortizing floors using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = floorbycir(___,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Floor Using a CIR Interest-Rate Tree

Define the `Strike` for a floor.

```
Strike = 0.02;
```

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = 'Jan-1-2017';
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',EndD
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = datetime(2017,1,1);
Maturity = datetime(2021,1,1);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
```

```
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [736696 737061 737426 737791]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Price the 2% floor.

```
[Price,PriceTree] = floorbycir(CIRT,Strike,Settle,Maturity)
```

```
Price = 1.4211e-14
```

```
PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
       tObs: [0 1 2 3 4]
      PTree: {1x5 cell}
    Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
      Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `cirtree`.

Data Types: `struct`

`Strike` — Rate at which cap is exercised

decimal

Rate at which cap is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for floor

datetime array | string array | date character vector

Settlement date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every floor is set to the `ValuationDate` of the CIR tree. The floor argument `Settle` is ignored.

To support existing code, `floorbycir` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for floor

datetime array | string array | date character vector

Maturity date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorbycir` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = floorbycir(CIRTree,CouponRate,Settle,Maturity,'Basis',3)`

`FloorReset` — Reset frequency payment per year

`1` (default) | numeric

Reset frequency payment per year, specified as the comma-separated pair consisting of `'FloorReset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward rate, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount

`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array.

For the NINST-by-1 cell array, each element is a NumDates-by-2 cell array where the first column is dates, and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use Principal to pass a schedule to compute the price for an amortizing floor.

Data Types: double | cell

Output Arguments

Price — Expected price of floor at time 0

vector

Expected price of the floor at time 0, returned as a NINST-by-1 vector.

PriceTree — Tree structure with values of floor at each node

vector

Tree structure with values of the floor at each node, returned as a MATLAB structure of trees containing vectors of instrument prices and a vector of observation times for each node:

- PriceTree.PTree contains floor prices.
- PriceTree.tObs contains the observation times.
- PriceTree.Connect contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are NumNodes elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- PriceTree.Probs contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

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:

$\max(FloorRate - CurrentRate, 0)$

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although floorbycir supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J.,and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir` | `instfloor`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Floor” on page 2-12
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

floorbyblk

Price floors using Black option pricing model

Syntax

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

Description

`[FloorPrice,Floorlets] = floorbyblk(RateSpec,Strike,Settle,Maturity, Volatility)` price floors using the Black option pricing model. `floorbyblk` computes prices of vanilla floors and amortizing floors.

Note Alternatively, you can use the `Floor` object to price floor instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[FloorPrice,Floorlets] = floorbyblk( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Floor Using the Black Option Pricing Model

This example shows how to price a floor using the Black option pricing model. 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.

```
ValuationDate = datetime(2008,1,1);
EndDates = datetime(2010,4,1);
Rates = 0.069394;
Compounding = -1;
Basis = 1;

% calculate the RateSpec
RateSpec = intenvset('ValuationDate', ValuationDate, ...
'StartDates', ValuationDate,'EndDates', EndDates, ...
'Rates', Rates,'Compounding', Compounding,'Basis', Basis);

Settle = datetime(2009,1,1); % floor starts in a year
Maturity = datetime(2009,4,1);
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
```

Price a Floor Using a Different Curve to Generate the Future Forward Rates

Define the OIS and Libor rates.

```
Settle = datetime(2013,3,15);
CurveDates = daysadd(Settle,360*[1/12 2/12 3/12 6/12 1 2 3 4 5 7 10],1);
OISRates = [.0018 .0019 .0021 .0023 .0031 .006  .011 .017 .021 .026 .03]';
LiborRates = [.0045 .0047 .005 .0055 .0075 .0109  .0162 .0216 .0262 .0309 .0348]';
```

Create an associated `RateSpec` for the OIS and Libor curves.

```
OISCurve = intenvset('Rates',OISRates,'StartDate',Settle,'EndDates',CurveDates,'Compounding',2,'
LiborCurve = intenvset('Rates',LiborRates,'StartDate',Settle,'EndDates',CurveDates,'Compounding'
```

Define the Floor instruments.

```
Maturity = [datetime(2018,3,15) ; datetime(2020,3,15)];
Strike = [.04;.05];
BlackVol = .2;
```

Price the floor instruments using the term structure `OISCurve` both for discounting the cash flows and generating future forward rates.

```
[Price, Floorlets] = floorbyblk(OISCurve, Strike, Settle, Maturity, BlackVol)
```

```
Price = 2×1

    9.9808
   16.9057


Floorlets = 2×7

    3.6783    3.0706    1.8275    0.7280    0.6764       NaN       NaN
    4.6753    4.0587    2.7921    1.4763    1.3442    1.4130    1.1462
```

Price the floor instruments using the term structure `LiborCurve` to generate future forward rates. The term structure `OISCurve` is used for discounting the cash flows.

```
[PriceLC, FloorletsLC] = floorbyblk(OISCurve, Strike, Settle, Maturity, BlackVol,'ProjectionCurve
```

```
PriceLC = 2×1

    8.0524
   14.3184


FloorletsLC = 2×7
```

```
    3.2385    2.5338    1.2895    0.5889    0.4017       NaN       NaN
    4.2355    3.5219    2.2286    1.2751    0.9169    1.1698    0.9706
```

Compute the Price of an Amortizing Floor Using the Black Model

Define the RateSpec.

```
Rates = [0.0358; 0.0421; 0.0473; 0.0527; 0.0543];
ValuationDate = datetime(2011,11,15);
StartDates = ValuationDate;
EndDates = [datetime(2012,11,15) ; datetime(2013,11,15) ; datetime(2014,11,15) ; datetime(2015,11
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate,'StartDates', StartDates,...
             'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
          EndTimes: [5x1 double]
        StartTimes: [5x1 double]
          EndDates: [5x1 double]
        StartDates: 734822
    ValuationDate: 734822
            Basis: 0
     EndMonthRule: 1
```

Define the floor instrument.

```
Settle = datetime(2011,11,15);
Maturity = datetime(2015,11,15);
Strike = 0.05;
Reset = 2;
Principal ={{datetime(2012,11,15) 100;datetime(2013,11,15) 70;datetime(2014,11,15) 40;datetime(20
```

Price the amortizing floor.

```
Volatility = 0.20;
Price = floorbyblk(RateSpec, Strike, Settle, Maturity, Volatility,...
'Reset',Reset,'Principal', Principal)
```

```
Price = 1.9315
```

Price a Floor Using the Shifted Black Model

Create the RateSpec.

```
ValuationDate = datetime(2016,3,1);
EndDates = [datetime(2017,3,1) ; datetime(2018,3,1) ; datetime(2019,3,1) ; datetime(2020,3,1) ; d
Rates = [-0.21; -0.12; 0.01; 0.10; 0.20]/100;
```

```
Compounding = 1;
Basis = 1;

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

RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: [5x1 double]
       StartDates: 736390
    ValuationDate: 736390
            Basis: 1
     EndMonthRule: 1
```

Price the floor with a negative strike using the Shifted Black model.

```
Settle = datetime(2016,6,1); % Floor starts in 3 months.
Maturity = datetime(2016,9,1);
ShiftedBlackVolatility = 0.31;
FloorRate = -0.001;  % -0.1 percent strike.
FloorReset = 4;
Principal = 100000;
Shift = 0.01; % 1 percent shift.

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

FloorPrice = 31.2099
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

Strike — Rate at which floor is exercised

decimal

Rate at which floor is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

Settle — Settlement date for floor

datetime array | string array | date character vector

Settlement date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorbyblk` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for floor
datetime array | string array | date character vector

Maturity date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorbyblk` also accepts serial date numbers as inputs, but they are not recommended.

`Volatility` — Volatilities values
numeric

Volatilities values, specified as a `NINST`-by-`1` vector of numeric values.

The `Volatility` input is not intended for volatility surfaces or cubes. If you specify a matrix for the `Volatility` input, `floorbyblk` internally converts it into a vector. `floorbyblk` assumes that the volatilities specified in the `Volatility` input are flat volatilities, which are applied equally to each of the floorlets.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[FloorPrice,Floorlets] = floorbyblk(RateSpec,Strike,Settle,Maturity,Volatility,'Reset',CapReset,'Principal',100000,'Basis',7)`

`Reset` — Reset frequency payment per year
`1` (default) | numeric

Reset frequency payment per year, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array. When `Principal` is a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array, where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing floor.

Data Types: `double` | `cell`

Basis — Day-count basis of instrument
0 (actual/actual) (default) | integer from 0 to 13

Day-count basis representing the basis used when annualizing the input forward rate, specified as a NINST-by-1 vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: double

ProjectionCurve — Rate curve used in generating future forward rates
if ProjectionCurve is not specified, then RateSpec is used both for discounting cash flows and projecting future forward rates (default) | structure

The rate curve to be used in generating the future forward rates. This structure must be created using intenvset. Use this optional input if the forward curve is different from the discount curve.

Data Types: struct

Shift — Shift in decimals for shifted Black model
0 (no shift) (default) | positive decimal

Shift in decimals for the shifted Black model, specified using a scalar or NINST-by-1 vector of rate shifts in positive decimals. Set this parameter to a positive rate shift in decimals to add a positive shift to the forward rate and strike, which effectively sets a negative lower bound for the forward rate. For example, a Shift of 0.01 is equal to a 1% shift.

Data Types: double

Output Arguments

FloorPrice — Expected price of floor
vector

Expected price of the floor, returned as a NINST-by-1 vector.

`Floorlets` — Floorlets
array

Floorlets, returned as a `NINST`-by-`NCF` array of floorlets, padded with `NaN`s.

More About

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:

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

Shifted Black

The Shifted Black model is essentially the same as the Black's model, except that it models the movements of (F + $Shift$) as the underlying asset, instead of F (which is the forward rate in the case of floorlets).

This model allows negative rates, with a fixed negative lower bound defined by the amount of shift; that is, the zero lower bound of Black's model has been shifted.

Algorithms

Black Model

$$dF = \sigma_{Black} F dw$$

$$call = e^{-\gamma T}[FN(d_1) - KN(d_2)]$$

$$put = e^{-\gamma T}[KN(-d_2) - FN(-d_1)]$$

$$d_1 = \frac{\ln\left(\frac{F}{K}\right) + \left(\frac{\sigma_B{}^2}{2}\right)T}{\sigma_B\sqrt{T}}, \quad d_2 = d_1 - \sigma_B\sqrt{T}$$

$$\sigma_B = \sigma_{Black}$$

Where F is the forward value and K is the strike.

Shifted Black Model

$$dF = \sigma_{Shifted\_Black}(F + Shift)dw$$

$$call = e^{-\gamma T}[(F + Shift)N(d_{s1}) - (K + Shift)N(d_{s2})]$$

$$put = e^{-\gamma T}[(K + Shift)N(-d_{s2}) - (F + Shift)N(-d_{s1})]$$

$$d_{s1} = \frac{\ln\left(\frac{F + Shift}{K + Shift}\right) + \left(\frac{\sigma_{sB}{}^2}{2}\right)T}{\sigma_{sB}\sqrt{T}}, \quad d_{s2} = d_{s1} - \sigma_{sB}\sqrt{T}$$

$$\sigma_{sB} = \sigma_{Shifted\_Black}$$

Where *F+Shift* is the forward value and *K+Shift* is the strike for the shifted version.

Version History

Introduced in R2009a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floorbyblk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`capbyblk` | `intenvset` | `floorbynormal` | `Floor`

Topics

“Floor” on page 2-12
“Work with Negative Interest Rates Using Functions” on page 2-18
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

External Websites

How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

floorbyhjm

Price floor instrument from Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree] = floorbyhjm(HJMTree,Strike,Settle,Maturity)
[Price,PriceTree] = floorbyhjm( ___ ,FloorReset,Basis,Principal,Options)
```

Description

`[Price,PriceTree] = floorbyhjm(HJMTree,Strike,Settle,Maturity)` computes the price of a floor instrument from a Heath-Jarrow-Morton interest-rate tree. `floorbyhjm` computes prices of vanilla floors and amortizing floors.

`[Price,PriceTree] = floorbyhjm( ___ ,FloorReset,Basis,Principal,Options)` adds optional arguments.

Examples

Price a 3% Floor Instrument Using an HJM Forward-Rate Tree

This example shows how to price a 3% floor instrument using an HJM forward-rate tree by loading 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;

Strike = 0.03;
Settle = datetime(2000,1,1);
Maturity = datetime(2004,1,1);

Price = floorbyhjm(HJMTree, Strike, Settle, Maturity)
```

```
Price = 0.0486
```

Compute the Price of an Amortizing Floor Using the HJM Model

Load `deriv.mat` to specify the `HJMTree` and then define the floor instrument.

```
load deriv.mat;
Settle = datetime(2000,1,1);
Maturity = datetime(2004,1,1);
Strike = 0.05;
FloorReset = 1;
Principal ={{datetime(2001,1,1) 100;datetime(2002,1,1) 80;datetime(2003,1,1) 70;datetime(2004,1,1
```

Price the amortizing floor.

```
Price = floorbyhjm(HJMTree, Strike, Settle, Maturity, FloorReset, Principal)

Price = 2.8215
```

Input Arguments

`HJMTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hjmtree`.

Data Types: `struct`

`Strike` — Rate at which floor is exercised

decimal

Rate at which the floor is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for floor

datetime array | string array | date character vector

Settlement date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every floor is set to the `ValuationDate` of the HJM tree. The floor argument `Settle` is ignored.

To support existing code, `floorbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for floor

datetime array | string array | date character vector

Maturity date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`FloorReset` — Reset frequency payment per year

`1` (default) | numeric

(Optional) Reset frequency payment per year, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis representing the basis used when annualizing the input forward rate, specified as a `NINST`-by-`1` vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing floor.

Data Types: `double` | `cell`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected price of floor at time 0
vector

Expected price of the floor at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure with values of floor at each node
vector

Tree structure with values of the floor at each node, returned as a MATLAB structure of trees containing vectors of instrument prices and a vector of observation times for each node:

- `PriceTree.tObs` contains the observation times.
- `PriceTree.PBush` contains the clean prices.

More About

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:

$\max(FloorRate - CurrentRate, 0)$

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floorbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`capbyhjm` | `cfbyhjm` | `hjmtree` | `swapbyhjm` | `floorbynormal`

Topics

“Computing Instrument Prices” on page 2-81
“Floor” on page 2-12
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

floorbyhw

Price floor instrument from Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = floorbyhw(HWTree,Strike,Settle,Maturity)
[Price,PriceTree] = floorbyhw( ___ ,FloorReset,Basis,Principal,Options)
```

Description

`[Price,PriceTree] = floorbyhw(HWTree,Strike,Settle,Maturity)` computes the price of a floor instrument from a Hull-White interest-rate tree. `capbyhw` computes prices of vanilla floors and amortizing floors.

Note Alternatively, you can use the `Floor` object to price floor instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = floorbyhw( ___ ,FloorReset,Basis,Principal,Options)` adds optional arguments.

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 = datetime(2004,1,1);
Maturity = datetime(2007,1,1);
```

Use `floorbyhw` to compute the price of the floor instrument.

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

```
Price = 0.4186
```

Compute the Price of an Amortizing and Vanilla Floors Using the HW Model

Define the `RateSpec`.

```
Rates = [0.035; 0.042; 0.047; 0.052; 0.054];
ValuationDate = datetime(2014,4,1);
StartDates = ValuationDate;
EndDates = datetime(2019,4,1);
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate,'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding)

RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: 737516
       StartDates: 735690
    ValuationDate: 735690
            Basis: 0
     EndMonthRule: 1
```

Define the floor instruments.

```
Settle = datetime(2014,4,1);
Maturity = datetime(2018,4,1);
Strike = 0.05;
FloorReset = 1;
Principal ={{datetime(2015,4,1) 100;datetime(2016,4,1) 60;datetime(2017,4,1) 40;datetime(2018,4,1
            100};
```

Build the HW Tree.

```
VolDates = [datetime(2015,4,1) ; datetime(2016,4,1) ; datetime(2017,4,1) ; datetime(2018,4,1)];
VolCurve = 0.05;
AlphaDates = datetime(2018,4,1);
AlphaCurve = 0.10;

HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
                      AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec)

HWTree = struct with fields:
      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [735690 736055 736421 736786]
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
     FwdTree: {1x4 cell}
```

Price the amortizing and vanilla floors.

```
Basis = 0;
Price  = floorbyhw(HWTree, Strike, Settle, Maturity, FloorReset, Basis, Principal)
```

```
Price = 2×1

    4.8675
   10.3881
```

Input Arguments

`HWTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hwtree`.

Data Types: `struct`

`Strike` — Rate at which floor is exercised

decimal

Rate at which the floor is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for floor

datetime array | string array | date character vector

Settlement date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every floor is set to the `ValuationDate` of the HW tree. The floor argument `Settle` is ignored.

To support existing code, `floorbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for floor

datetime array | string array | date character vector

Maturity date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`FloorReset` — Reset frequency payment per year

`1` (default) | numeric

(Optional) Reset frequency payment per year, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis representing the basis used when annualizing the input forward rate, specified as a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing floor.

Data Types: `double` | `cell`

`Options` — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, specified using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected price of floor at time 0
vector

Expected price of the floor at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure with values of floor at each node
vector

Tree structure with values of the floor at each node, returned as a MATLAB structure of trees containing vectors of instrument prices and a vector of observation times for each node:

- `PriceTree.PTree` contains floor prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

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:

$\max(FloorRate - CurrentRate, 0)$

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floorbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`capbyhw` | `cfbyhw` | `hwtree` | `swapbyhw` | `floorbynormal` | `Floor`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Floor” on page 2-12
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66

“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

floorbylg2f

Price floor using Linear Gaussian two-factor model

Syntax

```
FloorPrice = floorbylg2f(ZeroCurve,a,b,sigma,eta,rho,Strike,Maturity)
FloorPrice = floorbylg2f( ___ ,Name,Value)
```

Description

`FloorPrice = floorbylg2f(ZeroCurve,a,b,sigma,eta,rho,Strike,Maturity)` returns the floor price for a two-factor additive Gaussian interest-rate model.

Note Alternatively, you can use the `Floor` object to price floor instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`FloorPrice = floorbylg2f( ___ ,Name,Value)` adds optional name-value pair arguments.

Note Use the optional name-value pair argument, `Notional`, to pass a schedule to compute the price for an amortizing floor.

Examples

Price a Floor Using a Linear Gaussian Two-Factor Model

Define the `ZeroCurve`, `a`, `b`, `sigma`, `eta`, and `rho` parameters to compute the floor price.

```
Settle = datetime(2007,12,15);

ZeroTimes = [3/12 6/12 1 5 7 10 20 30]';
ZeroRates = [0.033 0.034 0.035 0.040 0.042 0.044 0.048 0.0475]';
CurveDates = daysadd(Settle,360*ZeroTimes,1);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

a = .07;
b = .5;
sigma = .01;
eta = .006;
rho = -.7;

FloorMaturity = daysadd(Settle,360*[1:5 7 10 15 20 25 30],1);

Strike = [0.035 0.037 0.038 0.039 0.040 0.042 0.044 0.046 0.047 0.047 0.047]';

Price = floorbylg2f(irdc,a,b,sigma,eta,rho,Strike,FloorMaturity)
```

```
Price = 11×1

         0
    0.4190
    0.8485
    1.3365
    1.8671
    3.1091
    4.9807
    7.8518
    9.8297
   11.4578
      ⋮
```

Price an Amortizing Floor Using a Linear Gaussian Two-Factor Model

Define the `ZeroCurve`, `a`, `b`, `sigma`, `eta`, `rho`, and `Notional` parameters for the amortizing floor.

```
Settle = datetime(2007,12,15);
% Define ZeroCurve
ZeroTimes = [3/12 6/12 1 5 7 10 20 30]';
ZeroRates = [0.033 0.034 0.035 0.040 0.042 0.044 0.048 0.0475]';
CurveDates = daysadd(Settle,360*ZeroTimes);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

% Define a, b, sigma, eta, and rho
a = .07;
b = .5;
sigma = .01;
eta = .006;
rho = -.7;

% Define the amortizing floors
FloorMaturity = daysadd(Settle,360*[1:5 7 10 15 20 25 30],1);
Strike = [0.025 0.036 0.037 0.038 0.039 0.041 0.043 0.045 0.046 0.046 0.046]';
Notional = {{datetime(2012,12,15) 100;datetime(2017,12,15) 70;datetime(2022,12,15) 40;datetime(20

% Price the amortizing floors
Price = floorbylg2f(irdc,a,b,sigma,eta,rho,Strike,FloorMaturity,'Notional',Notional)
```

```
Price = 11×1

         0
    0.2776
    0.6630
    1.1062
    1.5938
    2.5589
    3.9582
    5.4985
    6.1113
    6.2670
      ⋮
```

Input Arguments

`ZeroCurve` — Zero curve for Linear Gaussian two-factor model
structure

Zero curve for the Linear Gaussian two-factor model, specified using `IRDataCurve` or `RateSpec`.

Data Types: `struct`

`a` — Mean reversion for first factor for Linear Gaussian two-factor model
scalar

Mean reversion for the first factor for the Linear Gaussian two-factor model, specified as a scalar.

Data Types: `double`

`b` — Mean reversion for second factor for Linear Gaussian two-factor model
scalar

Mean reversion for the second factor for the Linear Gaussian two-factor model, specified as a scalar.

Data Types: `double`

`sigma` — Volatility for first factor for Linear Gaussian two-factor model
scalar

Volatility for the first factor for the Linear Gaussian two-factor model, specified as a scalar.

Data Types: `double`

`eta` — Volatility for second factor for Linear Gaussian two-factor model
scalar

Volatility for the second factor for the Linear Gaussian two-factor model, specified as a scalar.

Data Types: `double`

`rho` — Scalar correlation of factors
scalar

Scalar correlation of the factors, specified as a scalar.

Data Types: `double`

`Strike` — Floor strike price
nonnegative integer | vector of nonnegative integers

Floor strike price specified, as a nonnegative integer using a `NumFloors`-by-`1` vector of floor strike prices.

Data Types: `double`

`Maturity` — Floor maturity date
datetime array | string array | date character vector

Floor maturity date, specified using a `NumFloors`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorbylg2f` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = floorbylg2f(irdc,a,b,sigma,eta,rho,Strike,FloorMaturity,'Reset',1,'Notional',100)`

`Reset` — Frequency of floor payments per year
`2` (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of floor payments per year, specified as the comma-separated pair consisting of `'Reset'` and positive integers for the values `[1,2,4,6,12]` in a `NumFloors`-by-`1` vector.

Data Types: `double`

`Notional` — Notional value of floor
`100` (default) | nonnegative integer | vector of nonnegative integers

`NINST`-by-`1` of notional principal amounts or `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is the associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double`

Output Arguments

`FloorPrice` — Floor price
scalar | vector

Floor price, returned as a scalar or a `NumFloors`-by-`1` vector.

More About

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:

$\max(FloorRate - CurrentRate, 0)$

Algorithms

The following defines the two-factor additive Gaussian interest-rate model, given the `ZeroCurve`, `a`, `b`, `sigma`, `eta`, and `rho` parameters:

$$r(t) = x(t) + y(t) + \phi(t)$$

$$dx(t) = -a(x)(t)dt + \sigma(dW_1(t), x(0) = 0$$

$$dy(t) = -b(y)(t)dt + \eta(dW_2(t), y(0) = 0$$

where $dW_1(t)dW_2(t) = \rho dt$ is a two-dimensional Brownian motion with correlation ρ and ϕ is a function chosen to match the initial zero curve.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floorbylg2f` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Brigo, D. and F. Mercurio, *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

See Also

`capbylg2f` | `swaptionbylg2f` | `LinearGaussian2F` | `Floor`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Floor” on page 2-12
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

floorbynormal

Price floors using Normal or Bachelier pricing model

Syntax

```
[FloorPrice,Floorlets] = floorbynormal(RateSpec,Strike,Settle,Maturity,
Volatility)
[FloorPrice,Floorlets] = floorbynormal( ___ ,Name,Value)
```

Description

`[FloorPrice,Floorlets] = floorbynormal(RateSpec,Strike,Settle,Maturity, Volatility)` prices floors using the Normal (Bachelier) pricing model for negative rates. `floorbynormal` computes prices of vanilla floors and amortizing floors.

Note Alternatively, you can use the `Floor` object to price floor instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[FloorPrice,Floorlets] = floorbynormal( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Floor Using Normal Model for Negative Rates

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 .69394% continuously compounded and the volatility is 20%, use this data to compute the floor price. First, calculate the `RateSpec`, and then use `floorbynormal` to compute the `FloorPrice`.

```
ValuationDate = datetime(2008,1,1);
EndDates = datetime(2010,1,1);
Rates = 0.0069394;
Compounding = -1;
Basis = 1;

% calculate the RateSpec
RateSpec = intenvset('ValuationDate', ValuationDate, ...
'StartDates', ValuationDate,'EndDates', EndDates, ...
'Rates', Rates,'Compounding', Compounding,'Basis', Basis);

Settle = datetime(2009,1,1); % floor starts in a year
Maturity = datetime(2009,4,1);
Volatility = 0.20;
FloorRate = -0.006;
FloorReset = 4;
```

```
Principal=100000;

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

FloorPrice = 1.8212e+03
```

Price a Floor Using `floorbynormal` and Compare to `floorbyblk`

Define the `RateSpec`.

```
Settle = datetime(2016,1,20);
ZeroTimes = [.5 1 2 3 4 5 7 10 20 30]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = datemnth(Settle,12*ZeroTimes);
RateSpec = intenvset('StartDate',Settle,'EndDates',ZeroDates,'Rates',ZeroRates)

RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [10x1 double]
            Rates: [10x1 double]
         EndTimes: [10x1 double]
       StartTimes: [10x1 double]
         EndDates: [10x1 double]
       StartDates: 736349
    ValuationDate: 736349
            Basis: 0
     EndMonthRule: 1
```

Define the floor instrument and price with `floorbyblk`.

```
ExerciseDate = datetime(2026,1,20);

[~,ParSwapRate] = swapbyzero(RateSpec,[NaN 0],Settle,ExerciseDate)

ParSwapRate = 0.0216


Strike = .01;
BlackVol = .3;
NormalVol = BlackVol*ParSwapRate;

Price = floorbyblk(RateSpec,Strike,Settle,ExerciseDate,BlackVol)

Price = 1.2297
```

Price the floor instrument using `floorbynormal`.

```
Price_Normal = floorbynormal(RateSpec,Strike,Settle,ExerciseDate,NormalVol)

Price_Normal = 1.9099
```

Price the floor instrument using `floorbynormal` for a negative strike.

```
 Price_Normal = floorbynormal(RateSpec,-.005,Settle,ExerciseDate,NormalVol)
```

```
Price_Normal = 0.0857
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`Strike` — Rate at which floor is exercised

decimal

Rate at which floor is exercised, specified as a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for floor

datetime array | string array | date character vector

Settlement date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorbynormal` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for floor

datetime array | string array | date character vector

Maturity date for the floor, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorbynormal` also accepts serial date numbers as inputs, but they are not recommended.

`Volatility` — Normal volatilities values

numeric

Normal volatilities values, specified as a `NINST`-by-`1` vector of numeric values.

For more information on the Normal model, see “Work with Negative Interest Rates Using Functions” on page 2-18.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[FloorPrice,Floorlets] = floorbynormal(RateSpec,Strike,Settle,Maturity,Volatility,'Reset',CapReset,'Principal',100000,'Basis',7)`

`Reset` — Reset frequency payment per year
`1` (default) | numeric

Reset frequency payment per year, specified as the comma-separated pair consisting of `'Reset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector or a `NINST`-by-`1` cell array. Each element in the `NINST`-by-`1` cell array is a `NumDates`-by-`2` cell array, where the first column is dates, and the second column is the associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing cap.

Data Types: `double` | `cell`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis of instrument representing the basis used when annualizing the input forward rate, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers. Values are:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`ValuationDate` — Observation date of investment horizon
if `ValuationDate` is not specified, then `Settle` is used (default) | datetime scalar | string scalar | date character vector

Observation date of the investment horizon, specified as the comma-separated pair consisting of `'ValuationDate'` and a scalar datetime, string, or date character vector.

To support existing code, `floorbynormal` also accepts serial date numbers as inputs, but they are not recommended.

`ProjectionCurve` — Rate curve used in generating future cash flows
if `ProjectionCurve` is not specified, then `RateSpec` is used both for discounting cash flows and projecting future cash flows (default) | structure

The rate curve to be used in projecting the future cash flows, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a rate curve structure. This structure must be created using `intenvset`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `struct`

Output Arguments

`FloorPrice` — Expected price of floor
vector

Expected price of the floor, returned as a `NINST`-by-`1` vector.

`Floorlets` — Floorlets
array

Floorlets, returned as a `NINST`-by-`NCF` array of caplets, padded with `NaN`s.

More About

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:

$\max(FloorRate - CurrentRate, 0)$

Version History

Introduced in R2017a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `floorbynormal` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`capbynormal` | `intenvset` | `swaptionbynormal` | `floorbyblk` | `Floor`

Topics

“Calibrating Floorlets Using the Normal (Bachelier) Model” on page 2-159
“Floor” on page 2-12
“Work with Negative Interest Rates Using Functions” on page 2-18
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

External Websites

How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

floorvolstrip

Strip floorlet volatilities from flat floor volatilities

Syntax

```
[FloorletVols,FloorletPaymentDates,FloorStrikes] = floorvolstrip(ZeroCurve,
FloorSettle,FloorMaturity,FloorVolatility)
[FloorletVols,FloorletPaymentDates,FloorStrikes] = floorvolstrip( ___ ,
Name,Value)
```

Description

`[FloorletVols,FloorletPaymentDates,FloorStrikes] = floorvolstrip(ZeroCurve, FloorSettle,FloorMaturity,FloorVolatility)` strips floorlet volatilities from the flat floor volatilities by using the bootstrapping method. The function interpolates the cap volatilities on each floorlet payment date before stripping the floorlet volatilities.

`[FloorletVols,FloorletPaymentDates,FloorStrikes] = floorvolstrip( ___ , Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Stripping Floorlet Volatilities from At-The-Money (ATM) Floors

Compute the zero curve for discounting and projecting forward rates.

```
ValuationDate = datetime(2015,8,10);
ZeroRates = [0.12 0.24 0.40 0.73 1.09 1.62]/100;
CurveDates = datemnth(ValuationDate, [0.25 0.5 1 2 3 5]*12);
ZeroCurve = IRDataCurve('Zero',ValuationDate,CurveDates,ZeroRates)
```

```
ZeroCurve =
             Type: Zero
           Settle: 736186 (10-Aug-2015)
      Compounding: 2
            Basis: 0 (actual/actual)
     InterpMethod: linear
            Dates: [6x1 double]
             Data: [6x1 double]
```

Define the ATM floor volatility data.

```
FloorSettle = datetime(2015,8,12);
FloorMaturity = [datetime(2016,8,12) ; datetime(2017,8,14) ; datetime(2018,8,13) ; datetime(2019
FloorVolatility = [0.31;0.39;0.43;0.42;0.40];
```

Strip floorlet volatilities from ATM floors.

```
[FloorletVols, FloorletPaymentDates, ATMFloorStrikes] = floorvolstrip(ZeroCurve,...
    FloorSettle, FloorMaturity, FloorVolatility);
```

```
PaymentDates = cellstr(datestr(FloorletPaymentDates));
format;
table(PaymentDates, FloorletVols, ATMFloorStrikes)
```

```
ans=9×3 table
     PaymentDates      FloorletVols    ATMFloorStrikes
    _______________    ____________    _______________

    {'12-Aug-2016'}         0.31          0.0056551
    {'13-Feb-2017'}       0.3646          0.0073508
    {'14-Aug-2017'}      0.41948          0.0090028
    {'12-Feb-2018'}      0.43152           0.010827
    {'13-Aug-2018'}      0.46351           0.012617
    {'12-Feb-2019'}      0.40407           0.013862
    {'12-Aug-2019'}      0.39863           0.015105
    {'12-Feb-2020'}       0.3674           0.016369
    {'12-Aug-2020'}      0.35371            0.01762
```

Stripping Floorlet Volatilities from Floors with the Same Strikes

Compute the zero curve for discounting and projecting forward rates.

```
ValuationDate = datetime(2015,6,10);
ZeroRates = [0.02 0.10 0.28 0.75 1.15 1.80]/100;
CurveDates = datemnth(ValuationDate, [0.25 0.5 1 2 3 5]*12);
ZeroCurve = IRDataCurve('Zero',ValuationDate,CurveDates,ZeroRates)
```

```
ZeroCurve =
            Type: Zero
          Settle: 736125 (10-Jun-2015)
     Compounding: 2
           Basis: 0 (actual/actual)
    InterpMethod: linear
           Dates: [6x1 double]
            Data: [6x1 double]
```

Define the floor volatility data.

```
FloorSettle = datetime(2015,6,12);
FloorMaturity = [datetime(2016,6,13) ; datetime(2017,6,12) ; datetime(2018,6,12) ; datetime(2019
FloorVolatility = [0.41;0.43;0.43;0.41;0.38];
FloorStrike = 0.015;
```

Strip floorlet volatilities from floors with the same strike.

```
[FloorletVols, FloorletPaymentDates, FloorStrikes] = floorvolstrip(ZeroCurve, ...
    FloorSettle, FloorMaturity, FloorVolatility, 'Strike', FloorStrike);

PaymentDates = cellstr(datestr(FloorletPaymentDates));
format;
table(PaymentDates, FloorletVols, FloorStrikes)
```

```
ans=9×3 table
     PaymentDates      FloorletVols    FloorStrikes
```

```
    ________________    ____________    ____________

    {'13-Jun-2016'}         0.41           0.015
    {'12-Dec-2016'}         0.42           0.015
    {'12-Jun-2017'}      0.43433           0.015
    {'12-Dec-2017'}      0.43001           0.015
    {'12-Jun-2018'}         0.43           0.015
    {'12-Dec-2018'}      0.39173           0.015
    {'12-Jun-2019'}      0.37244           0.015
    {'12-Dec-2019'}      0.32056           0.015
    {'12-Jun-2020'}      0.28308           0.015
```

Stripping Floorlet Volatilities Using Manually Specified Floorlet Dates

Compute the zero curve for discounting and projecting forward rates.

```
ValuationDate = datetime(2015,5,19);
ZeroRates = [0.02 0.07 0.23 0.63 1.01 1.60]/100;
CurveDates = datemnth(ValuationDate, [0.25 0.5 1 2 3 5]*12);
ZeroCurve = IRDataCurve('Zero',ValuationDate,CurveDates,ZeroRates)
```

```
ZeroCurve =
            Type: Zero
          Settle: 736103 (19-May-2015)
    Compounding: 2
           Basis: 0 (actual/actual)
    InterpMethod: linear
           Dates: [6x1 double]
            Data: [6x1 double]
```

Define the floor volatility data.

```
FloorSettle = datetime(2015,5,19);
FloorMaturity = [datetime(2016,5,19) ; datetime(2017,5,19) ; datetime(2018,5,21) ; datetime(2019
FloorVolatility = [0.39;0.42;0.43;0.42;0.40];
FloorStrike = 0.010;
```

Specify the quarterly and semiannual dates.

```
FloorletDates = [cfdates(FloorSettle, datetime(2016,5,19), 4)...
     cfdates(datetime(2016,5,19),datetime(2020,5,19), 2)]';
FloorletDates(~isbusday(FloorletDates)) = ...
    busdate(FloorletDates(~isbusday(FloorletDates)), 'modifiedfollow');
```

Strip floorlet volatilities using specified `FloorletDates`.

```
[FloorletVols, FloorletPaymentDates, FloorStrikes] = floorvolstrip(ZeroCurve, ...
    FloorSettle, FloorMaturity, FloorVolatility, 'Strike', FloorStrike, ...
    'FloorletDates', FloorletDates);

PaymentDates = cellstr(datestr(FloorletPaymentDates));
format;
table(PaymentDates, FloorletVols, FloorStrikes)
```

```
ans=11×3 table
     PaymentDates      FloorletVols    FloorStrikes
    _______________    ____________    ____________

    {'19-Nov-2015'}         0.39           0.01
    {'19-Feb-2016'}         0.39           0.01
    {'19-May-2016'}         0.39           0.01
    {'21-Nov-2016'}       0.4058           0.01
    {'19-May-2017'}       0.4307           0.01
    {'20-Nov-2017'}      0.43317           0.01
    {'21-May-2018'}      0.44309           0.01
    {'19-Nov-2018'}      0.40831           0.01
    {'20-May-2019'}      0.39831           0.01
    {'19-Nov-2019'}       0.3524           0.01
    {'19-May-2020'}      0.32765           0.01
```

Stripping Floorlet Volatilities from Floors Using the Shifted Black Model

Compute the zero curve for discounting and projecting forward rates.

```
ValuationDate = datetime(2016,5,3);
ZeroRates = [-0.31 -0.21 -0.15 -0.10 0.009 0.19]/100;
CurveDates = datemnth(ValuationDate, [0.25 0.5 1 2 3 5]*12);
ZeroCurve = IRDataCurve('Zero',ValuationDate,CurveDates,ZeroRates)
```

```
ZeroCurve =
            Type: Zero
          Settle: 736453 (03-May-2016)
     Compounding: 2
           Basis: 0 (actual/actual)
    InterpMethod: linear
           Dates: [6x1 double]
            Data: [6x1 double]
```

Define the floor volatility (Shifted Black) data.

```
FloorSettle = datetime(2016,5,3);
FloorMaturity = [datetime(2017,5,3) ; datetime(2018,5,3) ; datetime(2019,5,3) ; datetime(2020,5,
FloorVolatility = [0.42;0.45;0.43;0.40;0.36]; % Shifted Black volatilities
Shift = 0.01; % 1 percent shift.
FloorStrike = -0.001; % -0.1 percent strike.
```

Strip floorlet volatilities from floors using the Shifted Black Model.

```
[FloorletVols, FloorletPaymentDates, FloorStrikes] = floorvolstrip(ZeroCurve, ...
FloorSettle,FloorMaturity,FloorVolatility,'Strike',FloorStrike,'Shift',Shift);

PaymentDates = string(datestr(FloorletPaymentDates));
format;
table(PaymentDates,FloorletVols,FloorStrikes)
```

```
ans=9×3 table
    PaymentDates     FloorletVols    FloorStrikes
    ____________     ____________    ____________
```

```
    "03-May-2017"         0.42          -0.001
    "03-Nov-2017"      0.44575          -0.001
    "03-May-2018"      0.47092          -0.001
    "05-Nov-2018"      0.41911          -0.001
    "03-May-2019"      0.40197          -0.001
    "04-Nov-2019"      0.36262          -0.001
    "04-May-2020"      0.33615          -0.001
    "03-Nov-2020"      0.27453          -0.001
    "03-May-2021"      0.23045          -0.001
```

Stripping Floorlet Volatilities from Floors Using Normal Model

Compute the zero curve for discounting and projecting forward rates.

```
ValuationDate = datetime(2018,5,1);
ZeroRates = [-0.31 -0.27 -0.18 -0.05 0.015 0.22]/100;
CurveDates = datemnth(ValuationDate, [0.25 0.5 1 2 3 5]*12);
ZeroCurve = IRDataCurve('Zero',ValuationDate,CurveDates,ZeroRates)
```

```
ZeroCurve =
             Type: Zero
           Settle: 737181 (01-May-2018)
      Compounding: 2
            Basis: 0 (actual/actual)
     InterpMethod: linear
            Dates: [6x1 double]
             Data: [6x1 double]
```

Define the normal floor volatility data.

```
FloorSettle = datetime(2018,5,1);
FloorMaturity = [datetime(2019,5,1) ; datetime(2020,5,1) ; datetime(2021,5,3) ; datetime(2022,5,2
FloorVolatility = [0.0065;0.0067;0.0064;0.0058;0.0055]; % Normal volatilities
FloorStrike = -0.005; % -0.5 percent strike.
```

Strip floorlet volatilities from floors using the Normal (Bachelier) model.

```
[FloorletVols, FloorletPaymentDates, FloorStrikes] = floorvolstrip(ZeroCurve, ...
    FloorSettle,FloorMaturity,FloorVolatility,'Strike',FloorStrike,'Model','normal');

PaymentDates = string(datestr(FloorletPaymentDates));
format;
table(PaymentDates,FloorletVols,FloorStrikes)
```

```
ans=9×3 table
    PaymentDates     FloorletVols    FloorStrikes
    ____________     ____________    ____________

    "01-May-2019"        0.0065         -0.005
    "01-Nov-2019"     0.0066644         -0.005
    "01-May-2020"     0.0068354         -0.005
    "02-Nov-2020"      0.006266         -0.005
    "03-May-2021"     0.0060101         -0.005
```

```
"01-Nov-2021"      0.004942         -0.005
"02-May-2022"     0.0042668         -0.005
"01-Nov-2022"     0.0047986         -0.005
"01-May-2023"     0.0044738         -0.005
```

Input Arguments

`ZeroCurve` — Zero rate curve

`RateSpec` object | `IRDataCurve` object

Zero rate curve, specified using a `RateSpec` or `IRDataCurve` object containing the zero rate curve for discounting according to its day count convention. If you do not specify the optional argument `ProjectionCurve`, the function uses `ZeroCurve` to compute the underlying forward rates as well. The observation date of the `ZeroCurve` specifies the valuation date. For more information on creating a `RateSpec`, see `intenvset`. For more information on creating an `IRDataCurve` object, see `IRDataCurve`.

Data Types: `struct`

`FloorSettle` — Common floor settle date

datetime scalar | string scalar | date character vector

Common floor settle date, specified as a scalar datetime, string, or date character vector. The `FloorSettle` date cannot be earlier than the `ZeroCurve` valuation date.

To support existing code, `floorvolstrip` also accepts serial date numbers as inputs, but they are not recommended.

`FloorMaturity` — Floor maturity dates

datetime array | string array | date character vector

Floor maturity dates, specified as an `NFloor`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorvolstrip` also accepts serial date numbers as inputs, but they are not recommended.

`FloorVolatility` — Flat floor volatilities

vector of positive decimals

Flat floor volatilities, specified as an `NFloor`-by-`1` vector of positive decimals.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[FloorletVols,FloorletPaymentDates,FloorStrikes] = floorvolstrip(ZeroCurve,FloorSettle,FloorMaturity,FloorVolatility,'Strike',.2)`

Strike — Floor strike rate
if not specified, all floors are at-the-money and the function computes the ATM strike for each floor maturing on each floorlet payment date (default) | scalar decimal | vector

Floor strike rate, specified as the comma-separated pair consisting of `'Strike'` and a scalar decimal value or an `NFloorletVols`-by-`1` vector. Use `Strike` as a scalar to specify a single strike that applies equally to all floors. Or, specify an `NCapletVols`-by-`1` vector of strikes for the floors.

Data Types: `double`

FloorletDates — Floorlet reset and payment dates
If not specified, the default is to automatically generate periodic floorlet dates (default) | datetime array | string array | date character vector

Floorlet reset and payment dates, specified as the comma-separated pair consisting of `'FloorletDates'` and an `NFloorletDates`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floorvolstrip` also accepts serial date numbers as inputs, but they are not recommended.

Use `FloorletDates` to manually specify all floorlet reset and payment dates. For example, some date intervals may be quarterly while others may be semiannual. All dates must be later than `FloorSettle` and cannot be later than the last `FloorMaturity` date. Dates are adjusted according to the `BusDayConvention` and `Holidays` inputs.

If `FloorletDates` is not specified, the default is to automatically generate periodic floorlet dates after `FloorSettle` based on the last `FloorMaturity` date as the reference date, using the following optional inputs: `Reset`, `EndMonthRule`, `BusDayConvention`, and `Holidays`.

Reset — Frequency of periodic payments per year within a floor
2 (default) | positive scalar integer with values 1,2, 3, 4, 6, or 12

Frequency of periodic payments per year within a floor, specified as the comma-separated pair consisting of `'Reset'` and a positive scalar integer with values 1,2, 3, 4, 6, or 12.

Note If you specify `FloorletDates`, the function ignores the input for `Reset`.

Data Types: `double`

EndMonthRule — End-of-month rule flag for generating floorlet dates
1 (in effect) (default) | scalar nonnegative integer `[0,1]`

End-of-month rule flag for generating floorlet dates, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`].

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

BusinessDayConvention — Business day conventions
`'modifiedfollow'` (default) | character vector with values `'actual'`, `'follow'`, `'modifiedfollow'`, `'previous'`, `'modifiedprevious'`

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector. Use this argument to specify how the function treats non-business days, which are days on which businesses are not open (such as weekends and statutory holidays).

- `'actual'` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `'follow'` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `'modifiedfollow'` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- `'previous'` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `'modifiedprevious'` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | vector of MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and `NHolidays`-by-`1` vector of MATLAB dates.

Data Types: `datetime`

`ProjectionCurve` — Rate curve for computing underlying forward rates
if not specified, the default is to use the `ZeroCurve` input for computing the underlying forward rates (default) | `RateSpec` object | `IRDatCurve` object

Rate curve for computing underlying forward rates, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a `RateSpec` object or `IRDatCurve` object. For more information on creating a `RateSpec`, see `intenvset`. For more information on creating an `IRDataCurve` object, see `IRDataCurve`.

Data Types: `struct`

`MaturityInterpMethod` — Method for interpolating floor volatilities on each floorlet maturity date before stripping floorlet volatilities
`'linear'` (default) | character vector with values: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'spline'`, `'pchip'`

Method for interpolating the floor volatilities on each floorlet maturity date before stripping the floorlet volatilities, specified as the comma-separated pair consisting of `'MaturityInterpMethod'` and a character vector with values: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'spline'`, or `'pchip'`.

- `'linear'` — Linear interpolation. The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.
- `'nearest'` — Nearest neighbor interpolation. The interpolated value at a query point is the value at the nearest sample grid point.

- `'next'` — Next neighbor interpolation. The interpolated value at a query point is the value at the next sample grid point.
- `'previous'` — Previous neighbor interpolation. The interpolated value at a query point is the value at the previous sample grid point.
- `'spline'` — Spline interpolation using not-a-knot end conditions. The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension.
- `'pchip'` — Shape-preserving piecewise cubic interpolation. The interpolated value at a query point is based on a shape-preserving piecewise cubic interpolation of the values at neighboring grid points.

For more information on interpolation methods, see `interp1`.

Note The function uses constant extrapolation to calculate volatilities falling outside the range of user-supplied data.

Data Types: `char`

`Limit` — Upper bound of implied volatility search interval
`10` (or 1000% per annum) (default) | positive scalar decimal

Upper bound of implied volatility search interval, specified as the comma-separated pair consisting of `'Limit'` and a positive scalar decimal.

Data Types: `double`

`Tolerance` — Implied volatility search termination tolerance
`1e-5` (default) | positive scalar

Implied volatility search termination tolerance, specified as the comma-separated pair consisting of `'Tolerance'` and a positive scalar.

Data Types: `double`

`OmitFirstFloorlet` — Flag to omit the first floorlet payment in the floors
`true` always omit the first floorlet (default) | logical

Flag to omit the first floorlet payment in the floors, specified as the comma-separated pair consisting of `'OmitFirstFloorlet'` and a scalar logical.

If the floors are spot-starting, the first floorlet payment is omitted. If the floors are forward-starting, the first floorlet payment is included. Regardless of the status of the floors, if you set this logical to `false`, then the function includes the first floorlet payment.

In general, “spot lag” is the delay between the fixing date and the effective date for LIBOR-like indices. "Spot lag" determines whether a floor is spot-starting or forward-starting (Corb, 2012). Floors are considered to be spot-starting if they settle within “spot lag” business days after the valuation date. Those that settle later are considered to be forward-starting. The first floorlet is omitted if floors are spot-starting, while it is included if they are forward-starting (Tuckman, 2012).

Data Types: `logical`

`Shift` — Shift in decimals for shifted SABR model
`0` (no shift) (default) | positive scalar decimal

Shift in decimals for the shifted SABR model (to be used with the Shifted Black model), specified as the comma-separated pair consisting of `'Shift'` and a positive scalar decimal value. Set this parameter to a positive shift in decimals to add a positive shift to the forward rate and strike, which effectively sets a negative lower bound for the forward rate and strike. For example, a `Shift` value of 0.01 is equal to a 1% shift.

Data Types: `double`

`Model` — Model used for implied volatility
`'lognormal'` (default) | character vector with value of `'lognormal'` or `'normal'` | string scalar with value of `"lognormal"` or `"normal"`

Model used for the implied volatility calculation, specified as the comma-separated pair consisting of `'Model'` and a scalar character vector or string scalar with one of the following values:

- `'lognormal'` - Implied Black (no shift) or Shifted Black volatility.
- `'normal'` - Implied Normal (Bachelier) volatility. If you specify `'normal'`, `Shift` must be zero.

The `floorvolstrip` function supports three volatility types.

'Model' Value	'Shift' Value	Volatility Type
`'lognormal'`	`Shift = 0`	Black
`'lognormal'`	`Shift > 0`	Shifted Black
`'normal'`	`Shift = 0`	Normal (Bachelier)

Data Types: `char` | `string`

Output Arguments

`FloorletVols` — Stripped floorlet volatilities
vector of decimals

Stripped floorlet volatilities, returned as a `NFloorletVols`-by-`1` vector of decimals.

Note `floorvolstrip` can output `NaN`s for some caplet volatilities. You might encounter this output if no volatility matches the caplet price implied by the user-supplied cap data.

`FloorletPaymentDates` — Payment dates
vector of date numbers

Payment dates (in date numbers), returned as an `NFloorletVols`-by-`1` vector of date numbers corresponding to `FloorletVols`.

`FloorStrikes` — Floor strikes
vector of decimals

Floor strikes, returned as a `NFloorletVols`-by-`1` vector of strikes in decimals for floors maturing on the corresponding `FloorletPaymentDates`.

Limitations

When bootstrapping the floorlet volatilities from ATM floors, the function reuses the floorlet volatilities stripped from the shorter maturity floors in the longer maturity floors without adjusting for the difference in strike. `floorvolstrip` follows the simplified approach described in Gatarek, 2006.

More About

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:

$\max(FloorRate - CurrentRate, 0)$

At-The-Money

A cap or floor is at-the-money (ATM) if its strike is equal to the forward swap rate.

The forward swap rate is the fixed rate of a swap that makes the present value of the floating leg equal to that of the fixed leg. In comparison, a caplet or floorlet is ATM if its strike is equal to the forward rate (not the forward swap rate). In general (except over a single period), the forward rate is not equal to the forward swap rate. So, to be precise, the individual caplets in an ATM cap have slightly different moneyness and are only approximately ATM (Alexander, 2003).

In addition, the swap rate changes with swap maturity. Similarly, the ATM cap strike also changes with cap maturity, so the ATM cap strikes are computed for each cap maturity before stripping the caplet volatilities. As a result, when stripping the caplet volatilities from the ATM caps with increasing maturities, the ATM strikes of consecutive caps are different.

Version History

Introduced in R2016a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `floorvolstrip` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Alexander, C. "Common Correlation and Calibrating the Lognormal Forward Rate Model." *Wilmott Magazine*, 2003.

[2] Corb, H. *Interest Rate Swaps and Other Derivatives.* Columbia Business School Publishing, 2012.

[3] Gatarek, D., P. Bachert, and R. Maksymiuk. *The LIBOR Market Model in Practice.* Chichester, UK: Wiley, 2006.

[4] Tuckman, B., and Serrat, A. *Fixed Income Securities: Tools for Today's Markets.* Hoboken, NJ: Wiley, 2012.

See Also

`interp1` | `intenvset` | `capvolstrip` | `floorbyblk` | `floorbynormal`

Topics

"Price Swaptions with Negative Strikes Using the Shifted SABR Model" on page 2-26
"Floor" on page 2-12
"Work with Negative Interest Rates Using Functions" on page 2-18

External Websites

How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

gapbybls

Determine price of gap digital options using Black-Scholes model

Syntax

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

Description

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

Examples

Compute Gap Option Prices Using the Black-Scholes Option Pricing Model

This example shows how to compute gap option prices using the Black-Scholes option pricing model. 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%.

```
Settle = datetime(2008,1,1);
Maturity = datetime(2008,7,1);
Compounding = -1;
Rates = 0.09;
% calculate the RateSpec
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 = 2×1

   -0.0053
    4.4866
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `gapbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `gapbybls` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as an `NINST`-by-`1` vector.

Data Types: `char` | `cell`

`Strike` — Pay-off strike value

vector

Pay-off strike value, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`StrikeThreshold` — Strike values that determine if the option pays off

vector

Strike values that determine if the option pays off, specified as an `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected prices for gap option

vector

Expected prices for gap option, returned as a `NINST`-by-`1` vector.

More About

Gap Option

A gap option is 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.

Version History

Introduced in R2009a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `gapbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`assetbybls` | `cashbybls` | `gapsensbybls` | `supersharebybls`

Topics

“Pricing Using the Black-Scholes Model” on page 3-82
“Digital Option” on page 3-26
“Supported Equity Derivative Functions” on page 3-19

gapsensbybls

Determine price or sensitivities of gap digital options using Black-Scholes model

Syntax

```
PriceSens = gapsensbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,
StrikeThreshold)
PriceSens = gapsensbybls( ___ ,Name,Value)
```

Description

`PriceSens = gapsensbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike, StrikeThreshold)` calculates gap European digital option prices or sensitivities using the Black-Scholes option pricing model.

`PriceSens = gapsensbybls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute Gap Option Prices and Sensitivities Using the Black-Scholes Option Pricing Model

This example shows how to compute gap option prices and sensitivities using the Black-Scholes option pricing model. 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%.

```
Settle = datetime(2008,1,1);
Maturity = datetime(2008,7,1);
Compounding = -1;
Rates = 0.09;
%create the RateSpec
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;
% compute the price
Pgap = gapbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec,...
Strike, StrikeThreshold)
```

```
Pgap = 2×1

   -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 = 2×1

    0.0724
    0.0724


Delta = 2×1

    0.2852
   -0.7148
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `gapsensbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `gapsensbybls` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as an `NINST`-by-`1` vector.

Data Types: `char` | `cell`

`Strike` — Pay-off strike value
vector

Pay-off strike value, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`StrikeThreshold` — Strike values that determine if the option pays off
vector

Strike values that determine if the option pays off, specified as an `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Gamma,Delta] = gapsensbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,StrikeThreshold,'OutSpec',{'gamma'; 'delta'})`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities for gap option
vector

Expected prices or sensitivities (defined using `OutSpec`) for gap option, returned as a `NINST`-by-`1` vector.

More About

Gap Option

A gap option is 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.

Version History

Introduced in R2009a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `gapsensbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`gapbybls`

Topics

“Pricing Using the Black-Scholes Model” on page 3-82
“Digital Option” on page 3-26
“Supported Equity Derivative Functions” on page 3-19

hedgeopt

Allocate optimal hedge for target costs or sensitivities

Syntax

```
[PortSens,PortCost,PortHolds] = hedgeopt(Sensitivities,Price,CurrentHolds)
[PortSens,PortCost,PortHolds] = hedgeopt( ___ ,FixedInd,NumCosts,TargetCost,
TargetSensConSet)
```

Description

`[PortSens,PortCost,PortHolds] = hedgeopt(Sensitivities,Price,CurrentHolds)` 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,PortCost,PortHolds] = hedgeopt( ___ ,FixedInd,NumCosts,TargetCost, TargetSensConSet)` adds additional optional arguments.

Examples

Allocate Optimal Hedge for Target Costs

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.

In this examples, portfolio target sensitivities are treated as equality constraints during the optimization process. You can use `hedgeopt` to specify what sensitivities you want, and `hedgeopt` computes what what it will cost to get those sensitivities.

```
load deriv.mat;
```

Compute the price and sensitivities

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

```
Delta = 8×1

 -272.6462
 -347.4315
   -8.0781
 -272.6462
```

```
   -1.0445
  294.9700
  -47.1629
 -282.0465


Gamma = 8×1
10^3 ×

    1.0299
    1.6227
    0.6434
    1.0299
    0.0033
    6.8526
    8.4600
    1.0597


Vega = 8×1

    0.0000
   -0.0397
   34.0746
    0.0000
         0
   93.6946
   93.6946
    0.0000


Price = 8×1

   98.7159
   97.5280
    0.0486
   98.7159
  100.5529
    6.2831
    0.0486
    3.6923
```

Extract the current portfolio holdings.

```
warning('on')
Holdings = instget(HJMInstSet, 'FieldName', 'Quantity')

Holdings = 8×1

   100
    50
   -50
    80
     8
    30
    40
    10
```

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.

Summarize the portfolio information.

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

```
   1.0e+03 *

    0.0987    0.1000   -0.2726    1.0299    0.0000
    0.0975    0.0500   -0.3474    1.6227   -0.0000
    0.0000   -0.0500   -0.0081    0.6434    0.0341
    0.0987    0.0800   -0.2726    1.0299    0.0000
    0.1006    0.0080   -0.0010    0.0033         0
    0.0063    0.0300    0.2950    6.8526    0.0937
    0.0000    0.0400   -0.0472    8.4600    0.0937
    0.0037    0.0100   -0.2820    1.0597    0.0000
```

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 = 1×3
10^5 ×

   -0.6191    7.8895    0.0485
```

Maintaining Existing Allocations

To illustrate using `hedgeopt`, suppose that you want to maintain your existing portfolio. `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.

```
FixedInd = [1 2 3 4 5 6 7 8];
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,Holdings, FixedInd, [], [], TargetSens)
```

```
Sens = 1×3
10^5 ×

   -0.6191    7.8895    0.0485


Cost = 0

Quantity = 1×8

   100    50   -50    80     8    30    40    10
```

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 = 2.3675e+04
```

```
Value1 = Quantity * Price
```

```
Value1 = 2.3675e+04
```

Partially Hedged Portfolio

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

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

Specify the instruments to be fixed.

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

Use `hedgeopt`:

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

```
Sens = 1×3
10^4 ×

   -2.3000   -0.3300    0.3000


Cost = 1.9174e+04

Quantity = 1×8

  100.0000 -141.0267  137.2638   80.0000    8.0000  -57.9606   40.0000   10.0000
```

Recompute `Value1`, the portfolio value after rebalancing.

```
Value1 = Quantity * Price
```

```
Value1 = 4.5006e+03
```

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 are changed.

Fully Hedged Portfolio

The example has illustrated 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 `0`s and call `hedgeopt` again.

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

```
Sens = 1×3
10^-10 ×

    0.1091    0.5821    0.0045


Cost = 2.3056e+04

Quantity = 1×8

  100.0000 -182.3615  -19.5501   80.0000    8.0000  -32.9674   40.0000   10.0000
```

Examining the outputs reveals that you have obtained a fully hedged portfolio but at an expense of over $20,000 and `Quantity` defines the positions required to achieve a fully hedged portfolio.

The resulting new portfolio value is

```
Value1 = Quantity * Price
```

```
Value1 = 618.7168
```

Input Arguments

`Sensitivities` — Sensitivities of each instrument

matrix

Sensitivities of each instrument, specified as a number of instruments (`NINST`) by number of sensitivities (`NSENS`) matrix of dollar sensitivities. Each row represents a different instrument. Each column represents a different sensitivity.

Data Types: `double`

`Price` — Instrument prices

vector

Instrument prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`CurrentHolds` — Contracts allocated to each instrument

vector

Contracts allocated to each instrument, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`FixedInd` — Number of fixed instruments

`[ ]` (default) | vector

(Optional) Number of fixed instruments, specified as an `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.

Data Types: `double`

NumCosts — Number of points generated along the cost frontier
10 (default) | integer

(Optional) Number of points generated along the cost frontier when a vector of target costs (TargetCost) is not defined, specified as an integer. 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 [].

Data Types: double

TargetCost — Target cost values along the cost frontier
[ ] (default) | vector

(Optional) Target cost values along the cost frontier, specified as a vector. 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.

Data Types: double

TargetSens — Target sensitivity values of the portfolio
[ ] (default) | vector

(Optional) Target sensitivity values of the portfolio, specified as a 1-by-NSENS vector containing the target sensitivity values of the portfolio. When specifying TargetSens, enter NumCosts and TargetCost as empty matrices [].

Data Types: double

ConSet — Additional conditions on the portfolio reallocations
[ ] (default) | matrix

(Optional) Additional conditions on the portfolio reallocations, specified as a 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).

Note 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, PortHolds, and the elements of PortCost are padded with NaNs as placeholders.

Data Types: double

Output Arguments

`PortSens` — Portfolio dollar sensitivities

matrix

Portfolio dollar sensitivities, returned as a number of points (`NPOINTS`-by-`NSENS`) matrix. When a perfect hedge exists, `PortSens` is zeros. Otherwise, the best hedge possible is chosen.

Note Not all problems are solvable (for example, the solution space may be infeasible, unbounded, or insufficiently constrained), or the solution may fail to converge. When a valid solution is not found, the corresponding rows of `PortSens` and `PortHolds` and elements of `PortCost` are padded with `NaN`'s as placeholders. In addition, the solution may not be unique.

`PortCost` — Total portfolio costs

vector

Total portfolio costs, returned as a `1`-by-`NPOINTS` vector.

`PortHolds` — Contracts allocated to each instrument

matrix

Contracts allocated to each instrument, returned as an `NPOINTS`-by-`NINST` matrix. These are the reallocated portfolios.

Version History

Introduced before R2006a

See Also

`hedgeslf` | `pcalims` | `portcons` | `portopt` | `lsqlin`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Hedging with hedgeopt” on page 4-4
“Instrument Constructors” on page 1-15
“Hedging” on page 4-2
“Supported Equity Derivative Functions” on page 3-19

hedgeslf

Self-financing hedge

Syntax

```
[PortSens,PortValue,PortHolds] = hedgeslf(Sensitivities,Price,CurrentHolds)
[PortSens,PortValue,PortHolds] = hedgeslf( ___ ,FixedInd,ConSet)
```

Description

`[PortSens,PortValue,PortHolds] = hedgeslf(Sensitivities,Price,CurrentHolds)` 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.

`hedgeslf` attempts to find the allocations of the portfolio that will make it closest to being self-financing, while reducing the sensitivities to zero. If no solution is found, `hedgeslf` finds the allocations that will minimize the sensitivities. If the resulting portfolio is self-financing, `PortValue` is equal to the value of the original portfolio.

`[PortSens,PortValue,PortHolds] = hedgeslf( ___ ,FixedInd,ConSet)` adds additional optional arguments.

Examples

Pricing and Hedging a Portfolio Using the Black-Karasinski Model

This example illustrates how MATLAB® can be used to create a portfolio of interest-rate derivatives securities, and price it using the Black-Karasinski interest-rate model. The example also shows some hedging strategies to minimize exposure to market movements.

Create the Interest-Rate Term Structure Based on Reported Data

The structure `RateSpec` is an interest-rate term structure that defines the initial rate specification from which the tree rates are derived. Use the information of annualized zero coupon rates in the table below to populate the `RateSpec` structure.

```
  From             To           Rate
27 Feb 2007    27 Feb 2008      0.0493
27 Feb 2007    27 Feb 2009      0.0459
27 Feb 2007    27 Feb 2010      0.0450
27 Feb 2007    27 Feb 2012      0.0446
27 Feb 2007    27 Feb 2014      0.0445
27 Feb 2007    27 Feb 2017      0.0450
27 Feb 2007    27 Feb 2027      0.0473
```

This data could be retrieved from the Federal Reserve Statistical Release page by using the Datafeed Toolbox™. In this case, the Datafeed Toolbox™ will connect to FRED® and pull back the rates of the following treasury notes.

```
 Terms    Symbol
=======   ======
   1   =  DGS1
   2   =  DGS2
   3   =  DGS3
   5   =  DGS5
   7   =  DGS7
   10  =  DGS10
   20  =  DGS20
```

Create the connection object:

```
c = fred;
```

Create the symbol fetch list:

```
FredNames   = { ...
  'DGS1'; ...      % 1  Year
  'DGS2'; ...      % 2  Year
  'DGS3'; ...      % 3  Year
  'DGS5'; ...      % 5  Year
  'DGS7'; ...      % 7  Year
  'DGS10'; ...     % 10 Year
  'DGS20'};        % 20 Year
```

Define the Terms:

```
Terms = [ 1; ...      % 1  Year
          2; ...      % 2  Year
          3; ...      % 3  Year
          5; ...      % 5  Year
          7; ...      % 7  Year
         10; ...      % 10 Year
         20];         % 20 Year
```

Set the `StartDate` to Feb 27, 2007:

```
StartDate = datenum('Feb-27-2007');

FredRet = fetch(c,FredNames,StartDate);
```

Set the `ValuationDate` based on the `StartDate`:

```
ValuationDate = StartDate;

EndDates = [];

Rates =[];
```

Create the `EndDates`:

```
for idx = 1:length(FredRet)

 %Pull the rates associated with Feb 27, 2007. All the Fred Rates come
 %back as percents
 Rates = [Rates; ...
     FredRet(idx).Data(1,2) / 100];

  %Determine the EndDates by adding the Term to the year of the
  %StartDate
```

```
    EndDates = [EndDates; ...
       round(datenum(...
           year(StartDate)+ Terms(idx,1), ...
           month(StartDate),...
           day(StartDate)))];

  end
```

Use the function `intenvset` to create the `RateSpec` with the following data:

```
Compounding = 1;
StartDate = datetime(2007,2,27);
Rates = [0.0493; 0.0459; 0.0450; 0.0446; 0.0446; 0.0450; 0.0473];
EndDates = [datetime(2008,2,27); datetime(2009,2,27) ; datetime(2010,2,27) ; datetime(2012,2,27)
ValuationDate = StartDate;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [7x1 double]
            Rates: [7x1 double]
          EndTimes: [7x1 double]
        StartTimes: [7x1 double]
          EndDates: [7x1 double]
       StartDates: 733100
    ValuationDate: 733100
            Basis: 0
     EndMonthRule: 1
```

Specify the Volatility Model

Create the structure `VolSpec` that specifies the volatility process with the following data.

```
Volatility = [0.011892; 0.01563; 0.02021; 0.02125; 0.02165; 0.02065; 0.01803];
Alpha = [0.0001];
VolSpec = bkvolspec(ValuationDate, EndDates, Volatility, EndDates(end), Alpha)
```

```
VolSpec = struct with fields:
             FinObj: 'BKVolSpec'
      ValuationDate: 733100
            VolDates: [7x1 double]
            VolCurve: [7x1 double]
          AlphaCurve: 1.0000e-04
          AlphaDates: 740405
    VolInterpMethod: 'linear'
```

Specify the Time Structure of the Tree

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 = bktimespec(ValuationDate, EndDates)
```

```
TimeSpec = struct with fields:
           FinObj: 'BKTimeSpec'
```

```
    ValuationDate: 733100
         Maturity: [7x1 double]
      Compounding: -1
            Basis: 0
     EndMonthRule: 1
```

Create the BK Tree

Use the previously computed values for `RateSpec`, `VolSpec`, and `TimeSpec` to create the BK tree.

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

```
BKTree = struct with fields:
      FinObj: 'BKFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 5 7 10]
        dObs: [733100 733465 733831 734196 734926 735657 736753]
      CFlowT: {1x7 cell}
       Probs: {1x6 cell}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]  [2 3 3 4 5 5 6]  [2 3 4 5 6 7 8]  [2 2 ... ]}
     FwdTree: {1x7 cell}
```

Visualize the interest rate evolution along the tree by looking at the output structure `BKTree`. The function `bktree` returns an inverse discount tree, which you can convert into an interest rate tree with the `cvtree` function.

```
BKTreeR = cvtree(BKTree)
```

```
BKTreeR = struct with fields:
      FinObj: 'BKRateTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 5 7 10]
        dObs: [733100 733465 733831 734196 734926 735657 736753]
      CFlowT: {1x7 cell}
       Probs: {1x6 cell}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]  [2 3 3 4 5 5 6]  [2 3 4 5 6 7 8]  [2 2 ... ]}
    RateTree: {1x7 cell}
```

Look at the upper, middle and lower branch paths of the tree.

```
OldFormat = get(0, 'format');
format short

%Rate at root node:
RateRoot      = trintreepath(BKTreeR, 0)
```

```
RateRoot = 0.0481
```

```
%Rates along upper branch:
RatePathUp    = trintreepath(BKTreeR, [1 1 1 1 1 1])
```

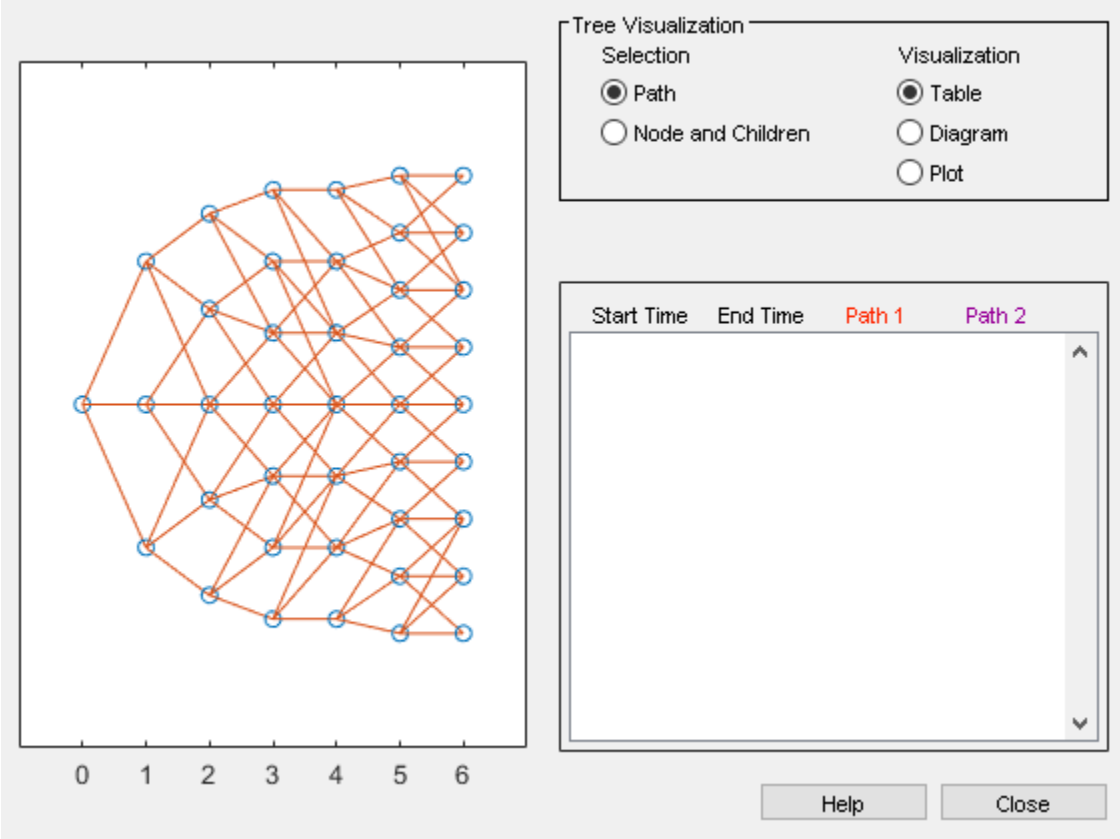
```
RatePathUp = 7×1

    0.0481
    0.0425
    0.0446
    0.0478
    0.0510
    0.0555
    0.0620
```

```
%Rates along middle branch:
RatePathMiddle = trintreepath(BKTreeR, [2 2 2 2 2 2])

RatePathMiddle = 7×1

    0.0481
    0.0416
    0.0423
    0.0430
    0.0436
    0.0449
    0.0484
```

```
%Rates along lower branch:
RatePathDown = trintreepath(BKTreeR, [3 3 3 3 3 3])

RatePathDown = 7×1

    0.0481
    0.0408
    0.0401
    0.0388
    0.0373
    0.0363
    0.0378
```

You can also display a graphical representation of the tree to examine interactively the rates on the nodes of the tree until maturity. The function `treeviewer` displays the structure of the rate tree in the left window. The tree visualization in the right window is blank, but by selecting **Table/Diagram** and clicking on the nodes you can examine the rates along the paths.

```
treeviewer(BKTreeR);
```



Create an Instrument Portfolio

Create a portfolio consisting of two bonds instruments and an option on the 5% bond.

```
% Two Bonds
CouponRate = [0.04;0.05];
Settle = datetime(2007,2,27);
Maturity = [datetime(2009,2,27) ; datetime(2010,2,27) ];
Period = 1;

% American Option on the 5% Bond
OptSpec = {'call'};
Strike = 98;
ExerciseDates = datetime(2010,2,27);
AmericanOpt = 1;

InstSet = instadd('Bond', CouponRate, Settle,  Maturity, Period);
InstSet = instadd(InstSet,'OptBond', 2, OptSpec, Strike, ExerciseDates, AmericanOpt);

% Assign Names and Holdings
Holdings = [10; 15;3];
Names = {'4% Bond'; '5% Bond'; 'Option 98'};

InstSet = instsetfield(InstSet, 'Index',1:3, 'FieldName', {'Quantity'}, 'Data', Holdings );
InstSet = instsetfield(InstSet, 'Index',1:3, 'FieldName', {'Name'}, 'Data', Names );
```

Examine the set of instruments contained in the variable `InstSet`.

```
instdisp(InstSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.04       27-Feb-2007    27-Feb-2009    1      0     1            NaN       NaN
2     Bond 0.05       27-Feb-2007    27-Feb-2010    1      0     1            NaN       NaN

Index Type    UnderInd OptSpec Strike ExerciseDates  AmericanOpt Quantity Name
3     OptBond 2        call    98     27-Feb-2010    1           3        Option 98
```

Price the Portfolio Using the BK Model

Calculate the price of each instrument in the portfolio.

```
[Price, PTree] = bkprice(BKTree, InstSet)
```

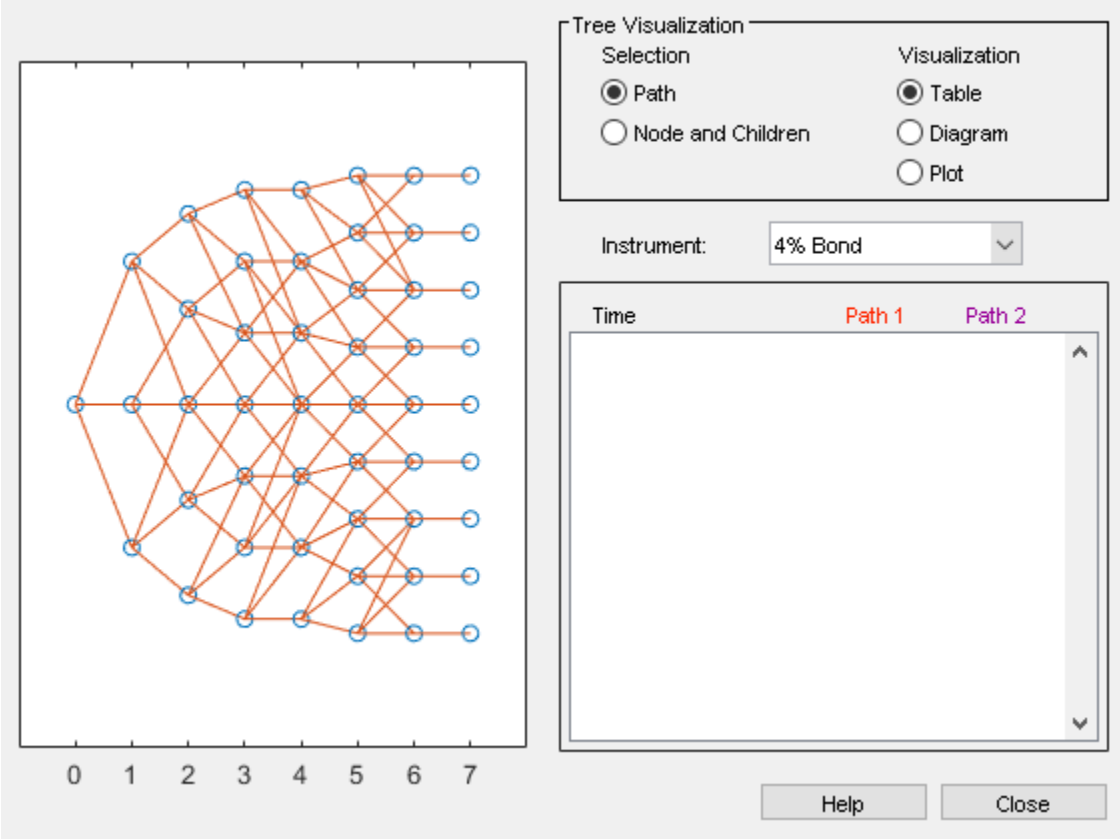
```
Price = 3×1

   98.8841
  101.3470
    3.3470


PTree = struct with fields:
     FinObj: 'BKPriceTree'
      PTree: {1x8 cell}
     AITree: {1x8 cell}
     ExTree: {1x8 cell}
       tObs: [0 1 2 3 5 7 10 20]
    Connect: {[2]  [2 3 4]  [2 3 4 5 6]  [2 3 3 4 5 5 6]  [2 3 4 5 6 7 8]  [2 2 3 ... ]}
      Probs: {1x6 cell}
```

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.

In the `Price` vector, the first element, 98.884, represents the price of the first instrument (4% Bond); the second element, 101.347, represents the price of the second instrument (5% Bond), and 3.347 represents the price of the American call option.

You can also display a graphical representation of the price tree to examine the prices on the nodes of the tree until maturity.

```
treeviewer(PTree,InstSet);
```



Add More Instruments to the Existing Portfolio

Add instruments to the existing portfolio: cap, floor, floating rate note, vanilla swap and a puttable and callable bond.

```
% Cap
StrikeC =0.035;
InstSet = instadd(InstSet,'Cap', StrikeC, Settle, datetime(2010,2,27));

% Floor
StrikeF =0.05;
InstSet = instadd(InstSet,'Floor', StrikeF, Settle, datetime(2009,2,27));

% Floating Rate Note
InstSet = instadd(InstSet,'Float', 30, Settle, datetime(2009,2,27));

% Vanilla Swap
 LegRate =[0.04 5];
 InstSet = instadd(InstSet,'Swap', LegRate, Settle, datetime(2010,2,27));

% Puttable and Callable Bonds
InstSet = instadd(InstSet,'OptEmBond', CouponRate, Settle,datetime(2010,2,27), {'put';'call'},..
                  Strike, datetime(2010,2,27),'AmericanOpt', 1, 'Period', 1);

% Process Names and Holdings
Holdings = [15 ;5 ;8; 7; 9; 4];
Names = {'3.5% Cap';'5% Floor';'30BP Float';'4%/5BP Swap'; 'PuttBond'; 'CallBond' };
```

```
InstSet = instsetfield(InstSet, 'Index',4:9, 'FieldName', {'Quantity'}, 'Data', Holdings );
InstSet = instsetfield(InstSet, 'Index',4:9, 'FieldName', {'Name'}, 'Data', Names );
```

Examine the set of instruments contained in the variable `InstSet`.

```
instdisp(InstSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCoup
1     Bond 0.04       27-Feb-2007    27-Feb-2009    1      0     1            NaN       NaN
2     Bond 0.05       27-Feb-2007    27-Feb-2010    1      0     1            NaN       NaN

Index Type    UnderInd OptSpec Strike ExerciseDates  AmericanOpt Quantity Name
3     OptBond 2        call    98     27-Feb-2010    1           3        Option 98

Index Type Strike Settle         Maturity       CapReset Basis Principal Quantity Name
4     Cap  0.035  27-Feb-2007    27-Feb-2010    1        0     100       15       3.5% Cap

Index Type  Strike Settle         Maturity       FloorReset Basis Principal Quantity Name
5     Floor 0.05   27-Feb-2007    27-Feb-2009    1          0     100       5        5% Floor

Index Type  Spread Settle         Maturity       FloatReset Basis Principal EndMonthRule CapRate
6     Float 30     27-Feb-2007    27-Feb-2009    1          0     100       1            Inf

Index Type LegRate   Settle         Maturity       LegReset Basis Principal LegType EndMonthRule
7     Swap [0.04  5] 27-Feb-2007    27-Feb-2010    [NaN]    0     100       [NaN]   1

Index Type      CouponRate Settle         Maturity       OptSpec Strike ExerciseDates
8     OptEmBond 0.04       27-Feb-2007    27-Feb-2010    put     98     27-Feb-2007   27-Feb-2010
9     OptEmBond 0.05       27-Feb-2007    27-Feb-2010    call    98     27-Feb-2007   27-Feb-2010
```

Hedging

The idea behind hedging is to minimize exposure to market movements. As the underlying changes, the proportions of the instruments forming the portfolio may need to be adjusted to keep the sensitivities within the desired range.

Calculate sensitivities using the BK model.

```
[Delta, Gamma, Vega, Price] = bksens(BKTree, InstSet);
```

Get the current portfolio holdings.

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

Create 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.

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

```
         98.88         10.00       -185.47        528.47             0
        101.35         15.00       -277.51       1045.05             0
          3.35          3.00       -223.52      11843.32             0
```

```
          2.77         15.00        250.04       2921.11         -0.00
          0.75          5.00       -132.97      11566.69             0
        100.56          8.00         -0.80          2.02             0
         -1.53          7.00       -272.08       1027.85          0.00
         98.60          9.00       -168.92      21712.82             0
         98.00          4.00        -53.99     -10798.27             0
```

The first column above is the dollar unit price of each instrument, the second column is the number of contracts of each instrument, and the third, fourth, and fifth columns are the dollar delta, gamma, and vega sensitivities.

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

```
TargetSens  = Holdings' * Sensitivities
```

```
TargetSens = 1×3

      -7249.21     317573.92         -0.00
```

Obtain a Neutral Sensitivity Portfolio Using `hedgeslf`

Suppose you want to obtain a delta, gamma and vega neutral portfolio. The function `hedgeslf` finds the reallocation in a portfolio of financial instruments closest to being self-financing (maintaining constant portfolio value).

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

```
Sens = 3×1

         -0.00
         -0.00
         -0.00


Value1 =
       4637.54


Quantity = 9×1

         10.00
          5.26
         -5.11
          7.06
         -3.05
         12.45
         -7.36
          8.47
         10.37
```

The function `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, you can verify the cost by comparing the portfolio values.

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

```
Value0 =
       4637.54
```

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.

Adding Constraints to Hedge a Portfolio

Suppose that you want to place upper and lower bounds on the individual instruments in the portfolio. Let's say that you want to bound the position of all instruments to within +/- 11 contracts.

Applying these constraints disallows the current positions in the fifth and eighth instruments. All other instruments are currently within the upper/lower bounds.

```
% Specify the lower and upper bounds
LowerBounds = [-11  -11  -11  -11  -11  -11  -11  -11  -11];
UpperBounds = [ 11   11   11   11   11   11   11   11   11];

% Use the function portcons to build the constraints
ConSet = portcons('AssetLims', LowerBounds, UpperBounds);

% Apply the constraints to the portfolio
[Sens, Value, Quantity1] = hedgeslf(Sensitivities, Price, Holdings, [], ConSet)
```

```
Sens = 3×1

             0
             0
             0


Value =
             0


Quantity1 = 9×1

             0
             0
             0
             0
             0
             0
             0
             0
             0
```

Observe that the `hedgeslf` function enforces the bounds on the fifth and eighth instruments, and the portfolio continues to be fully hedged and self-financing.

```
set(0, 'format', OldFormat);
```

Input Arguments

`Sensitivities` — Sensitivities of each instrument

matrix

Sensitivities of each instrument, specified as a number of instruments (`NINST`) by number of sensitivities (`NSENS`) matrix of dollar sensitivities. Each row represents a different instrument. Each column represents a different sensitivity.

Data Types: `double`

`Price` — Instrument prices

vector

Instrument prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`CurrentHolds` — Contracts allocated to each instrument

vector

Contracts allocated to each instrument, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`FixedInd` — Number of fixed instruments

`1` (default) | vector

(Optional) Number of fixed instruments, specified as an `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]`. The default is `FixedInd = 1`; the holdings in the first instrument are held fixed. If no instruments are to be held fixed, enter `FixedInd = [ ]`.

Data Types: `double`

`ConSet` — Additional conditions on the portfolio reallocations

`[ ]` (default) | matrix

(Optional) Additional conditions on the portfolio reallocations, specified as a 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)`.

Note Constraints `PortHolds(FixedInd) = CurrentHolds(FixedInd)` are appended to any constraints passed in `ConSet`. Pass `FixedInd = [ ]` to specify all constraints through `ConSet`. The default constraints generated by `portcons` are inappropriate, since they require the sum of all holdings to be positive and equal to `1`.

Data Types: `double`

Output Arguments

`PortSens` — Portfolio dollar sensitivities
vector

Portfolio dollar sensitivities, returned as a number of points `NSENS`-by-`1` vector. When a perfect hedge exists, `PortSens` is zeros. Otherwise, the best hedge possible is chosen.

`PortValue` — Total portfolio value
numeric

Total portfolio value, returned as a scalar value. When a perfectly self-financing hedge exists, `PortValue` is equal to the value `dot(Price,CurrentHolds)` of the initial portfolio.

`PortHolds` — Contracts allocated to each instrument
vector

Contracts allocated to each instrument, returned as an `NINST`-by-`1` vector. This is the reallocated portfolio.

Version History

Introduced before R2006a

See Also

`hedgeopt` | `lsqlin` | `portcons`

Topics
“Portfolio Creation Using Functions” on page 1-6
“Self-Financing Hedges with hedgeslf” on page 4-9
“Instrument Constructors” on page 1-15
“Hedging” on page 4-2
“Supported Equity Derivative Functions” on page 3-19

hjmprice

Instrument prices from Heath-Jarrow-Morton interest-rate tree

Syntax

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

Description

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

`hjmprice` handles instrument types: `'Bond'`, `'CashFlow'`, `'OptBond'`, `'OptEmBond'`, `'OptEmBond'`, `'OptFloat'`, `'OptEmFloat'`, `'Fixed'`, `'Float'`, `'Cap'`, `'Floor'`, `'RangeFloat'`, `'Swap'`. See `instadd` to construct defined types.

`[Price,PriceTree] = hjmprice(___,Options)` adds an optional input argument for `Options`.

Examples

Price the Cap and Bond Instruments Contained in an Instrument Set

Load the HJM tree and instruments from the data file `deriv.mat`.

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

instdisp(HJMSubSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCouponDate LastCouponDate StartDate Face Na
1     Bond 0.04       01-Jan-2000    01-Jan-2003    1      NaN   NaN          NaN       NaN             NaN            NaN       NaN  4%
2     Bond 0.04       01-Jan-2000    01-Jan-2004    2      NaN   NaN          NaN       NaN             NaN            NaN       NaN  4%

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

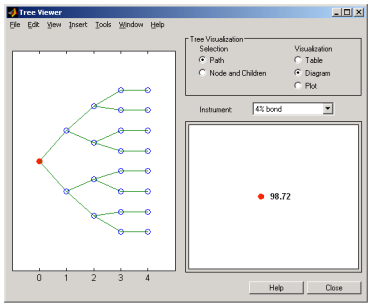
Use `hjmprice` to price the instruments.

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

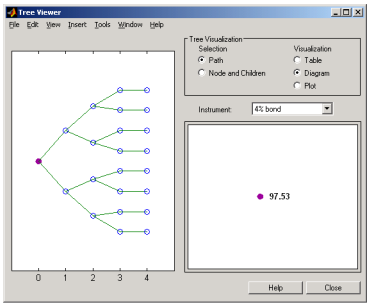
```
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In checktree (line 289)
  In hjmprice (line 85)

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.

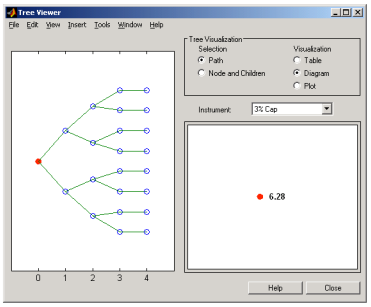
```
treeviewer(PriceTree)
```



First 4% Bond (Maturity 2003)



Second 4% Bond (Maturity 2004)



3% Cap

Price Multi-Stepped Coupon Bonds

The data for the interest-rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2010';
StartDates = ValuationDate;
EndDates = {'Jan-1-2011'; 'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'};
Compounding = 1;
```

Create a `RateSpec`.

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

```
RS = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
       StartDates: 734139
    ValuationDate: 734139
            Basis: 0
     EndMonthRule: 1
```

Create a portfolio of stepped coupon bonds with different maturities.

```
Settle = '01-Jan-2010';
Maturity = {'01-Jan-2011';'01-Jan-2012';'01-Jan-2013';'01-Jan-2014'};
CouponRate = {{'01-Jan-2011' .042;'01-Jan-2012' .05; '01-Jan-2013' .06; '01-Jan-2014' .07}};

ISet = instbond(CouponRate, Settle, Maturity, 1);
instdisp(ISet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCoup
1     Bond [Cell]     01-Jan-2010    01-Jan-2011    1      0     1            NaN       NaN
2     Bond [Cell]     01-Jan-2010    01-Jan-2012    1      0     1            NaN       NaN
3     Bond [Cell]     01-Jan-2010    01-Jan-2013    1      0     1            NaN       NaN
4     Bond [Cell]     01-Jan-2010    01-Jan-2014    1      0     1            NaN       NaN
```

Build the tree with the following data:

```
Volatility = [.2; .19; .18; .17];
CurveTerm = [ 1;  2;   3;   4];
HJMTimeSpec = hjmtimespec(ValuationDate, EndDates);
HJMVolSpec = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
HJMT = hjmtree(HJMVolSpec,RS,HJMTimeSpec)
```

```
HJMT = struct with fields:
      FinObj: 'HJMFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734139 734504 734869 735235]
        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]}
```

Compute the price of the stepped coupon bonds.

```
PHJM = hjmprice(HJMT, ISet)
```

```
PHJM = 4×1

  100.6763
  100.7368
  100.9266
  101.0115
```

Price a Portfolio of Stepped Callable Bonds and Stepped Vanilla Bonds

The data for the interest-rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2010';
StartDates = ValuationDate;
EndDates = {'Jan-1-2011'; 'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'};
Compounding = 1;
```

Create a RateSpec.

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

```
RS = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
          EndTimes: [4x1 double]
        StartTimes: [4x1 double]
          EndDates: [4x1 double]
        StartDates: 734139
    ValuationDate: 734139
            Basis: 0
     EndMonthRule: 1
```

Create an instrument portfolio of three stepped callable bonds and three stepped vanilla bonds and display the instrument portfolio.

```
Settle = '01-Jan-2010';
Maturity = {'01-Jan-2012';'01-Jan-2013';'01-Jan-2014'};
CouponRate = {{'01-Jan-2011' .042;'01-Jan-2012' .05; '01-Jan-2013' .06; '01-Jan-2014' .07}};
OptSpec='call';
Strike=100;
ExerciseDates='01-Jan-2011'; %Callable in one year

% Bonds with embedded option
ISet = instoptembnd(CouponRate, Settle, Maturity, OptSpec, Strike,...
ExerciseDates, 'Period', 1);

% Vanilla bonds
ISet = instbond(ISet, CouponRate, Settle, Maturity, 1);

instdisp(ISet)
```

```
Index Type      CouponRate Settle         Maturity       OptSpec Strike ExerciseDates  Period Bas
1     OptEmBond [Cell]     01-Jan-2010    01-Jan-2012    call    100    01-Jan-2011    1      0
2     OptEmBond [Cell]     01-Jan-2010    01-Jan-2013    call    100    01-Jan-2011    1      0
3     OptEmBond [Cell]     01-Jan-2010    01-Jan-2014    call    100    01-Jan-2011    1      0

Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCoup
4     Bond [Cell]     01-Jan-2010    01-Jan-2012    1      0     1            NaN       NaN
5     Bond [Cell]     01-Jan-2010    01-Jan-2013    1      0     1            NaN       NaN
6     Bond [Cell]     01-Jan-2010    01-Jan-2014    1      0     1            NaN       NaN
```

Build the tree with the following data:

```
Volatility = [.2; .19; .18; .17];
CurveTerm = [ 1;  2;   3;   4];
HJMTimeSpec = hjmtimespec(ValuationDate, EndDates);
HJMVolSpec = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
HJMT = hjmtree(HJMVolSpec,RS,HJMTimeSpec)
```

```
HJMT = struct with fields:
      FinObj: 'HJMFwdTree'
```

```
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734139 734504 734869 735235]
        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]}
```

Price the instrument set using `hjmprice`.

```
PHJM = hjmprice(HJMT, ISet)
```

```
PHJM = 6×1

  100.3682
  100.1557
   99.9232
  100.7368
  100.9266
  101.0115
```

The first three rows correspond to the price of the stepped callable bonds and the last three rows correspond to the price of the stepped vanilla bonds.

Compute the Price of a Portfolio of Instruments

The data for the interest-rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2011';
StartDates = ValuationDate;
EndDates = {'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'; 'Jan-1-2015'};
Compounding = 1;
```

Create a `RateSpec`.

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

```
RS = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
          EndTimes: [4x1 double]
        StartTimes: [4x1 double]
          EndDates: [4x1 double]
        StartDates: 734504
    ValuationDate: 734504
            Basis: 0
     EndMonthRule: 1
```

Create an instrument portfolio with two range notes and a floating rate note with the following data and display the results:

```
Spread = 200;
Settle = 'Jan-1-2011';
Maturity = 'Jan-1-2014';

% First Range Note
RateSched(1).Dates = {'Jan-1-2012'; 'Jan-1-2013'  ; 'Jan-1-2014'};
RateSched(1).Rates  = [0.045 0.055; 0.0525  0.0675; 0.06 0.08];

% Second Range Note
RateSched(2).Dates = {'Jan-1-2012'; 'Jan-1-2013' ; 'Jan-1-2014'};
RateSched(2).Rates  = [0.048 0.059; 0.055  0.068 ; 0.07 0.09];

% Create an InstSet
InstSet = instadd('RangeFloat', Spread, Settle, Maturity, RateSched);

% Add a floating-rate note
InstSet = instadd(InstSet, 'Float', Spread, Settle, Maturity);

% Display the portfolio instrument
instdisp(InstSet)
```

```
Index Type       Spread Settle         Maturity       RateSched FloatReset Basis Principal EndMo
1     RangeFloat 200    01-Jan-2011    01-Jan-2014    [Struct]  1          0     100       1
2     RangeFloat 200    01-Jan-2011    01-Jan-2014    [Struct]  1          0     100       1

Index Type  Spread Settle         Maturity       FloatReset Basis Principal EndMonthRule CapRate
3     Float 200    01-Jan-2011    01-Jan-2014    1          0     100       1            Inf
```

The data to build the tree is as follows:

```
Volatility = [.2; .19; .18; .17];
CurveTerm = [ 1;  2;   3;   4];
MaTree = {'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'; 'Jan-1-2015'};
HJMTS = hjmtimespec(ValuationDate, MaTree);
HJMVS = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
HJMT = hjmtree(HJMVS, RS, HJMTS)
```

```
HJMT = struct with fields:
      FinObj: 'HJMFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734504 734869 735235 735600]
        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]}
```

Price the portfolio.

```
Price = hjmprice(HJMT, InstSet)
```

```
Price = 3×1
```

```
   91.1555
   90.6656
  105.5147
```

Create a Float-Float Swap and Price with `hjmprice`

Use `instswap` to create a float-float swap and price the swap with `hjmprice`.

```
RateSpec = intenvset('Rates',.05,'StartDate',today,'EndDate',datemnth(today,60));
IS = instswap([.02 .03],today,datemnth(today,60),[], [], [], [1 1]);
VolSpec = hjmvolspec('Constant', .2);
TimeSpec = hjmtimespec(today,cfdates(today,datemnth(today,60),1));
HJMTree = hjmtree(VolSpec,RateSpec,TimeSpec);
hjmprice(HJMTree,IS)
```

```
ans = -4.3220
```

Price Multiple Swaps with `hjmprice`

Use `instswap` to create multiple swaps and price the swaps with `hjmprice`.

```
RateSpec = intenvset('Rates',.05,'StartDate',today,'EndDate',datemnth(today,60));
IS = instswap([.03 .02],today,datemnth(today,60),[], [], [], [1 1]);
IS = instswap(IS,[200 300],today,datemnth(today,60),[], [], [], [0 0]);
IS = instswap(IS,[.08 300],today,datemnth(today,60),[], [], [], [1 0]);
VolSpec = hjmvolspec('Constant', .2);
TimeSpec = hjmtimespec(today,cfdates(today,datemnth(today,60),1));
HJMTree = hjmtree(VolSpec,RateSpec,TimeSpec);
hjmprice(HJMTree,IS)
```

```
ans = 3×1

    4.3220
   -4.3220
   -0.2701
```

Input Arguments

`HJMTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hjmtree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of NINST instruments, specified using instadd. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: struct

Options — Derivatives pricing options structure
structure

(Optional) Derivatives pricing options structure, created using derivset.

Data Types: struct

Output Arguments

Price — Price for each instrument
vector

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

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.
- optfloatbybdt — Price a floating-rate note with an option from an HJM tree.
- optemfloatbybdt — Price a floating-rate note with an embedded option from an HJM tree.
- rangefloatbyhjm — Price range floating note using an HJM tree.
- swapbyhjm — Price a swap from an HJM tree.
- swaptionbyhjm — Price a swaption from an HJM tree.

PriceTree — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.

Version History

Introduced before R2006a

See Also

`hjmsens` | `hjmtree` | `hjmvolspec` | `instadd` | `intenvprice` | `intenvsens`

Topics

“Computing Instrument Prices” on page 2-81
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

hjmsens

Instrument prices and sensitivities from Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Delta,Gamma,Vega,Price] = hjmsens(HJMTree,InstSet)
[Delta,Gamma,Vega,Price] = hjmsens(___,Options)
```

Description

`[Delta,Gamma,Vega,Price] = hjmsens(HJMTree,InstSet)` computes instrument sensitivities and prices for instruments using an interest-rate tree created with the `hjmtree` function. All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide by the respective instrument price.

`hjmsens` handles instrument types: `'Bond'`, `'CashFlow'`, `'OptBond'`, `'OptEmBond'`, `'OptEmBond'`, `'OptFloat'`, `'OptEmFloat'`, `'Fixed'`, `'Float'`, `'Cap'`, `'Floor'`, `'RangeFloat'`, `'Swap'`. See `instadd` for information on instrument types.

`[Delta,Gamma,Vega,Price] = hjmsens(___,Options)` adds an optional input argument for `Options`.

Examples

Compute Instrument Sensitivities Using an HJM Interest-Rate Tree

Load the tree and instruments from the `deriv.mat` data file. Compute `Delta` and `Gamma` for the cap and bond instruments contained in the instrument set.

```
load deriv.mat;
HJMSubSet = instselect(HJMInstSet,'Type', {'Fixed', 'Cap'});
instdisp(HJMSubSet)
```

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

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

Compute the `Delta` and `Gamma` for the cap and bond instruments.

```
[Delta, Gamma] = hjmsens(HJMTree, HJMSubSet)
```

```
Delta = 2×1

 -272.6462
  294.9700


Gamma = 2×1
10^3 ×
```

```
    1.0299
    6.8526
```

Input Arguments

`HJMTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hjmtree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Delta` — Rate of change of instruments prices with respect to changes in interest rate

vector

Rate of change of instruments prices with respect to changes in the interest rate, returned as a `NINST`-by-`1` vector of deltas. `Delta` is computed by finite differences in calls to `hjmtree`.

Note `Delta` is calculated based on yield shifts of 100 basis points.

`Gamma` — Rate of change of instruments deltas with respect to changes in interest rate

vector

Rate of change of instruments deltas with respect to changes in the interest rate, returned as a `NINST`-by-`1` vector of gammas. `Gamma` is computed by finite differences in calls to `hjmtree`.

Note `Gamma` is calculated based on yield shifts of 100 basis points.

`Vega` — Rate of change of instruments prices with respect to changes in volatility

vector

Rate of change of instruments prices with respect to changes in the volatility, returned as a `NINST`-by-`1` vector of vegas. Volatility is $\sigma(t, T)$ of the interest rate. `Vega` is computed by finite differences in calls to `hjmtree`. For information on the volatility process, see `hjmvolspec`.

Note `Vega` is calculated based on 1% shift in the volatility process.

`Price` — Price of each instrument

vector

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

Version History

Introduced before R2006a

See Also

`hjmprice` | `hjmtree` | `hjmvolspec` | `instadd`

Topics

“Computing Instrument Sensitivities” on page 2-89
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

hjmtimespec

Specify time structure for Heath-Jarrow-Morton interest-rate tree

Syntax

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

Description

`TimeSpec = hjmtimespec(ValuationDate,Maturity)` sets the number of levels and node times for a `hjmtree` and determines the mapping between dates and time for rate quoting.

`TimeSpec = hjmtimespec( ___ ,Compounding)` adds the optional argument `Compounding`.

Examples

Set the Number of Levels and Node Times for an HJM Tree

This example shows how to specify an eight-period tree with semiannual nodes (every six months) and use exponential compounding to report rates.

```
Compounding = -1;
ValuationDate = datetime(1999,1,15);
Maturity = datemnth(ValuationDate, 6*(1:8)');
TimeSpec = hjmtimespec(ValuationDate, Maturity, Compounding)
```

```
TimeSpec = struct with fields:
           FinObj: 'HJMTimeSpec'
    ValuationDate: 730135
         Maturity: [8x1 double]
      Compounding: -1
            Basis: 0
     EndMonthRule: 1
```

Input Arguments

`ValuationDate` — Pricing date and first observation in the tree

datetime scalar | string scalar | date character vector

Pricing date and first observation in the tree, specified as a scalar datetime, string, or date character vector.

To support existing code, `hjmtimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Dates marking the cash flow dates of the tree

datetime array | string array | date character vector

Dates marking the cash flow dates of the tree, specified as `NLEVELS`-by-`1` vector using a datetime array, string array, or date character vectors. Cash flows with these maturities fall on tree nodes. `Maturity` should be in increasing order.

To support existing code, `hjmtimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Compounding` — Rate at which the input zero rates were compounded when annualized
1 (default) | integer with value of 1, 2, 3, 4, 6, 12, 365, or -1

(Optional) Rate at which the input zero rates were compounded when annualized, specified as a scalar integer value.

- If `Compounding` = 1, 2, 3, 4, 6, 12:

  `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units; for example, `T = F` is one year.
- If `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.
- If `Compounding` = −1:

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

Data Types: `double`

Output Arguments

`TimeSpec` — Specification for the time layout for `hjmtree`
structure

Specification for the time layout for `hjmtree`, returned as a structure. 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(end)`.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `hjmtimespec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmtree` | `hjmvolspec`

Topics

“Specifying the Time Structure (TimeSpec)” on page 2-70
“Creating Trees” on page 2-72
“Examining Trees” on page 2-72
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

hjmtree

Build Heath-Jarrow-Morton interest-rate tree

Syntax

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

Description

`HJMTree = hjmtree(VolSpec,RateSpec,TimeSpec)` creates a structure containing time and forward-rate information on a bushy tree.

Examples

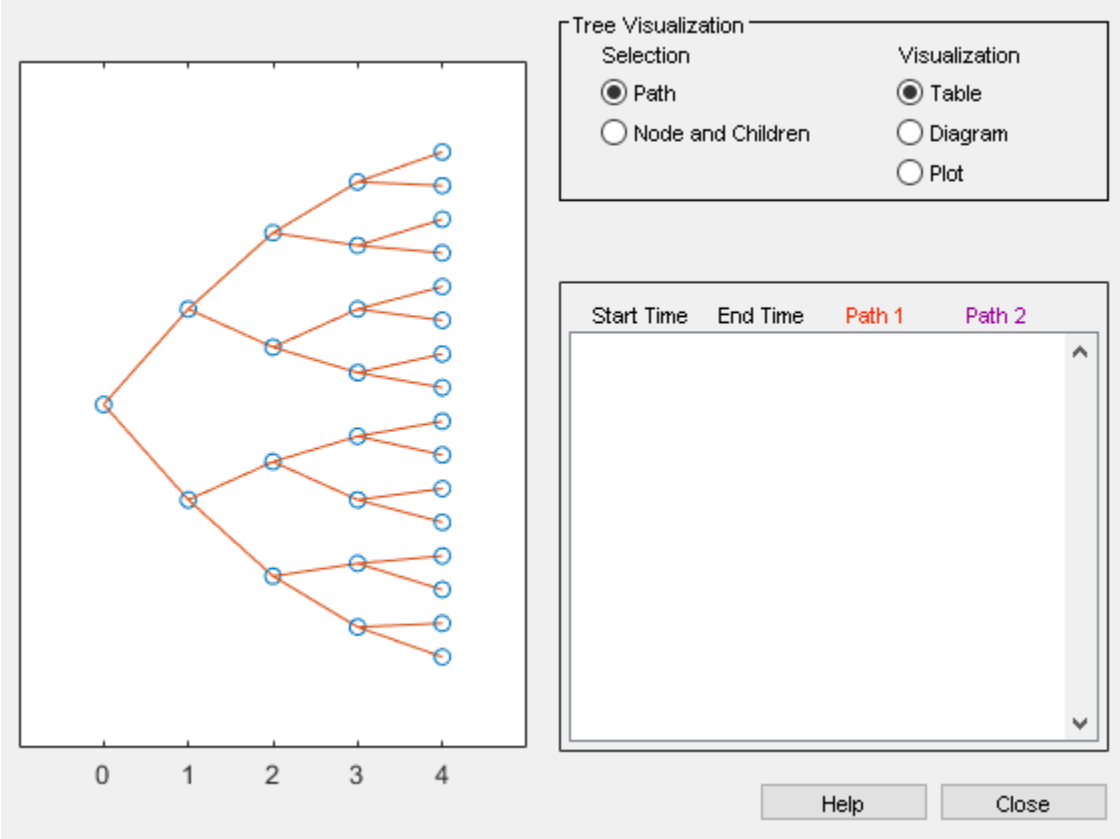
Create a HJMTree

Using the data provided, create a HJM volatility specification (using `hjmvolspec`), rate specification (using `intenvset`), and tree time layout specification (using `hjmtimespec`). Then use these specifications to create a 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)

HJMTree = struct with fields:
      FinObj: 'HJMFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 4]
        dObs: [730486 730852 731217 731582 731947]
        TFwd: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [4]}
      CFlowT: {[5x1 double]  [4x1 double]  [3x1 double]  [2x1 double]  [5]}
     FwdTree: {1x5 cell}
```

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

```
treeviewer(HJMTree)
```



Input Arguments

`VolSpec` — Volatility process specification

structure

Volatility process specification, specified using the `VolSpec` output obtained from `hjmvolspec`. `VolSpec` sets the number of factors and the rules for computing the volatility $\sigma(t, T)$ for each factor.

Data Types: `struct`

`RateSpec` — Interest-rate specification for initial risk-free rate curve

structure

Interest-rate specification for initial rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`TimeSpec` — Time tree layout specification

structure

Time tree layout specification, specified using the `TimeSpec` output obtained from `hjmtimespec`. The TimeSpec defines the observation dates of the HJM tree and the `Compounding` rule for date to time mapping and price-yield formulas.

Data Types: `struct`

Output Arguments

`HJMTree` — Time and interest-rate information of a bushy tree

structure

Time and interest-rate information of a bushy tree, returned as a structure.

Version History

Introduced before R2006a

See Also

`hjmprice` | `hjmtimespec` | `hjmvolspec` | `intenvset`

Topics

“Creating Trees” on page 2-72
“Examining Trees” on page 2-72
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

hjmvolspec

Specify Heath-Jarrow-Morton interest-rate volatility process

Syntax

```
VolSpec = hjmvolspec(Factor,Sigma_0)
VolSpec = hjmvolspec(Factor,CurveVol,CurveTerm)
VolSpec = hjmvolspec(Factor,Sigma_0,Lambda)
VolSpec = hjmvolspec(Factor,Sigma_0,CurveDecay,CurveTerm)
VolSpec = hjmvolspec(Factor,CurveProp,CurveTerm,MaxSpot)
```

Description

`VolSpec = hjmvolspec(Factor,Sigma_0)` creates a Constant volatility (Ho-Lee) structure for `hjmtree` by specifying the `Factor` as `'Constant'`.

`VolSpec = hjmvolspec(Factor,CurveVol,CurveTerm)` creates a Stationary volatility structure for `hjmtree` by specifying the `Factor` as `'Stationary'`.

`VolSpec = hjmvolspec(Factor,Sigma_0,Lambda)` creates an Exponential volatility structure for `hjmtree` by specifying the `Factor` as `'Exponential'`.

`VolSpec = hjmvolspec(Factor,Sigma_0,CurveDecay,CurveTerm)` creates a Vasicek, Hull-White volatility structure for `hjmtree` by specifying the `Factor` as `'Vasicek'`.

`VolSpec = hjmvolspec(Factor,CurveProp,CurveTerm,MaxSpot)` creates a Nearly proportional stationary volatility structure for `hjmtree` by specifying the `Factor` as `'Proportional'`.

Examples

Compute the VolSpec Structure to Specify a Proportional Volatility Model for HJMTree

This example shows how to compute the `VolSpec` structure to specify the volatility model for `hjmtree` when volatility is single-factor proportional.

```
CurveProp = [0.11765; 0.08825; 0.06865];
CurveTerm = [1; 2; 3];
VolSpec = hjmvolspec('Proportional', CurveProp, CurveTerm, 1e6)
```

```
VolSpec = struct with fields:
          FinObj: 'HJMVolSpec'
    FactorModels: {'Proportional'}
      FactorArgs: {{1x3 cell}}
      SigmaShift: 0
      NumFactors: 1
       NumBranch: 2
         PBranch: [0.5000 0.5000]
     Fact2Branch: [-1 1]
```

Compute the VolSpec Structure to Specify an Exponential Volatility Model for HJMTree

This example shows how to compute the `VolSpec` structure to specify the volatility model for `hjmtree` when volatility is two-factor exponential and constant.

```
VolSpec = hjmvolspec('Exponential', 0.1, 1, 'Constant', 0.2)
```

```
VolSpec = struct with fields:
          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]
```

Input Arguments

Factor — Volatility factor

character vector with value of `'Constant'`, `'Stationary'`, `'Exponential'`, `'Vasicek'`, or `'Proportional'`

Volatility factor, specified as a character vector with one of the following values:

- `'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'`

  $\sigma(t,T)$ = `Sigma_0*exp(-Decay(T-t))`
- `'Proportional'`

  $\sigma(t,T)$ = `Prop(T-t)*max(SpotRate(t),MaxSpot)`

Note You can specify more than one `Factor` by concatenating Factor names and their associated parameters.

Data Types: `char`

Sigma_0 — Base volatility over a unit time

numeric

Base volatility over a unit, specified as a scalar numeric value.

Data Types: `double`

`Lambda` — Decay factor
numeric

Decay factor, specified as a scalar numeric value.

Data Types: `double`

`CurveVol` — Number of curve `Vol` values at sample points
numeric vector

Number of curve `Vol` values at sample points, specified as a `NCURVES`-by`1` vector.

Data Types: `double`

`CurveTerm` — Number of curve `Term` values at sample points
numeric vector

Number of curve `Term` values at sample points, specified as a `NCURVES`-by-`1` vector.

Data Types: `double`

`CurveDecay` — Number of curve `Decay` values at sample points
numeric vector

Number of curve `Decay` values at sample points, specified as a `NPOINTS`-by-`1` vector.

Data Types: `double`

`CurveProp` — Number of curve `Prop` values at sample points
numeric vector

Number of curve `Prop` values at sample points, specified as a `NCURVES`-by-`1` vector.

Data Types: `double`

`MaxSpot` — Maximum spot rate
numeric

Maximum spot rate, specified as a scalar numeric value.

Data Types: `double`

Output Arguments

`VolSpec` — Specification for the volatility model for `bktree`
structure

Structure specifying the volatility model for `bktree`. `hjmvolspec` defines an HJM forward-rate volatility process based on the specified input `Factor`.

More About

Volatility Process

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).

Version History

Introduced before R2006a

See Also

`hjmtimespec` | `hjmtree`

Topics

“Specifying the Volatility Model (VolSpec)” on page 2-68
“Creating Trees” on page 2-72
“Examining Trees” on page 2-72
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

HullWhite1F

Create Hull-White one-factor model

Description

The Hull-White one-factor model is specified using the zero curve, alpha, and sigma parameters.

Specifically, the HullWhite1F model is defined using the following equations:

$$dr = [\theta(t) - a(t)r]dt + \sigma(t)dW$$

where:

dr is the change in the short-term interest rate over a small interval.

r is the short-term interest rate.

$\Theta(t)$ is a function of time determining the average direction in which r moves, chosen such that movements in r are consistent with today's zero coupon yield curve.

α is the mean reversion rate.

dt is a small change in time.

σ is the annual standard deviation of the short rate.

W is the Brownian motion.

Creation

Syntax

```
HW1F = HullWhite1F(ZeroCurve,Alpha,Sigma)
```

Description

`HW1F = HullWhite1F(ZeroCurve,Alpha,Sigma)` creates a `HullWhite1F` (`HW1F`) object using the required arguments to set the Properties on page 11-717.

Properties

`ZeroCurve` — Zero curve
`IRDataCurve` object | `RateSpec`

Zero curve, specified as an output from `IRDataCurve` or a `RateSpec` that is obtained from `intenvset`. This is the zero curve used to evolve the path of future interest rates.

Data Types: `object` | `struct`

Alpha — Mean reversion
numeric

Mean reversion, specified either as a scalar or function handle which takes time as input and returns a scalar mean reversion value.

Data Types: double

Sigma — Volatility
numeric

Volatility, specified either as a scalar or function handle which takes time as input and returns a scalar mean volatility.

Data Types: double

Object Functions

simTermStructs Simulate term structures for Hull-White one-factor model

Examples

Create a Hull-White One-Factor Model Using an IRDataCurve

Create a Hull-White one-factor model using an IRDataCurve.

```
Settle = datetime(2007,12,15);
CurveTimes = [1:5 7 10 20]';
ZeroRates = [.01 .018 .024 .029 .033 .034 .035 .034]';
CurveDates = daysadd(Settle,360*CurveTimes,1);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

alpha = .1;
sigma = .01;

HW1F = HullWhite1F(irdc,alpha,sigma)
```

```
HW1F =
  HullWhite1F with properties:

    ZeroCurve: [1x1 IRDataCurve]
        Alpha: @(t,V)inAlpha
        Sigma: @(t,V)inSigma
```

Use the simTermStructs method with the HullWhite1F model to simulate term structures.

```
SimPaths = simTermStructs(HW1F, 10,'nTrials',100);
```

Create a Hull-White One-Factor Model Using a RateSpec

Create a Hull-White one-factor model using a RateSpec.

```
Settle = datetime(2007,12,15);
CurveTimes = [1:5 7 10 20]';
ZeroRates = [.01 .018 .024 .029 .033 .034 .035 .034]';
CurveDates = daysadd(Settle,360*CurveTimes,1);

RateSpec = intenvset('Rates',ZeroRates,'EndDates',CurveDates,'StartDate',Settle);

alpha = .1;
sigma = .01;

HW1F = HullWhite1F(RateSpec,alpha,sigma)

HW1F =
  HullWhite1F with properties:

    ZeroCurve: [1x1 IRDataCurve]
        Alpha: @(t,V)inAlpha
        Sigma: @(t,V)inSigma
```

Use the `simTermStructs` method with the `HullWhite1F` model to simulate term structures.

```
SimPaths = simTermStructs(HW1F, 10,'nTrials',100);
```

Simulate the Price of a Bond Using a Hull-White One-Factor Model Until the Bond's Maturity

Define the zero curve data.

```
Settle = datetime(2016,4,4);
ZeroTimes = [3/12 6/12 1 5 7 10 20 30]';
ZeroRates = [0.033 0.034 0.035 0.040 0.042 0.044 0.048 0.0475]';
ZeroDates = datemnth(Settle,ZeroTimes*12);
RateSpec = intenvset('StartDates', Settle,'EndDates', ZeroDates, 'Rates', ZeroRates)

RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [8x1 double]
            Rates: [8x1 double]
         EndTimes: [8x1 double]
       StartTimes: [8x1 double]
         EndDates: [8x1 double]
       StartDates: 736424
    ValuationDate: 736424
            Basis: 0
     EndMonthRule: 1
```

Define the bond parameters.

```
Maturity = datemnth(Settle,12*5);
CouponRate = 0;
```

Define the Hull-White parameters.

```
alpha = .1;
sigma = .01;
HW1F = HullWhite1F(RateSpec,alpha,sigma)

HW1F =
  HullWhite1F with properties:

    ZeroCurve: [1x1 IRDataCurve]
        Alpha: @(t,V)inAlpha
        Sigma: @(t,V)inSigma
```

Define the simulation parameters.

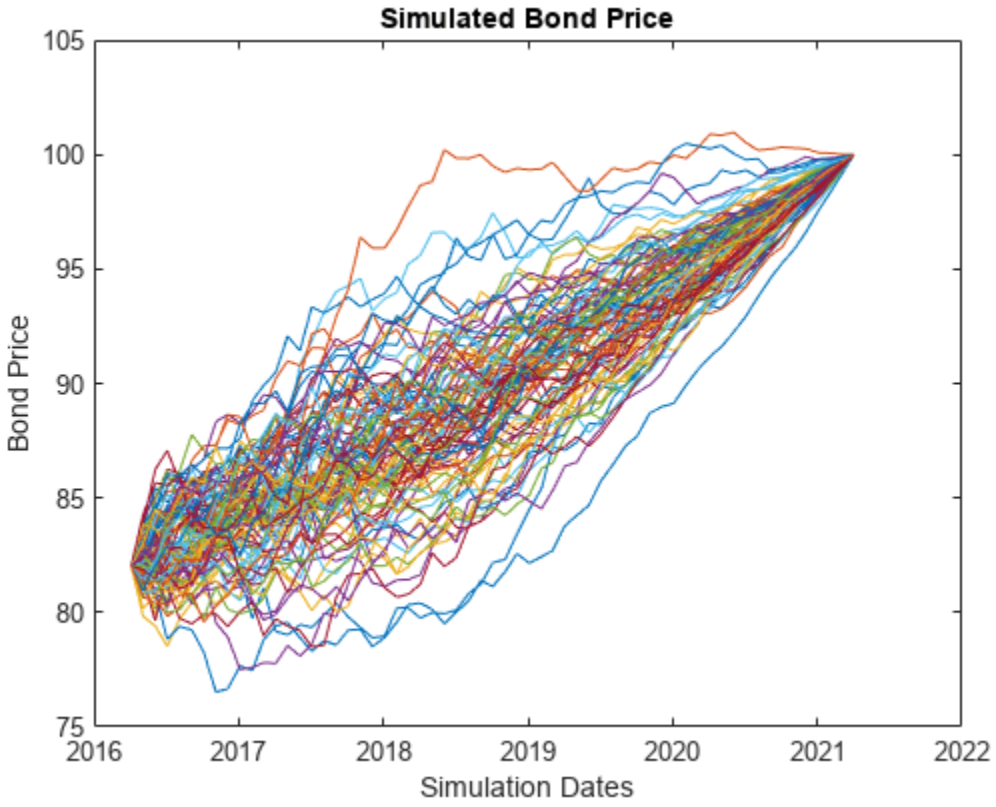
```
nTrials = 100;
nPeriods = 12*5;
deltaTime = 1/12;
SimZeroCurvePaths = simTermStructs(HW1F, nPeriods,'nTrials',nTrials,'deltaTime',deltaTime);
SimDates = datemnth(Settle,1:nPeriods);
```

Preallocate and initialize for the simulation.

```
SimBondPrice = zeros(nPeriods+1,nTrials);
SimBondPrice(1,:,:) = bondbyzero(RateSpec,CouponRate,Settle,Maturity);
SimBondPrice(end,:,:) = 100;
```

Compute the bond values for each simulation date and path, note that you can vectorize over the trial dimension.

```
for periodidx=1:nPeriods-1
    simRateSpec = intenvset('StartDate',SimDates(periodidx),'EndDates',...
        datemnth(SimDates(periodidx),ZeroTimes*12),'Rates',squeeze(SimZeroCurvePaths(periodidx+1
    SimBondPrice(periodidx+1,:) = bondbyzero(simRateSpec,CouponRate,SimDates(periodidx),Maturity
end

plot([Settle SimDates],SimBondPrice)
datetick
ylabel('Bond Price')
xlabel('Simulation Dates')
title('Simulated Bond Price')
```



Simulate the Total Return of a Bond Portfolio Until Maturity

Define the zero curve data.

```
Settle = datetime(2016,4,4);
ZeroTimes = [3/12 6/12 1 5 7 10 20 30]';
ZeroRates = [-0.01 -0.009 -0.0075 -0.003 -0.002 -0.001 0.002 0.0075]';
ZeroDates = datemnth(Settle,ZeroTimes*12);
RateSpec = intenvset('StartDates', Settle,'EndDates', ZeroDates, 'Rates', ZeroRates)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [8x1 double]
            Rates: [8x1 double]
          EndTimes: [8x1 double]
        StartTimes: [8x1 double]
          EndDates: [8x1 double]
        StartDates: 736424
    ValuationDate: 736424
            Basis: 0
     EndMonthRule: 1
```

Define the bond parameters for the five bonds in the portfolio.

```
Maturity = datemnth(Settle,12*5);  % All bonds have the same maturity
CouponRate = [0.035;0.04;0.02;0.015;0.042];  % Different coupon rates for the bonds
nBonds = length(CouponRate);
```

Define the Hull-White parameters.

```
alpha = .1;
sigma = .01;
HW1F = HullWhite1F(RateSpec,alpha,sigma)
```

```
HW1F =
  HullWhite1F with properties:

    ZeroCurve: [1x1 IRDataCurve]
        Alpha: @(t,V)inAlpha
        Sigma: @(t,V)inSigma
```

Define the simulation parameters.

```
nTrials = 1000;
nPeriods = 12*5;
deltaTime = 1/12;
SimZeroCurvePaths = simTermStructs(HW1F, nPeriods,'nTrials',nTrials,'deltaTime',deltaTime);
SimDates = datemnth(Settle,1:nPeriods);
```

Preallocate and initialize for the simulation.

```
SimBondPrice = zeros(nPeriods+1,nBonds,nTrials);
SimBondPrice(1,:,:) = repmat(bondbyzero(RateSpec,CouponRate,Settle,Maturity)',[1 1 nTrials]);
SimBondPrice(end,:,:) = 100;

[BondCF,BondCFDates,~,CFlowFlags] = cfamounts(CouponRate,Settle,Maturity);
BondCF(CFlowFlags == 4) = BondCF(CFlowFlags == 4) - 100;
SimBondCF = zeros(nPeriods+1,nBonds,nTrials);
```

Compute bond values for each simulation date and path. Note that you can vectorize over the trial dimension.

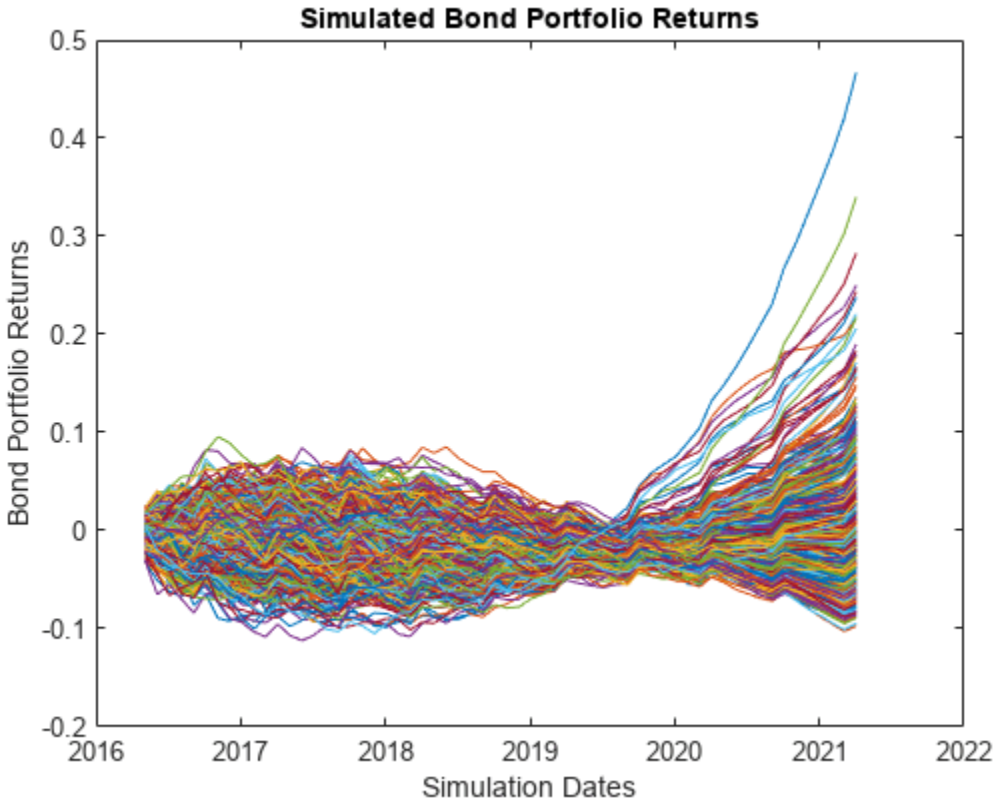
```
for periodidx=1:nPeriods
    if periodidx < nPeriods
        simRateSpec = intenvset('StartDate',SimDates(periodidx),'EndDates',...
            datemnth(SimDates(periodidx),ZeroTimes*12),'Rates',squeeze(SimZeroCurvePaths(periodic
        SimBondPrice(periodidx+1,:,:) = bondbyzero(simRateSpec,CouponRate,SimDates(periodidx),Ma
    end

    simidx = SimDates(periodidx) == BondCFDates;
    SimCF = zeros(1,nBonds);
    SimCF(any(simidx,2)) = BondCF(simidx);
    ReinvestRate = 1 + SimZeroCurvePaths(periodidx+1,1,:);
    SimBondCF(periodidx+1,:,:) = bsxfun(@times,bsxfun(@plus,SimBondCF(periodidx,:,:),SimCF),Reinv
end
```

Compute the total return series.

```
TotalCF = SimBondPrice + SimBondCF;
```

Assume the bond portfolio is equally weighted and plot the simulated bond portfolio returns.

```
TotalCF = squeeze(sum(TotalCF,2));

TotRetSeries = bsxfun(@rdivide,TotalCF(2:end,:),TotalCF(1,:)) - 1;
plot(SimDates,TotRetSeries)
datetick
ylabel('Bond Portfolio Returns')
xlabel('Simulation Dates')
title('Simulated Bond Portfolio Returns')
```



More About

Hull-White One-Factor Model

The Hull-White model is a single-factor, no-arbitrage yield curve model in which the short-term rate of interest is the random factor or state variable.

No-arbitrage means that the model parameters are consistent with the bond prices implied in the zero coupon yield curve.

Version History

Introduced in R2013a

References

[1] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[2] Hull, J. *Options, Futures, and Other Derivatives*. Prentice-Hall, 2011.

See Also

`LinearGaussian2F` | `LiborMarketModel` | `hwcalbycap` | `hwcalbyfloor` | `simTermStructs`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Supported Interest-Rate Instrument Functions” on page 2-3

simTermStructs

Simulate term structures for Hull-White one-factor model

Syntax

```
[ZeroRates,ForwardRates] = simTermStructs(HW1F,nPeriods)
[ZeroRates,ForwardRates] = simTermStructs(___,Name,Value)
```

Description

`[ZeroRates,ForwardRates] = simTermStructs(HW1F,nPeriods)` simulates future zero curve paths using a specified `HullWhite1F` object.

`[ZeroRates,ForwardRates] = simTermStructs(___,Name,Value)` adds optional name-value pair arguments.

Examples

Simulate Term Structures for the HullWhite1F Model

Create a `HW1F` object.

```
Settle = datetime(2007,12,15);
CurveTimes = [1:5 7 10 20]';
ZeroRates = [.01 .018 .024 .029 .033 .034 .035 .034]';
CurveDates = daysadd(Settle,360*CurveTimes,1);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

alpha = .1;
sigma = .01;

HW1F = HullWhite1F(irdc,alpha,sigma)
```

```
HW1F =
  HullWhite1F with properties:

    ZeroCurve: [1x1 IRDataCurve]
        Alpha: @(t,V)inAlpha
        Sigma: @(t,V)inSigma
```

Simulate the term structures for the specified `HW1F` object.

```
SimPaths = simTermStructs(HW1F, 10,'nTrials',100);
```

Input Arguments

`HW1F` — `HullWhite1F` object
object

`HullWhite1F` object, specified using the `HW1F` object created using `HullWhite1F`.

Data Types: `object`

`nPeriods` — Number of simulation periods
numeric

Number of simulation periods, specified as a numeric value. For example, to simulate 12 years with an annual spacing, specify `12` as the `nPeriods` input and `1` as the optional `deltaTime` input (note that the default value for `deltaTime` is `1`).

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[ZeroRates,ForwardRates] = simTermStructs(HW1F,NPeriods,'nTrials',100,'deltaTime',dt)`

`deltaTime` — Time step between `nPeriods`
`1` (default) | numeric

Time step between `nPeriods` measured in years, specified as the comma-separated pair consisting of `'deltaTime'` and a scalar numeric value. For example, to simulate 12 years with an annual spacing, specify `12` as the `nPeriods` input and `1` as the optional `deltaTime` input (note that the default value for `deltaTime` is `1`).

Data Types: `double`

`nTrials` — Number of simulated trials
`1` (default) | positive integer

Number of simulated trials (sample paths), specified as the comma-separated pair consisting of `'nTrials'` and a positive scalar integer value of `nPeriods` observations each. If you do not specify a value for this argument, the default is `1`, indicating a single path of correlated state variables.

Data Types: `double`

`antithetic` — Flag indicating whether antithetic sampling is used to generate Gaussian random variates
`false` (default) | positive integer

Flag indicating whether antithetic sampling is used to generate the Gaussian random variates that drive the zero-drift, unit-variance rate Brownian vector $dW(t)$, specified as the comma-separated pair consisting of `'antithetic'` and a Boolean scalar flag. For details, see `simBySolution`.

Data Types: `logical`

`Z` — Direct specification of dependent random noise process
Gaussian variates generated by `simBySolution` function (default) | numeric

Direct specification of the dependent random noise process, specified as the comma-separated pair consisting of `'Z'` and a numeric value. The `Z` value is used to generate the zero-drift, unit-variance

rate Brownian vector *dW*(*t*) that drives the simulation. For details, see `simBySolution` for the HWV model. If you do not specify a value for `Z`, `simBySolution` generates Gaussian variates.

Data Types: `double`

`Tenor` — Maturities to compute at each time step
tenor of `HullWhite1F` object zero curve (default) | numeric vector

Maturities to compute at each time step, specified as the comma-separated pair consisting of `'Tenor'` and a numeric vector.

`Tenor` enables you to choose a different set of rates to output than the underlying rates. For example, you may want to simulate quarterly data but only report annual rates; this can be done by specifying the optional input `Tenor`.

Data Types: `double`

Output Arguments

`ZeroRates` — Simulated zero-rate term structures
matrix

Simulated zero-rate term structures, returned as a `nPeriods+1`-by-`nTenors`-by-`nTrials` matrix.

`ForwardRates` — Simulated forward-rate term structures
matrix

Simulated zero-rate term structures, returned as a `nPeriods+1`-by-`nTenors`-by-`nTrials` matrix. The `ForwardRates` output is computed using the simulated short rates and by using the model definition to recover the entire yield curve at each simulation date.

Version History

Introduced in R2013a

See Also

`HullWhite1F`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Supported Interest-Rate Instrument Functions” on page 2-3

hwcalbycap

Calibrate Hull-White tree using caps

Syntax

```
[Alpha,Sigma,OptimOut] = hwcalbycap(RateSpec,MarketStrikeMarketMaturity,
MarketVolatility)
[Alpha,Sigma,OptimOut = hwcalbycap(RateSpec,MarketStrikeMarketMaturity,
MarketVolatility,Strike,Settle,Maturity)
[Alpha,Sigma,OptimOut] = hwcalbycap( ___ ,Name,Value)
```

Description

`[Alpha,Sigma,OptimOut] = hwcalbycap(RateSpec,MarketStrikeMarketMaturity, MarketVolatility)` calibrates the `Alpha` (mean reversion) and `Sigma` (volatility) using cap market data and the Hull-White model using the entire cap surface.

The Hull-White calibration functions (`hwcalbycap` and `hwcalbyfloor`) support three models: Black (default), Bachelier or Normal, and Shifted Black. For more information, see the optional arguments for `Shift` and `Model`.

`[Alpha,Sigma,OptimOut = hwcalbycap(RateSpec,MarketStrikeMarketMaturity, MarketVolatility,Strike,Settle,Maturity)` estimates the `Alpha` (mean reversion) and `Sigma` (volatility) using cap market data and the Hull-White model to price a cap at a particular maturity/volatility using the additional optional input arguments for `Strike`, `Settle`, and `Maturity`.

`Strike`, `Settle`, and `Maturity` arguments are specified to calibrate to a specific point on the market volatility surface. If omitted, the calibration is performed across all the market instruments

For an example of calibrating using the Hull-White model with `Strike`, `Settle`, and `Maturity` input arguments, see “Calibrating Hull-White Model Using Market Data” on page 2-92.

`[Alpha,Sigma,OptimOut] = hwcalbycap( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calibrate Hull-White Model from Market Data Using the Entire Cap Volatility Surface

This example shows how to use `hwcalbycap` input arguments for `MarketStrike`, `MarketMaturity`, and `MarketVolatility` to calibrate the HW model using the entire cap volatility surface.

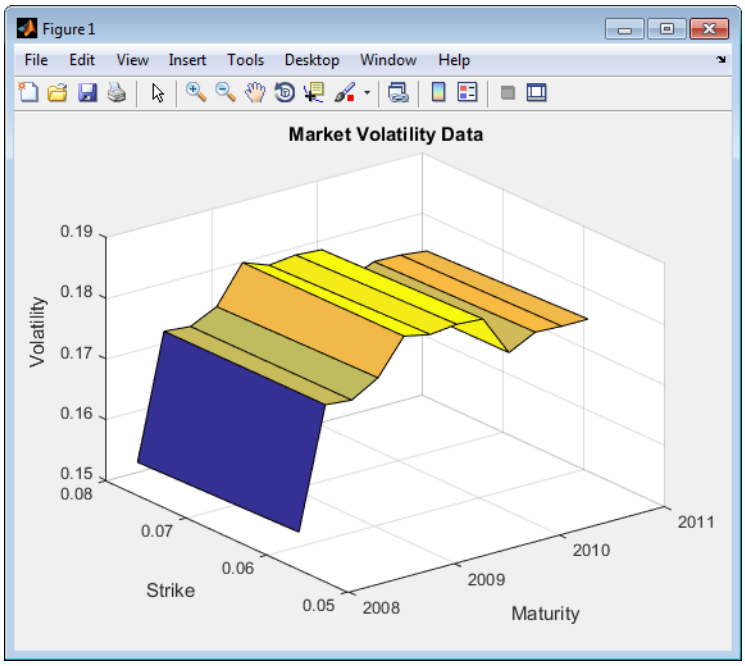
Cap market volatility data covering two strikes over 12 maturity dates.

```
Reset = 4;
MarketStrike = [0.0590; 0.0790];

MarketMaturity = {'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'};
```

```
MarketMaturity = datenum(MarketMaturity);

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

Plot market volatility surface.

```
[AllMaturities,AllStrikes] = meshgrid(MarketMaturity,MarketStrike);
figure;
surf(AllMaturities,AllStrikes,MarketVolaltility)
datetick
xlabel('Maturity')
ylabel('Strike')
zlabel('Volatility')
title('Market Volatility Data')
```



Set interest rate term structure and create a `RateSpec`.

```
Settle = '21-Jan-2008';
Compounding = 4;
Basis = 0;
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];
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'};
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
    'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding,...
    'Basis',Basis)
```

```
RateSpec =

          FinObj: 'RateSpec'
     Compounding: 4
            Disc: [12x1 double]
           Rates: [12x1 double]
       EndTimes: [12x1 double]
     StartTimes: [12x1 double]
       EndDates: [12x1 double]
     StartDates: 733428
```

```
    ValuationDate: 733428
            Basis: 0
     EndMonthRule: 1
```

Calibrate Hull-White model from market data.

```
o = optimoptions('lsqnonlin','TolFun',1e-5,'Display','off');

[Alpha, Sigma] = hwcalbycap(RateSpec, MarketStrike, MarketMaturity,...
    MarketVolaltility, 'Reset', Reset,'Basis', Basis, 'OptimOptions', o)
```

```
Warning: LSQNONLIN did not converge to an optimal solution. It exited with exitflag = 3.

> In hwcalbycapfloor>optimizeOverCapSurface at 232
  In hwcalbycapfloor at 79
  In hwcalbycap at 81

Alpha =

    0.0943


Sigma =

    0.0146
```

Compare with Black prices.

```
BlkPrices = capbyblk(RateSpec,AllStrikes(:), Settle, AllMaturities(:),...
    MarketVolaltility(:),'Reset',Reset,'Basis',Basis);
```

```
BlkPrices =

    0.0604
         0
    0.2729
    0.0006
    0.6498
    0.0412
    1.1121
    0.1426
    1.6426
    0.3131
    2.1869
    0.4998
    2.7056
    0.6894
    3.2124
    0.8815
    3.7311
    1.0686
    4.2246
    1.2790
    4.7027
    1.4810
    5.1877
    1.6919
```

Setup Hull-White tree using calibrated parameters, alpha, and sigma.

```
VolDates    = EndDates;
VolCurve    = Sigma*ones(numel(EndDates),1);
AlphaDates  = EndDates;
AlphaCurve  = Alpha*ones(numel(EndDates),1);
HWVolSpec   = hwvolspec(Settle, VolDates, VolCurve, AlphaDates, AlphaCurve);

HWTimeSpec  = hwtimespec(Settle, EndDates, Compounding);
HWTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec, 'Method', 'HW2000')
```

```
HWTree =

      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 0.6593 1.6612 2.6593 3.6612 4.6593 5.6612 6.6593 7.6612 8.6593 9.6612 10.6593]
        dObs: [733428 733488 733580 733672 733763 733853 733945 734037 734128 734218 734310 734402]
      CFlowT: {1x12 cell}
       Probs: {1x11 cell}
     Connect: {1x11 cell}
     FwdTree: {1x12 cell}
```

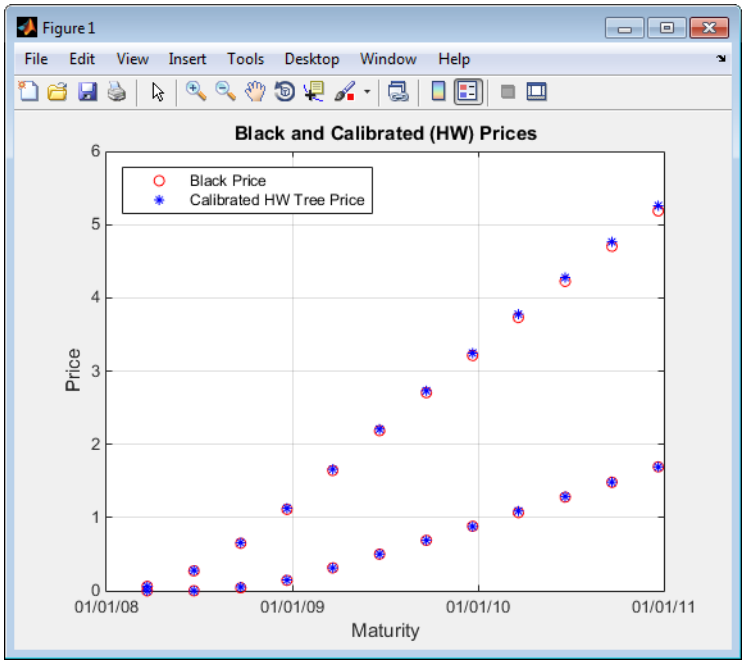
Compute Hull-White prices based on the calibrated tree.

```
HWPrices = capbyhw(HWTree, AllStrikes(:), Settle, AllMaturities(:), Reset, Basis)
```

```
HWPrices =

    0.0601
         0
    0.2788
         0
    0.6580
    0.0518
    1.1254
    0.1485
    1.6591
    0.3123
    2.2076
    0.5022
    2.7319
    0.6883
    3.2459
    0.8774
    3.7771
    1.0900
    4.2769
    1.2875
    4.7645
    1.4845
    5.2572
    1.6921
```

Plot Black prices against the calibrated Hull-White tree prices.

```
figure;
plot(AllMaturities(:), BlkPrices, 'or', AllMaturities(:), HWPrices, '*b');
datetick('x', 2)
xlabel('Maturity');
ylabel('Price');
title('Black and Calibrated (HW) Prices');
legend('Black Price', 'Calibrated HW Tree Price','Location', 'NorthWest');
grid on
```



Calibrating Caplets Using the Normal (Bachelier) Model

This example shows how to use `hwcalbycap` to calibrate market data with the Normal (Bachelier) model to price caplets. Use the Normal (Bachelier) model to perform calibrations when working with negative interest rates, strikes, and normal implied volatilities.

Consider a cap with these parameters:

```
Settle = datetime(2016,12,30);
Maturity = datetime(2019,12,30);
Strike = -0.001075;
Reset = 2;
Principal = 100;
Basis = 0;
```

The caplets and market data for this example are defined as:

```
capletDates = cfdates(Settle, Maturity, Reset, Basis);
datestr(capletDates')
```

```
ans = 6x11 char array
    '30-Jun-2017'
    '30-Dec-2017'
    '30-Jun-2018'
    '30-Dec-2018'
    '30-Jun-2019'
    '30-Dec-2019'
```

```
% Market data information
MarketStrike = [-0.0013; 0];
MarketMat =  [datetime(2017,6,30) ; datetime(2017,12,30) ; datetime(2018,6,30) ; datetime(2018,12
```

```
MarketVol = [0.184 0.2329 0.2398 0.2467 0.2906 0.3348;   % First row in table corresponding to St
             0.217 0.2707 0.2760 0.2814 0.3160 0.3508];  % Second row in table corresponding to S
```

Define the `RateSpec` using `intenvset`.

```
Rates= [-0.002210;-0.002020;-0.00182;-0.001343;-0.001075];
ValuationDate = datetime(2016,12,30);
EndDates =  [datetime(2017,6,30) ; datetime(2017,12,30) ; datetime(2018,6,30) ; datetime(2018,12
Compounding = 2;
Basis = 0;

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

Use `hwcalbycap` to find values for the volatility parameters `Alpha` and `Sigma` using the Normal (Bachelier) model.

```
format short
o=optimoptions('lsqnonlin','TolFun',100*eps);
warning ('off','fininst:hwcalbycapfloor:NoConverge')
[Alpha, Sigma, OptimOut] = hwcalbycap(RateSpec, MarketStrike, MarketMat,...
MarketVol, Strike, Settle, Maturity, 'Reset', Reset, 'Principal', Principal,...
'Basis', Basis, 'OptimOptions', o, 'model', 'normal')
```

```
Local minimum possible.
lsqnonlin stopped because the size of the current step is less than
the value of the step size tolerance.

Alpha = 1.0000e-06

Sigma = 0.3384

OptimOut = struct with fields:
     resnorm: 1.5181e-04
    residual: [5x1 double]
    exitflag: 2
      output: [1x1 struct]
      lambda: [1x1 struct]
    jacobian: [5x2 double]
```

The `OptimOut.residual` field of the `OptimOut` structure is the optimization residual. This value contains the difference between the Normal (Bachelier) caplets and those calculated during the optimization. Use the `OptimOut.residual` value to calculate the percentual difference (error) compared to Normal (Bachelier) caplet prices, and then decide whether the residual is acceptable. There is almost always some residual, so decide if it is acceptable to parameterize the market with a single value of `Alpha` and `Sigma`.

Price the caplets using the market data and Normal (Bachelier) model to obtain the reference caplet values. To determine the effectiveness of the optimization, calculate reference caplet values using the Normal (Bachelier) formula and the market data. Note, you must first interpolate the market data to obtain the caplets for calculation.

```
MarketMatNum = datenum(MarketMat);
[Mats, Strikes] = meshgrid(MarketMatNum, MarketStrike);
FlatVol = interp2(Mats, Strikes, MarketVol, datenum(Maturity), Strike, 'spline');
```

```
[CapPrice, Caplets] = capbynormal(RateSpec, Strike, Settle, Maturity, FlatVol,...
'Reset', Reset, 'Basis', Basis, 'Principal', Principal);
Caplets = Caplets(2:end)'
```

```
Caplets = 5×1

    4.7392
    6.7799
    8.2609
    9.6136
   10.6455
```

Compare the optimized values and Normal (Bachelier) values, and display the results graphically. After calculating the reference values for the caplets, compare the values analytically and graphically to determine whether the calculated single values of `Alpha` and `Sigma` provide an adequate approximation.

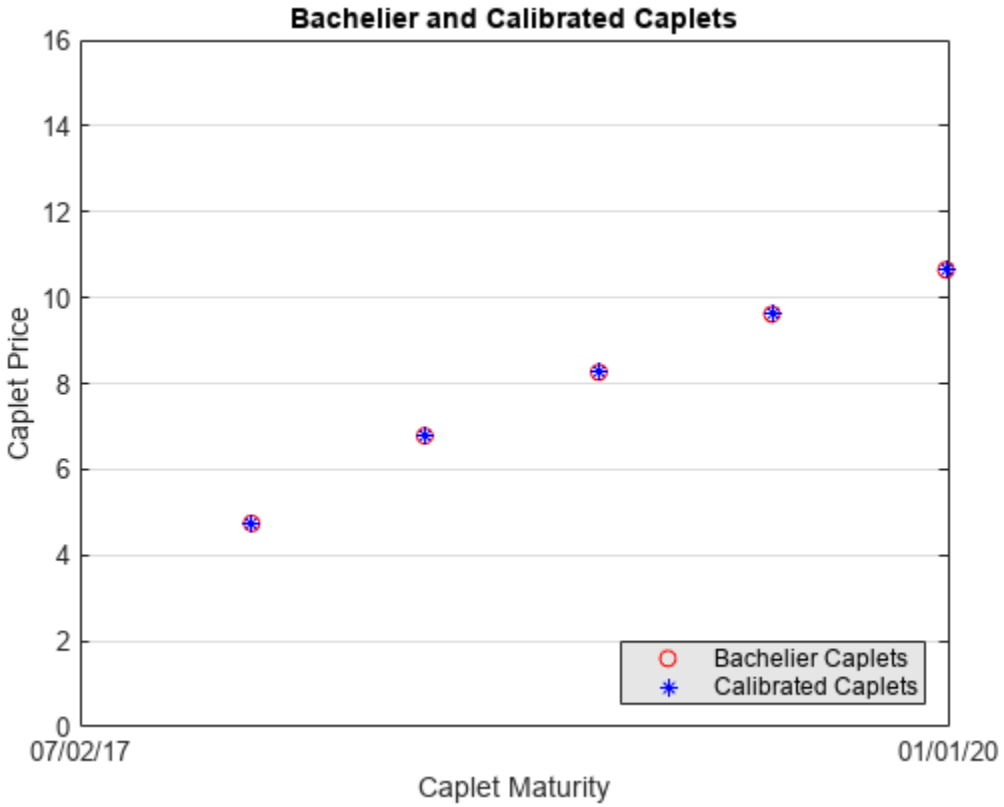
```
OptimCaplets = Caplets+OptimOut.residual;

disp('   ');



disp(' Bachelier   Calibrated Caplets');
```

```
 Bachelier   Calibrated Caplets
```

```
disp([Caplets        OptimCaplets])
```

```
    4.7392    4.7453
    6.7799    6.7851
    8.2609    8.2657
    9.6136    9.6112
   10.6455   10.6379
```

```
plot(MarketMatNum(2:end), Caplets, 'or', MarketMatNum(2:end), OptimCaplets, '*b');
datetick('x', 2)
xlabel('Caplet Maturity');
ylabel('Caplet Price');
ylim ([0 16]);
title('Bachelier and Calibrated Caplets');
h = legend('Bachelier Caplets', 'Calibrated Caplets');
set(h, 'color', [0.9 0.9 0.9]);
set(h, 'Location', 'SouthEast');
set(gcf, 'NumberTitle', 'off')
grid on
```



Input Arguments

`RateSpec` — Interest-rate specification for initial rate curve

structure

Interest-rate specification for initial rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`MarketStrike` — Market cap strike

vector

Market cap strike, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`MarketMaturity` — Market cap maturity date

datetime array | string array | date character vector

Market cap maturity dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `hwcalbycap` also accepts serial date numbers as inputs, but they are not recommended.

`MarketVolatility` — Market flat volatilities
matrix

Market flat volatilities, specified as a `NSTRIKES`-by-`NMATS` matrix of market flat volatilities, where `NSTRIKES` is the number of caplet strikes from `MarketStrike` and `NMATS` is the caplet maturity dates from `MarketMaturity`.

Data Types: `double`

`Strike` — Rate at which cap is exercised
decimal scalar

(Optional) Rate at which the cap is exercised, specified as a decimal scalar value.

Data Types: `single`

`Settle` — Settlement date of the cap
datetime scalar | string scalar | date character vector

(Optional) Settlement date of the cap, specified as a datetime, string, or data character vector.

To support existing code, `hwcalbycap` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date of the cap
datetime scalar | string scalar | date character vector

(Optional) Maturity date of the cap, specified as scalar datetime, string, or data character vector.

To support existing code, `hwcalbycap` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Alpha,Sigma,OptimOut] = hwcalbycap(RateSpec,MarketStrike,MarketMaturity,MarketVolaltility,'Reset',2,'Principal',100000,'Basis',3,'OptimOptions',o)`

`Reset` — Frequency of payments per year
`1` (default) | numeric

Frequency of payments per year, specified as the comma-separated pair consisting of `'Reset'` and a scalar numeric value.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | nonnegative integer

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a scalar nonnegative integer.

Data Types: `double`

`Basis` — Day-count basis used when annualizing the input forward rate
`0` (actual/actual) (default) | integers of the set `[0...13]`

Day-count basis used when annualizing the input forward rate, specified as the comma-separated pair consisting of `'Basis'` and a scalar value. Values are:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`LB` — Lower bounds
`[0;0]` (default) | numeric vector

Lower bounds, specified as the comma-separated pair consisting of `'LB'` and a 2-by-1 vector of the lower bounds, defined as `[LBSigma; LBAlpha]`, used in the search algorithm function. For more information, see `lsqnonlin`.

Data Types: `double`

`UB` — Upper bounds
`[ ]` (unbound) (default) | numeric vector

Upper bounds, specified as the comma-separated pair consisting of `'UB'` and a 2-by-1 vector of the upper bounds, defined as `[UBSigma; LBAlpha]`, used in the search algorithm function. For more information, see `lsqnonlin`.

Data Types: `double`

`X0` — Initial values
`[0.5;0.5]` (default) | numeric vector

Initial values, specified as the comma-separated pair consisting of `'X0'` and a 2-by-1 vector of the initial values, defined as `[Sigma0; Alpha0]`, used in the search algorithm function. For more information, see `lsqnonlin`.

Data Types: `double`

`OptimOptions` — Optimization parameters
structure

Optimization parameters, specified as the comma-separated pair consisting of `'OptimOptions'` and a structure defined by using `optimoptions`.

Data Types: `struct`

`Shift` — Shift in decimals for shifted Black model
`0` (no shift) (default) | positive decimal

Shift in decimals for the shifted Black model, specified as the comma-separated pair consisting of `'Shift'` and a scalar positive decimal value. Set this parameter to a positive shift in decimals to add a positive shift to forward rate and `Strike`, which effectively sets a negative lower bound for forward rate and `Strike`. For example, a `Shift` value of `0.01` is equal to a 1% shift.

Data Types: `double`

`Model` — Indicator for model used for calibration routine
`lognormal` (Black model) (default) | values are `normal` and `lognormal`

Indicator for model used for calibration routine, specified as the comma-separated pair consisting of `'Model'` and a scalar character vector with a value of `normal` or `lognormal`.

Data Types: `char`

Output Arguments

`Alpha` — Mean reversion value obtained from calibrating the cap using market information
scalar numeric

Mean reversion value obtained from calibrating the cap using market information, returned as a scalar value.

`Sigma` — Volatility value obtained from calibrating cap using market information
scalar numeric

Volatility value obtained from calibrating the cap using market information, returned as a scalar.

`OptimOut` — Optimization results
numeric structure

Optimization results, returned as a structure.

Version History

Introduced in R2009a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `hwcalbycap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`capbyblk` | `hwcalbyfloor` | `hwtree` | `lsqnonlin` | `HullWhite1F`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Supported Interest-Rate Instrument Functions” on page 2-3

hwcalbyfloor

Calibrate Hull-White tree using floors

Syntax

```
[Alpha,Sigma,OptimOut] = hwcalbyfloor(RateSpec,MarketStrikeMarketMaturity,
MarketVolatility)
[Alpha,Sigma,OptimOut = hwcalbyfloor(RateSpec,MarketStrikeMarketMaturity,
MarketVolatility,Strike,Settle,Maturity)
[Alpha,Sigma,OptimOut] = hwcalbyfloor( ___ ,Name,Value)
```

Description

`[Alpha,Sigma,OptimOut] = hwcalbyfloor(RateSpec,MarketStrikeMarketMaturity, MarketVolatility)` calibrates the `Alpha` (mean reversion) and `Sigma` (volatility) using floor market data and the Hull-White model using the entire floor surface.

The Hull-White calibration functions (`hwcalbyfloor` and `hwcalbycap`) support three models: Black (default), Bachelier or Normal, and Shifted Black. For more information, see the optional arguments for `Shift` and `Model`.

`[Alpha,Sigma,OptimOut = hwcalbyfloor(RateSpec,MarketStrikeMarketMaturity, MarketVolatility,Strike,Settle,Maturity)` estimates the `Alpha` (mean reversion) and `Sigma` (volatility) using floor market data and the Hull-White model to price a floor at a particular maturity/volatility using the additional optional input arguments for `Strike`, `Settle`, and `Maturity`.

`Strike`, `Settle`, and `Maturity` arguments are specified to calibrate to a specific point on the market volatility surface. If omitted, the calibration is performed across all the market instruments

For an example of calibrating using the Hull-White model with `Strike`, `Settle`, and `Maturity` input arguments, see “Calibrating Hull-White Model Using Market Data” on page 2-92.

`[Alpha,Sigma,OptimOut] = hwcalbyfloor( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calibrate Hull-White Model from Market Data Using the Entire Floor Volatility Surface

This example shows how to use `hwcalbyfloor` input arguments for `MarketStrike`, `MarketMaturity`, and `MarketVolatility` to calibrate the HW model using the entire floor volatility surface.

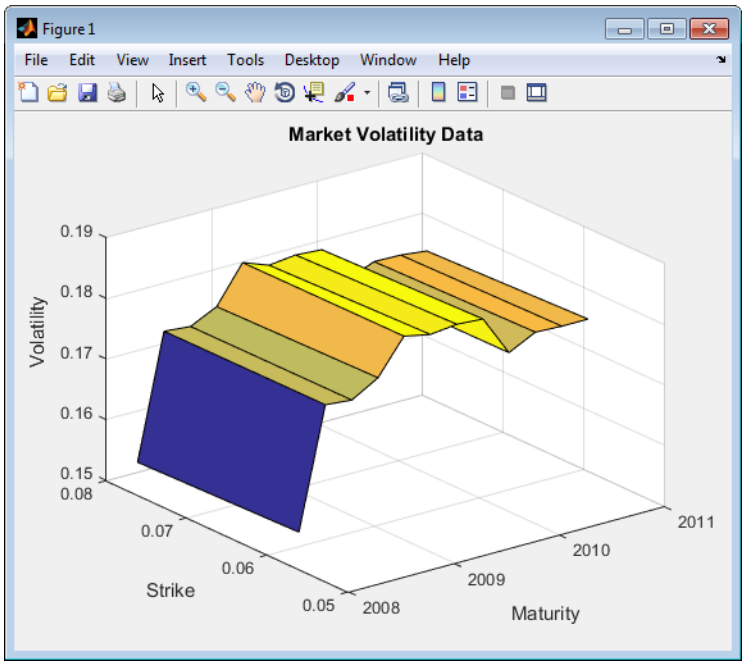
Floor market volatility data covering two strikes over 12 maturity dates.

```
Reset = 4;
MarketStrike = [0.0590; 0.0790];

MarketMaturity = {'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'};
```

```
MarketMaturity = datenum(MarketMaturity);

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

Plot market volatility surface.

```
[AllMaturities,AllStrikes] = meshgrid(MarketMaturity,MarketStrike);
figure;
surf(AllMaturities,AllStrikes,MarketVolaltility)
datetick
xlabel('Maturity')
ylabel('Strike')
zlabel('Volatility')
title('Market Volatility Data')
```



Set interest rate term structure and create a `RateSpec`.

```
Settle = '21-Jan-2008';
Compounding = 4;
Basis = 0;
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];
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'};
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
    'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding,...
    'Basis',Basis)
```

```
RateSpec =

          FinObj: 'RateSpec'
     Compounding: 4
            Disc: [12x1 double]
           Rates: [12x1 double]
       EndTimes: [12x1 double]
     StartTimes: [12x1 double]
       EndDates: [12x1 double]
     StartDates: 733428
```

```
    ValuationDate: 733428
            Basis: 0
     EndMonthRule: 1
```

Calibrate Hull-White model from market data.

```
o = optimoptions('lsqnonlin','TolFun',1e-5,'Display','off');

[Alpha, Sigma] = hwcalbyfloor(RateSpec, MarketStrike, MarketMaturity,...
    MarketVolaltility, 'Reset', Reset,'Basis', Basis, 'OptimOptions', o)
```

```
Warning: LSQNONLIN did not converge to an optimal solution. It exited with exitflag = 3.

> In hwcalbycapfloor>optimizeOverCapSurface at 232
  In hwcalbycapfloor at 79
  In hwcalbyfloor at 81

Alpha =

    0.0835


Sigma =

    0.0145
```

Compare with Black prices.

```
BlkPrices = floorbyblk(RateSpec,AllStrikes(:), Settle, AllMaturities(:),...
    MarketVolaltility(:),'Reset',Reset,'Basis',Basis)
```

```
BlkPrices =

         0
    0.2659
    0.0010
    0.5426
    0.0021
    0.6841
    0.0042
    0.7947
    0.0081
    0.8970
    0.0128
    0.9947
    0.0217
    1.1145
    0.0340
    1.2448
    0.0402
    1.3415
    0.0610
    1.4947
    0.0827
    1.6458
    0.1071
    1.7951
```

Setup Hull-White tree using calibrated parameters, alpha, and sigma.

```
VolDates    = EndDates;
VolCurve    = Sigma*ones(numel(EndDates),1);
AlphaDates  = EndDates;
AlphaCurve  = Alpha*ones(numel(EndDates),1);
HWVolSpec   = hwvolspec(Settle, VolDates, VolCurve, AlphaDates, AlphaCurve);
```

```
HWTimeSpec  = hwtimespec(Settle, EndDates, Compounding);
HWTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec, 'Method', 'HW2000')
```

```
HWTree =

      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 0.6593 1.6612 2.6593 3.6612 4.6593 5.6612 6.6593 7.6612 8.6593 9.6612 10.6593]
        dObs: [733428 733488 733580 733672 733763 733853 733945 734037 734128 734218 734310 734402]
      CFlowT: {1x12 cell}
       Probs: {1x11 cell}
     Connect: {1x11 cell}
     FwdTree: {1x12 cell}
```

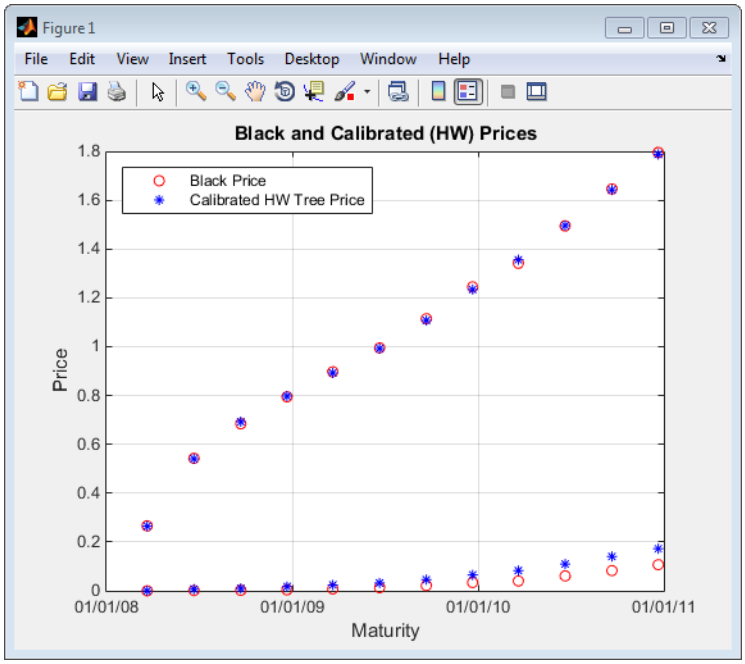
Compute Hull-White prices based on the calibrated tree.

```
HWPrices = floorbyhw(HWTree, AllStrikes(:), Settle, AllMaturities(:), Reset, Basis)
```

```
HWPrices =

         0
    0.2644
    0.0067
    0.5404
    0.0101
    0.6924
    0.0169
    0.7974
    0.0236
    0.8919
    0.0320
    0.9919
    0.0460
    1.1074
    0.0649
    1.2340
    0.0829
    1.3558
    0.1096
    1.4957
    0.1406
    1.6418
    0.1724
    1.7877
```

Plot Black prices against the calibrated Hull-White tree prices.

```
figure;
plot(AllMaturities(:), BlkPrices, 'or', AllMaturities(:), HWPrices, '*b');
datetick('x', 2)
xlabel('Maturity');
ylabel('Price');
title('Black and Calibrated (HW) Prices');
legend('Black Price', 'Calibrated HW Tree Price','Location', 'NorthWest');
grid on
```



Calibrating Floorlets Using the Normal (Bachelier) Model

This example shows how to use `hwcalbyfloor` to calibrate market data with the Normal (Bachelier) model to price floorlets. Use the Normal (Bachelier) model to perform calibrations when working with negative interest rates, strikes, and normal implied volatilities.

Consider a floor with these parameters:

```
Settle = datetime(2016,12,30);
Maturity = datetime(2019,12,30);
Strike = -0.004075;
Reset = 2;
Principal = 100;
Basis = 0;
```

The floorlets and market data for this example are defined as:

```
floorletDates = cfdates(Settle, Maturity, Reset, Basis);
datestr(floorletDates')
```

```
ans = 6x11 char array
    '30-Jun-2017'
    '30-Dec-2017'
    '30-Jun-2018'
    '30-Dec-2018'
    '30-Jun-2019'
    '30-Dec-2019'
```

```
% Market data information
MarketStrike = [-0.00595; 0];
```

```
MarketMat = [datetime(2017,6,30) ; datetime(2017,12,30) ; datetime(2018,6,30) ; datetime(2018,12
MarketVol = [0.184 0.2329 0.2398 0.2467 0.2906 0.3348;   % First row in table corresponding to St
             0.217 0.2707 0.2760 0.2814 0.3160 0.3508];  % Second row in table corresponding to S
```

Define the `RateSpec` using `intenvset`.

```
Rates= [-0.003210;-0.003020;-0.00182;-0.001343;-0.001075];
ValuationDate = datetime(2016,12,30);
EndDates = [datetime(2017,6,30) ; datetime(2017,12,30) ; datetime(2018,6,30) ; datetime(2018,12,3
Compounding = 2;
Basis = 0;

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

Use `hwcalbyfloor` to find values for the volatility parameters `Alpha` and `Sigma` using the Normal (Bachelier) model.

```
format short
o=optimoptions('lsqnonlin','TolFun',100*eps);
warning ('off','fininst:hwcalbycapfloor:NoConverge')
[Alpha, Sigma, OptimOut] = hwcalbyfloor(RateSpec, MarketStrike, MarketMat,...
MarketVol, Strike, Settle, Maturity, 'Reset', Reset, 'Principal', Principal,...
'Basis', Basis, 'OptimOptions', o, 'model', 'normal')
```

```
Local minimum possible.
lsqnonlin stopped because the size of the current step is less than
the value of the step size tolerance.

Alpha = 1.0000e-06

Sigma = 0.3410

OptimOut = struct with fields:
     resnorm: 1.9233e-04
    residual: [5x1 double]
    exitflag: 2
      output: [1x1 struct]
      lambda: [1x1 struct]
    jacobian: [5x2 double]
```

The `OptimOut.residual` field of the `OptimOut` structure is the optimization residual. This value contains the difference between the Normal (Bachelier) floorlets and those calculated during the optimization. Use the `OptimOut.residual` value to calculate the percentual difference (error) compared to Normal (Bachelier) floorlet prices, and then decide whether the residual is acceptable. There is almost always some residual, so decide if it is acceptable to parameterize the market with a single value of `Alpha` and `Sigma`.

Price the floorlets using the market data and Normal (Bachelier) model to obtain the reference floorlet values. To determine the effectiveness of the optimization, calculate reference floorlet values using the Normal (Bachelier) formula and the market data. Note, you must first interpolate the market data to obtain the floorlets for calculation.

```
MarketMatNum = datenum(MarketMat);
[Mats, Strikes] = meshgrid(MarketMatNum, MarketStrike);
FlatVol = interp2(Mats, Strikes, MarketVol, datenum(Maturity), Strike, 'spline');
```

```
[FloorPrice, Floorlets] = floorbynormal(RateSpec, Strike, Settle, Maturity, FlatVol,...
'Reset', Reset, 'Basis', Basis, 'Principal', Principal);
Floorlets = Floorlets(2:end)'
```

```
Floorlets = 5×1

    4.7637
    6.7180
    8.1833
    9.5825
   10.6090
```

Compare the optimized values and Normal (Bachelier) values, and display the results graphically. After calculating the reference values for the floorlets, compare the values analytically and graphically to determine whether the calculated single values of `Alpha` and `Sigma` provide an adequate approximation.

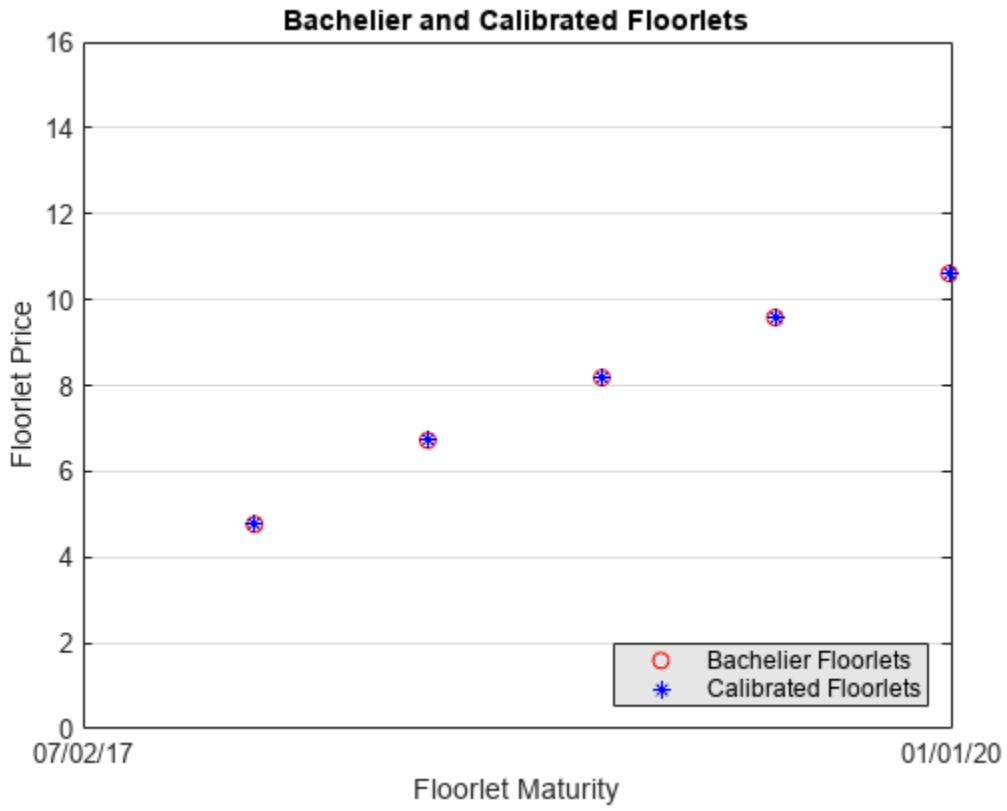
```
OptimFloorlets = Floorlets+OptimOut.residual;

disp('   ');


disp(' Bachelier   Calibrated Floorlets');
```

```
 Bachelier   Calibrated Floorlets
```

```
disp([Floorlets        OptimFloorlets])
```

```
    4.7637    4.7685
    6.7180    6.7263
    8.1833    8.1878
    9.5825    9.5795
   10.6090   10.6007
```

```
plot(MarketMatNum(2:end), Floorlets, 'or', MarketMatNum(2:end), OptimFloorlets, '*b');
datetick('x', 2)
xlabel('Floorlet Maturity');
ylabel('Floorlet Price');
ylim ([0 16]);
title('Bachelier and Calibrated Floorlets');
h = legend('Bachelier Floorlets', 'Calibrated Floorlets');
set(h, 'color', [0.9 0.9 0.9]);
set(h, 'Location', 'SouthEast');
set(gcf, 'NumberTitle', 'off')
grid on
```



Input Arguments

`RateSpec` — Interest-rate specification for initial rate curve

structure

Interest-rate specification for initial rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`MarketStrike` — Market floor strike

vector

Market floor strike, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`MarketMaturity` — Market floor maturity date

datetime array | string array | date character vector

Market floor maturity dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `hwcalbyfloor` also accepts serial date numbers as inputs, but they are not recommended.

`MarketVolatility` — Market flat volatilities
matrix

Market flat volatilities, specified as a `NSTRIKES`-by-`NMATS` matrix of market flat volatilities, where `NSTRIKES` is the number of caplet strikes from `MarketStrike` and `NMATS` is the caplet maturity dates from `MarketMaturity`.

Data Types: `double`

`Strike` — Rate at which floor is exercised
decimal scalar

(Optional) Rate at which the floor is exercised, specified as a decimal scalar value.

Data Types: `double`

`Settle` — Settlement date of the floor
datetime scalar | string scalar | date character vector

(Optional) Settlement date of the floor, specified as a scalar datetime, string, or data character vector.

To support existing code, `hwcalbyfloor` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date of the floor
datetime scalar | string scalar | date character vector

(Optional) Maturity date of the floor, specified as scalar datetime, string, or data character vector.

To support existing code, `hwcalbyfloor` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Alpha,Sigma,OptimOut] = hwcalbyfloor(RateSpec,MarketStrike,MarketMaturity,MarketVolaltility,'Reset',2,'Principal',100000,'Basis',3,'OptimOptions',o)`

`Reset` — Frequency of payments per year
`1` (default) | numeric

Frequency of payments per year, specified as the comma-separated pair consisting of `'Reset'` and a scalar numeric value.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | nonnegative integer

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a scalar nonnegative integer.

Data Types: `double`

`Basis` — Day-count basis used when annualizing the input forward rate
`0` (actual/actual) (default) | integers of the set `[0...13]`

Day-count basis used when annualizing the input forward rate, specified as the comma-separated pair consisting of `'Basis'` and a scalar value. Values are:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`LB` — Lower bounds
`[0;0]` (default) | numeric vector

Lower bounds, specified as the comma-separated pair consisting of `'LB'` and a 2-by-1 vector of the lower bounds, defined as `[LBSigma; LBAlpha]`, used in the search algorithm function. For more information, see `lsqnonlin`.

Data Types: `double`

`UB` — Upper bounds
`[ ]` (unbound) (default) | numeric vector

Upper bounds, specified as the comma-separated pair consisting of `'UB'` and a 2-by-1 vector of the upper bounds, defined as `[UBSigma; LBAlpha]`, used in the search algorithm function. For more information, see `lsqnonlin`.

Data Types: `double`

`X0` — Initial values
`[0.5;0.5]` (default) | numeric vector

Initial values, specified as the comma-separated pair consisting of `'X0'` and a 2-by-1 vector of the initial values, defined as `[Sigma0; Alpha0]`, used in the search algorithm function. For more information, see `lsqnonlin`.

Data Types: `double`

`OptimOptions` — Optimization parameters
structure

Optimization parameters, specified as the comma-separated pair consisting of `'OptimOptions'` and a structure defined by using `optimoptions`.

Data Types: `struct`

`Shift` — Shift in decimals for shifted Black model
`0` (no shift) (default) | positive decimal

Shift in decimals for the shifted Black model, specified as the comma-separated pair consisting of `'Shift'` and a scalar positive decimal value. Set this parameter to a positive shift in decimals to add a positive shift to forward rate and `Strike`, which effectively sets a negative lower bound for forward rate and `Strike`. For example, a `Shift` value of `0.01` is equal to a 1% shift.

Data Types: `double`

`Model` — Indicator for model used for calibration routine
`lognormal` (Black model) (default) | values are `normal` and `lognormal`

Indicator for model used for calibration routine, specified as the comma-separated pair consisting of `'Model'` and a scalar character vector with a value of `normal` or `lognormal`.

Data Types: `char`

Output Arguments

`Alpha` — Mean reversion value obtained from calibrating the floor using market information
scalar numeric

Mean reversion value obtained from calibrating the floor using market information, returned as a scalar value.

`Sigma` — Volatility value obtained from calibrating floor using market information
scalar numeric

Volatility value obtained from calibrating the floor using market information, returned as a scalar.

`OptimOut` — Optimization results
numeric structure

Optimization results, returned as a structure.

Version History

Introduced in R2009a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `hwcalbyfloor` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`floorbyblk` | `hwcalbycap` | `hwtree` | `lsqnonlin` | `HullWhite1F`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Supported Interest-Rate Instrument Functions” on page 2-3

hwprice

Instrument prices from Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = hwprice(HWTree,InstSet)
[Price,PriceTree] = hwprice( ___ ,Options)
```

Description

`[Price,PriceTree] = hwprice(HWTree,InstSet)` 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.

`hwprice` handles instrument types: `'Bond'`, `'CashFlow'`, `'OptBond'`, `'OptEmBond'`, `'OptEmBond'`, `'OptFloat'`, `'OptEmFloat'`, `'Fixed'`, `'Float'`, `'Cap'`, `'Floor'`, `'RangeFloat'`, `'Swap'`. See `instadd` to construct defined types.

`[Price,PriceTree] = hwprice( ___ ,Options)` adds an optional input argument for `Options`.

Examples

Price the Cap and Bond Instruments Contained in an Instrument Set

Load the HW tree and instruments from the data file `deriv.mat`.

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

instdisp(HWSubSet)
```

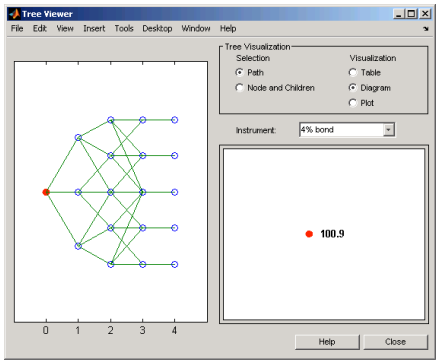
```
instdisp(HWSubSet)
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCouponDate LastCouponDate StartDate Face Na
1     Bond 0.04       01-Jan-2004    01-Jan-2007    1      0     1            NaN       NaN             NaN            NaN       100  4%
2     Bond 0.04       01-Jan-2004    01-Jan-2008    1      0     1            NaN       NaN             NaN            NaN       100  4%

Index Type Strike Settle         Maturity       CapReset Basis Principal Name   Quantity
3     Cap  0.06   01-Jan-2004    01-Jan-2008    1        0     100       6% Cap 10
```

Price the cap and bond instruments.

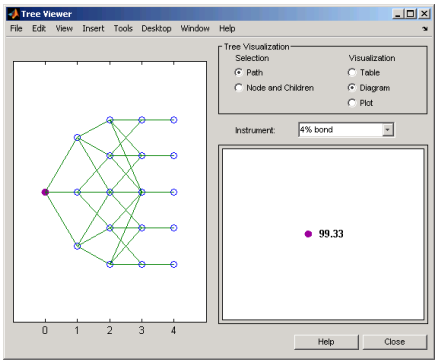
```
[Price, PriceTree] = hwprice(HWTree, HWSubSet);
```

```
100.9188
 99.3296
  0.5837
```

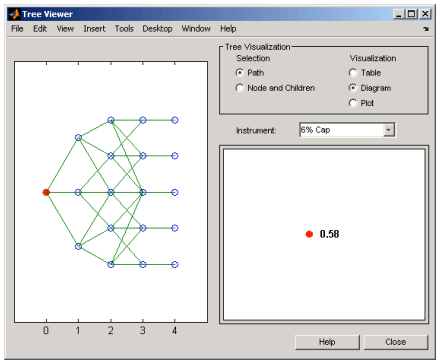
You can use `treeviewer` to see the prices of these three instruments along the price tree.



First 4% Bond (Maturity 2007)



Second 4% Bond (Maturity 2008)



6% Cap

Price Multi-Stepped Coupon Bonds

The data for the interest-rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2010';
StartDates = ValuationDate;
EndDates = {'Jan-1-2011'; 'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'};
Compounding = 1;
```

Create the `RateSpec`.

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

```
RS = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
       StartDates: 734139
    ValuationDate: 734139
```

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

Create a portfolio of stepped coupon bonds with different maturities.

```
Settle = '01-Jan-2010';
Maturity = {'01-Jan-2011';'01-Jan-2012';'01-Jan-2013';'01-Jan-2014'};
CouponRate = {{'01-Jan-2011' .042;'01-Jan-2012' .05; '01-Jan-2013' .06; '01-Jan-2014' .07}};

ISet = instbond(CouponRate, Settle, Maturity, 1);
instdisp(ISet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond [Cell]     01-Jan-2010    01-Jan-2011    1      0     1            NaN       NaN
2     Bond [Cell]     01-Jan-2010    01-Jan-2012    1      0     1            NaN       NaN
3     Bond [Cell]     01-Jan-2010    01-Jan-2013    1      0     1            NaN       NaN
4     Bond [Cell]     01-Jan-2010    01-Jan-2014    1      0     1            NaN       NaN
```

Build the tree with the following data:

```
VolDates = ['1-Jan-2011'; '1-Jan-2012'; '1-Jan-2013'; '1-Jan-2014'];
VolCurve = 0.01;
AlphaDates = '01-01-2014';
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RS.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RS, HWTimeSpec)
```

```
HWT = struct with fields:
      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734139 734504 734869 735235]
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
     FwdTree: {1x4 cell}
```

Compute the price of the stepped coupon bonds.

```
PHW = hwprice(HWT, ISet)
```

```
PHW = 4×1

  100.6763
  100.7368
  100.9266
  101.0115
```

Price a Portfolio of Stepped Callable Bonds and Stepped Vanilla Bonds

The data for the interest-rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2010';
StartDates = ValuationDate;
EndDates = {'Jan-1-2011'; 'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'};
Compounding = 1;
```

Create a RateSpec.

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

```
RS = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
          EndTimes: [4x1 double]
        StartTimes: [4x1 double]
          EndDates: [4x1 double]
        StartDates: 734139
    ValuationDate: 734139
            Basis: 0
     EndMonthRule: 1
```

Create an instrument portfolio of three stepped callable bonds and three stepped vanilla bonds.

```
Settle = '01-Jan-2010';
Maturity = {'01-Jan-2012';'01-Jan-2013';'01-Jan-2014'};
CouponRate = {{'01-Jan-2011' .042;'01-Jan-2012' .05; '01-Jan-2013' .06; '01-Jan-2014' .07}};
OptSpec='call';
Strike=100;
ExerciseDates='01-Jan-2011'; %Callable in one year
```

Bonds with embedded option.

```
ISet = instoptembnd(CouponRate, Settle, Maturity, OptSpec, Strike,...
ExerciseDates, 'Period', 1);
```

Vanilla bonds.

```
ISet = instbond(ISet, CouponRate, Settle, Maturity, 1);
```

Display the instrument portfolio.

```
instdisp(ISet)
```

```
Index Type      CouponRate Settle         Maturity       OptSpec Strike ExerciseDates  Period Bas
1     OptEmBond [Cell]     01-Jan-2010    01-Jan-2012    call    100    01-Jan-2011    1      0
2     OptEmBond [Cell]     01-Jan-2010    01-Jan-2013    call    100    01-Jan-2011    1      0
3     OptEmBond [Cell]     01-Jan-2010    01-Jan-2014    call    100    01-Jan-2011    1      0

Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCoup
4     Bond [Cell]     01-Jan-2010    01-Jan-2012    1      0     1            NaN       NaN
5     Bond [Cell]     01-Jan-2010    01-Jan-2013    1      0     1            NaN       NaN
```

```
6     Bond [Cell]     01-Jan-2010    01-Jan-2014    1      0     1            NaN       NaN
```

Build the tree with the following data:

```
VolDates = ['1-Jan-2011'; '1-Jan-2012'; '1-Jan-2013'; '1-Jan-2014'];
VolCurve = 0.01;
AlphaDates = '01-01-2014';
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RS.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RS, HWTimeSpec)
```

```
HWT = struct with fields:
      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734139 734504 734869 735235]
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
     FwdTree: {1x4 cell}
```

Compute the price of the stepped callable bonds and the stepped vanilla bonds.

```
PHW = hwprice(HWT, ISet)
```

```
PHW = 6×1

  100.4089
  100.2043
  100.0197
  100.7368
  100.9266
  101.0115
```

The first three rows correspond to the price of the stepped callable bonds and the last three rows correspond to the price of the stepped vanilla bonds.

Compute the Price of a Portfolio of Instruments

The data for the interest-rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2011';
StartDates = ValuationDate;
EndDates = {'Jan-1-2012'; 'Jan-1-2013'; 'Jan-1-2014'; 'Jan-1-2015'};
Compounding = 1;
```

Create a `RateSpec`.

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

RS = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
       StartDates: 734504
    ValuationDate: 734504
            Basis: 0
     EndMonthRule: 1
```

Create an instrument portfolio with two range notes and a floating rate note with the following data:

```
Spread = 200;
Settle = 'Jan-1-2011';
Maturity = 'Jan-1-2014';

% First Range Note
RateSched(1).Dates = {'Jan-1-2012'; 'Jan-1-2013'  ; 'Jan-1-2014'};
RateSched(1).Rates  = [0.045 0.055; 0.0525  0.0675; 0.06 0.08];

% Second Range Note
RateSched(2).Dates = {'Jan-1-2012'; 'Jan-1-2013' ; 'Jan-1-2014'};
RateSched(2).Rates  = [0.048 0.059; 0.055  0.068 ; 0.07 0.09];
```

Create `InstSet`, add a floating-rate note, and display the portfolio instruments.

```
InstSet = instadd('RangeFloat', Spread, Settle, Maturity, RateSched);

% Add a floating-rate note
InstSet = instadd(InstSet, 'Float', Spread, Settle, Maturity);

% Display the portfolio instrument
instdisp(InstSet)

Index Type       Spread Settle         Maturity       RateSched FloatReset Basis Principal EndMo
1     RangeFloat 200    01-Jan-2011    01-Jan-2014    [Struct]  1          0     100       1
2     RangeFloat 200    01-Jan-2011    01-Jan-2014    [Struct]  1          0     100       1

Index Type  Spread Settle         Maturity       FloatReset Basis Principal EndMonthRule CapRate
3     Float 200    01-Jan-2011    01-Jan-2014    1          0     100       1            Inf
```

The data to build the tree is as follows:

```
VolDates = ['1-Jan-2012'; '1-Jan-2013'; '1-Jan-2014';'1-Jan-2015'];
VolCurve = 0.01;
AlphaDates = '01-01-2015';
AlphaCurve = 0.1;

HWVS = hwvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
```

```
HWTS = hwtimespec(RS.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVS, RS, HWTS)
```

```
HWT = struct with fields:
      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734504 734869 735235 735600]
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
     FwdTree: {1x4 cell}
```

Price the portfolio.

```
Price = hwprice(HWT, InstSet)
```

```
Price = 3×1

   99.3327
   98.1580
  105.5147
```

Create a Float-Float Swap and Price with `hwprice`

Use `instswap` to create a float-float swap and price the swap with `hwprice`.

```
RateSpec = intenvset('Rates',.05,'StartDate',today,'EndDate',datemnth(today,60));
IS = instswap([.02 .03],today,datemnth(today,60),[], [], [], [1 1]);
VolSpec = hwvolspec(today,datemnth(today,60),.01,datemnth(today,60),.1);
TimeSpec = hwtimespec(today,cfdates(today,datemnth(today,60),1));
HWTree = hwtree(VolSpec,RateSpec,TimeSpec);
hwprice(HWTree,IS)
```

```
ans = -4.3220
```

Price Multiple Swaps with `hwprice`

Use `instswap` to create multiple swaps and price the swaps with `hwprice`.

```
RateSpec = intenvset('Rates',.05,'StartDate',today,'EndDate',datemnth(today,60));
IS = instswap([.03 .02],today,datemnth(today,60),[], [], [], [1 1]);
IS = instswap(IS,[200 300],today,datemnth(today,60),[], [], [], [0 0]);
IS = instswap(IS,[.08 300],today,datemnth(today,60),[], [], [], [1 0]);
VolSpec = hwvolspec(today,datemnth(today,60),.01,datemnth(today,60),.1);
TimeSpec = hwtimespec(today,cfdates(today,datemnth(today,60),1));
HWTree = hwtree(VolSpec,RateSpec,TimeSpec);
hwprice(HWTree,IS)
```

```
ans = 3×1

    4.3220
   -4.3220
   -0.2701
```

Input Arguments

`HWTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hwtree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

(Optional) Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Price for each instrument

vector

Price for each instrument, returned as a `NINST`-by-`1` vector. The prices are computed by backward dynamic programming on the interest-rate tree. If an instrument cannot be priced, a `NaN` is returned in that entry.

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.

- `optfloatbybdt`: Price a floating-rate note with an option from a Hull-White tree.
- `optemfloatbybdt`: Price a floating-rate note with an embedded option from a Hull-White tree.
- `rangefloatbyhw`: Price range floating note using a Hull-White tree.
- `swapbyhw`: Price a swap from a Hull-White tree.
- `swaptionbyhw`: Price a swaption from a Hull-White tree.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

Version History

Introduced before R2006a

See Also

`hwsens` | `hwtree` | `instadd` | `intenvprice` | `intenvsens`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

hwsens

Instrument prices and sensitivities from Hull-White interest-rate tree

Syntax

```
[Delta,Gamma,Vega,Price] = hwsens(HWTree,InstSet)
[Delta,Gamma,Vega,Price] = hwsens( ___ ,Options)
```

Description

`[Delta,Gamma,Vega,Price] = hwsens(HWTree,InstSet)` computes instrument sensitivities and prices for instruments using an interest-rate tree created with the `hwtree` function. All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide by the respective instrument price.

`hwsens` handles instrument types: `'Bond'`, `'CashFlow'`, `'OptBond'`, `'OptEmBond'`, `'OptEmBond'`, `'OptFloat'`, `'OptEmFloat'`, `'Fixed'`, `'Float'`, `'Cap'`, `'Floor'`, `'RangeFloat'`, `'Swap'`. See `instadd` for information on instrument types.

`[Delta,Gamma,Vega,Price] = hwsens( ___ ,Options)` adds an optional input argument for `Options`.

Examples

Compute Instrument Sensitivities Using an HW Interest-Rate Tree

Load the tree and instruments from the `deriv.mat` 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 Basis EndMonthRule IssueDate FirstCou
1     Bond 0.04       01-Jan-2004    01-Jan-2007    1      0     1            NaN       NaN
2     Bond 0.04       01-Jan-2004    01-Jan-2008    1      0     1            NaN       NaN

Index Type Strike Settle         Maturity       CapReset Basis Principal Name   Quantity
3     Cap  0.06   01-Jan-2004    01-Jan-2008    1        0     100       6% Cap 10
```

Compute the `Delta` and `Gamma` for the cap and bond instruments.

```
[Delta, Gamma] = hwsens(HWTree, HWSubSet)
```

```
Delta = 3×1

 -291.2580
 -374.6368
```

```
   60.9580


Gamma = 3×1
10^3 ×

    0.8584
    1.4609
    5.5994
```

Input Arguments

`HWTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hwtree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Delta` — Rate of change of instruments prices with respect to changes in interest rate

vector

Rate of change of instruments prices with respect to changes in the interest rate, returned as a `NINST`-by-1 vector of deltas. `Delta` is computed by finite differences in calls to `hwtree`.

Note `Delta` is calculated based on yield shifts of 100 basis points.

`Gamma` — Rate of change of instruments deltas with respect to changes in interest rate

vector

Rate of change of instruments deltas with respect to changes in the interest rate, returned as a `NINST`-by-1 vector of gammas. `Gamma` is computed by finite differences in calls to `hwtree`.

Note `Gamma` is calculated based on yield shifts of 100 basis points.

`Vega` — Rate of change of instruments prices with respect to changes in volatility

vector

Rate of change of instruments prices with respect to changes in the volatility, returned as a `NINST`-by-`1` vector of vegas. Volatility is $\sigma(t, T)$ of the interest rate. `Vega` is computed by finite differences in calls to `hwtree`. For information on the volatility process, see `hwvolspec`.

Note `Vega` is calculated based on 1% shift in the volatility process.

`Price` — Price of each instrument

vector

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

Version History

Introduced before R2006a

See Also

`hwprice` | `hwtree` | `hwvolspec` | `instadd`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

hwtimespec

Specify time structure for Hull-White interest-rate tree

Syntax

```
TimeSpec = hwtimespec(ValuationDate,Maturity)
TimeSpec = hwtimespec( ___ ,Compounding)
```

Description

`TimeSpec = hwtimespec(ValuationDate,Maturity)` sets the number of levels and node times for a `hwtree` and determines the mapping between dates and time for rate quoting.

`TimeSpec = hwtimespec( ___ ,Compounding)` adds the optional argument `Compounding`.

Examples

Set the Number of Levels and Node Times for a Hull-White Tree

This example shows how to specify a four-period tree with annual nodes and use annual compounding to report rates.

```
ValuationDate = datetime(2004,1,1);
Maturity = [datetime(2004,12,31) ; datetime(2005,12,31) ; datetime(2006,12,31) ; datetime(2007,12
Compounding = 1;
TimeSpec = hwtimespec(ValuationDate, Maturity, Compounding)
```

```
TimeSpec = struct with fields:
           FinObj: 'HWTimeSpec'
    ValuationDate: 731947
         Maturity: [4x1 double]
      Compounding: 1
            Basis: 0
     EndMonthRule: 1
```

Input Arguments

`ValuationDate` — Pricing date and first observation in the tree

datetime scalar | string scalar | date character vector

Pricing date and first observation in the tree, specified as a scalar datetime, string, or data character vector.

To support existing code, `hwtimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Dates marking the cash flow dates of the tree

datetime array | string array | date character vector

Dates marking the cash flow dates of the tree, specified as `NLEVELS`-by-`1` vector using a datetime array, string array, or date character vectors. Cash flows with these maturities fall on tree nodes. `Maturity` should be in increasing order.

To support existing code, `hwtimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Compounding` — Rate at which the input zero rates were compounded when annualized
1 (default) | integer with value of 1, 2, 3, 4, 6, 12, 365, or -1

(Optional) Rate at which the input zero rates were compounded when annualized, specified as a scalar integer value.

- If `Compounding` = `1`, `2`, `3`, `4`, `6`, `12`:

  `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units; for example, `T = F` is one year.
- If `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.
- If `Compounding` = `−1`:

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

Data Types: `double`

Output Arguments

`TimeSpec` — Specification for the time layout for `hwtree`
structure

Specification for the time layout for `hwtree`, returned as a structure. 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(end)`.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `hwtimespec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hwtree` | `hwvolspec`

Topics

“Specifying the Time Structure (TimeSpec)” on page 2-70
“Creating Trees” on page 2-72
“Examining Trees” on page 2-72
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

hwtree

Build Hull-White interest-rate tree

Syntax

```
HWTree = hwtree(VolSpec,RateSpec,TimeSpec)
HWTree = hwtree( ___ ,Name,Value)
```

Description

`HWTree = hwtree(VolSpec,RateSpec,TimeSpec)` builds a Hull-White interest-rate tree.

`HWTree = hwtree( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Create an HWTree

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 = struct with fields:
      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 3 4 5 6]}
FwdTree: {1x4 cell}
```

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

Input Arguments

`VolSpec` — Volatility process specification

structure

Volatility process specification, specified using the `VolSpec` obtained from `hwvolspec`. See `hwvolspec` for information on the volatility process.

Data Types: `struct`

`RateSpec` — Interest-rate specification for initial rate curve

structure

Interest-rate specification for initial rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`TimeSpec` — Time layout specification

structure

Time layout specification, specified using the `TimeSpec` obtained from `hwtimespec`. The `TimeSpec` 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.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `HWTree = hwtree(VolSpec,RateSpec,TimeSpec,'Method','HW1996')`

`Method` — Hull-White method upon which tree-node connectivity algorithm is based

`HW2000` (default) | character vector with values of `HW1996` or `HW2000`

Hull-White method upon which the tree-node connectivity algorithm is based, specified a character vector with a value of `HW1996` or `HW2000`.

Note `hwtree` supports two tree-node connectivity algorithms. `HW1996` is based on the original paper published in the *Journal of Derivatives*, and `HW2000` is the general version of the algorithm, as specified in the paper published in August 2000.

Data Types: `char`

Output Arguments

`HWTree` — Hull-White interest-rate tree

structure

Hull-White interest-rate tree, returned as a structure containing time and interest rate information of a trinomial recombining tree.

The `HWTree` structure returned contains all the information necessary to propagate back any cash flows occurring during the time span of the tree. The main fields of `HWTree` are:

- `HWTree.tObs` contains the time factor of each level of the tree.
- `HWTree.dObs` contains the date of each level of the tree.
- `HWTree.Probs` contains a cell array of `3`-by-`N` numeric arrays with the up/mid/down probabilities of each node of the tree except for the last level. The cells in the cell array are ordered from root node. The arrays are `3`-by-`N` with the first row corresponding to an up-move, the mid row to a mid-move and so on. Each column of the array represents a node starting from the top node of a given level.
- `HWTree.Connect` contains a cell array with connectivity information for each node of the tree. The arrangement is similar to `HWTree.Probs`, with the exception that it has only one row in each cell. The number represents the node in the next level to which the middle branch connects to. The top branch connects to the value above (minus one) and the lower branch connects to the value below (plus one).
- `HWTree.FwdTree` contains the forward spot rate from one node to the next. The forward spot rate is defined as the inverse of the discount factor.

Version History

Introduced before R2006a

References

[1] Hull, J., and A. White. "Using Hull-White Interest Rate Trees." *Journal of Derivatives.* 1996.

[2] Hull, J., and A. White. "*The General Hull-White Model and Super Calibration.*" August 2000.

See Also

`hwcalbycap` | `hwcalbyfloor` | `hwprice` | `hwtimespec` | `hwvolspec` | `intenvset`

Topics

“Creating Trees” on page 2-72
“Examining Trees” on page 2-72
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2

“Supported Interest-Rate Instrument Functions” on page 2-3

hwvolspec

Specify Hull-White interest-rate volatility process

Syntax

```
VolSpec = hwvolspec(ValuationDate,VolDates,VolCurve,AlphaDates,AlphaCurve)
VolSpec = hwvolspec( ___ ,InterpMethod)
```

Description

`VolSpec = hwvolspec(ValuationDate,VolDates,VolCurve,AlphaDates,AlphaCurve)` 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-2.

`VolSpec = hwvolspec( ___ ,InterpMethod)` adds the optional argument `InterpMethod`.

Examples

Create a Structure Specifying the Volatility for `hwtree`

This example shows how to create a Hull-White volatility specification (`VolSpec`) using the following data.

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

Input Arguments

`ValuationDate` — Observation date of the investment horizon
datetime scalar | string scalar | date character vector

Observation date of the investment horizon, specified as a scalar datetime, string, or data character vector.

To support existing code, `hwvolspec` also accepts serial date numbers as inputs, but they are not recommended.

`VolDates` — Number of points of yield volatility end dates
datetime array | string array | date character vector

Number of points of yield volatility end dates, specified as a `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `hwvolspec` also accepts serial date numbers as inputs, but they are not recommended.

`VolCurve` — Yield volatility values
decimal

Yield volatility values, specified as a `NPOINTS`-by-`1` vector of decimal values. The term structure of `VolCurve` is the yield volatility represented by the value of the volatility of the yield from time `t` = 0 to time `t` + *i*, where *i* is any point within the volatility curve.

Note The number of points in `VolCurve` and `AlphaCurve` do not have to be the same.

Data Types: `double`

`AlphaDates` — Mean reversion end dates
datetime array | string array | date character vector

Mean reversion end dates, specified as a `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `hwvolspec` also accepts serial date numbers as inputs, but they are not recommended.

`AlphaCurve` — Positive mean reversion values
positive decimal

Positive mean reversion values, specified as a `NPOINTS`-by-`1` vector of positive decimal values.

Note The number of points in `VolCurve` and `AlphaCurve` do not have to be the same.

Data Types: `double`

`InterpMethod` — Interpolation method
`'linear'` (default) | character vector with values supported by `interp1`

(Optional) Interpolation method, specified as a character vector with values supported by `interp1`.

Data Types: `char`

Output Arguments

`VolSpec` — Specification for the volatility model for `hwtree`

structure

Structure specifying the volatility model for `hwtree`.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `hwvolspec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hwcalbycap` | `hwcalbyfloor` | `interp1`

Topics

“Specifying the Volatility Model (VolSpec)” on page 2-68
“Creating Trees” on page 2-72
“Examining Trees” on page 2-72
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

impvbybaw

Calculate implied volatility using Barone-Adesi and Whaley option pricing model

Syntax

```
Volatility = impvbybaw(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,
OptPrice)
Volatility = impvbybaw( ___ ,Name,Value)
```

Description

`Volatility = impvbybaw(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike, OptPrice)` calculates implied volatility using the Barone-Adesi and Whaley option pricing model.

`Volatility = impvbybaw( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Implied Volatility for an American Option Using the Barone-Adesi and Whaley Option Pricing Model

This example shows how to compute implied volatility using the Barone-Adesi and Whaley option pricing model. Consider three American call options with exercise prices of $100 that expire on July 1, 2017. The underlying stock is trading at $100 on January 1, 2017 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 Barone-Adesi and Whaley option pricing model.

```
AssetPrice = 100;
Settle = datetime(2017,1,1);
Maturity = datetime(2017,6,1);
Strike = 100;
DivAmount = 0.1;
Rate = 0.05;
```

Define the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9794
            Rates: 0.0500
        EndTimes: 0.4167
       StartTimes: 0
         EndDates: 736847
       StartDates: 736696
    ValuationDate: 736696
```

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

Define the `StockSpec`.

```
StockSpec = stockspec(NaN, AssetPrice, {'continuous'}, DivAmount)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: NaN
         AssetPrice: 100
       DividendType: {'continuous'}
    DividendAmounts: 0.1000
    ExDividendDates: []
```

Define the American option.

```
OptSpec = {'call'};
OptionPrice = [4.063;6.77;9.46];
```

Compute the implied volatility for the American option.

```
ImpVol =  impvbybaw(RateSpec, StockSpec, Settle, Maturity, OptSpec,...
Strike, OptionPrice)
```

```
ImpVol = 3×1

    0.1916
    0.3010
    0.4093
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date for the American option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `impvbybaw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date for the American option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `impvbybaw` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | string array with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors or string arrays with values `'call'` or `'put'`.

Data Types: `char` | `string`

`Strike` — American option strike price value
nonnegative scalar | nonnegative vector

American option strike price value, specified as a nonnegative scalar or `NINST`-by-`1` matrix of strike price values. Each row is the schedule for one option.

Data Types: `single` | `double`

`OptPrice` — American option price
nonnegative scalar | nonnegative vector

American option prices from which the implied volatility of the underlying asset is derived, specified as a nonnegative scalar or `NINST`-by-`1` matrix of strike price values.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Volatility = impvbybaw(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,OptionPrice)`

`Limit` — Lower and upper bound of implied volatility search interval
`[0.1 10]` (or 10% to 1000% per annum) (default) | positive value

Lower and upper bound of implied volatility search interval, specified as the comma-separated pair consisting of `'Limit'` and a 1-by-2 positive vector.

Data Types: `double`

`Tolerance` — Implied volatility search termination tolerance
`1e-6` (default) | positive scalar

Implied volatility search termination tolerance, specified as the comma-separated pair consisting of `'Tolerance'` and a positive scalar.

Data Types: `double`

Output Arguments

`Volatility` — Expected implied volatility values
matrix

Expected implied volatility values, returned as a `NINST`-by-`1` matrix. If no solution can be found, a `NaN` is returned.

Version History

Introduced in R2017a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `impvbybaw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Barone-Adesi, G. and Robert E. Whaley. “Efficient Analytic Approximation of American Option Values.” *The Journal of Finance.* Volume 42, Issue 2 (June 1987), 301–320.

[2] Haug, E. *The Complete Guide to Option Pricing Formulas. Second Edition.* McGraw-Hill Education, January 2007.

See Also

`optstockbybaw` | `optstocksensbybaw`

Topics

“Supported Equity Derivative Functions” on page 3-19

impvbybjs

Determine implied volatility using Bjerksund-Stensland 2002 option pricing model

Syntax

```
Volatility = impvbybjs(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,
OptPrice)
Volatility = impvbybjs( ___ ,Name,Value)
```

Description

`Volatility = impvbybjs(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike, OptPrice)` computes implied volatility using the Bjerksund-Stensland 2002 pricing model.

Note `impvbybjs` computes implied volatility of American options with continuous dividend yield using the Bjerksund-Stensland option pricing model.

`Volatility = impvbybjs( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Implied Volatility Using the Bjerksund-Stensland 2002 Option Pricing Model

This example shows how to compute implied volatility using the Bjerksund-Stensland 2002 option pricing model. 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 = datetime(2008,1,1);
Maturity = datetime(2008,6,1);
Strike = 100;
DivAmount = 0.1;
Rate = 0.1;

% define the RateSpec and StockSpec
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rate, 'Compounding', -1, 'Basis', 1);

StockSpec = stockspec(NaN, AssetPrice, {'continuous'}, DivAmount);

OptSpec = {'call'};
OptionPrice = [4.063;6.77;9.46];

ImpVol =  impvbybjs(RateSpec, StockSpec, Settle, Maturity, OptSpec,...
Strike, OptionPrice)
```

```
ImpVol = 3×1

    0.1633
    0.2723
    0.3810
```

The implied volatility is 15% for the first call, and 25% and 35% for the second and third call options.

Input Arguments

`RateSpec` — Interest-rate term structure
structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset
structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `impvbybjs` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date for the American option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `impvbybjs` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'`

Definition of the option from which the implied volatility is derived, specified as a `NINST`-by-`1` cell array of character vectors with a value of `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price value
nonnegative scalar | nonnegative vector

Option strike price value, specified as a nonnegative scalar or `NINST`-by-`1` vector of strike price values. Each row is the schedule for one option.

Data Types: `double`

`OptPrice` — American option price
nonnegative scalar | nonnegative vector

American option prices from which the implied volatility of the underlying asset is derived, specified as a nonnegative scalar or `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Volatility = impvbybjs(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,OptPrice,'Limit',[0.2 20],'Tolerance',1e-5)`

`Limit` — Lower and upper bound of implied volatility search interval
`[0.1 10]` (10% to 1000% per annum) (default) | positive vector

Lower and upper bound of implied volatility search interval, specified as the comma-separated pair consisting of `'Limit'` and a `1`-by-`2` positive vector.

Data Types: `double`

`Tolerance` — Implied volatility search termination tolerance
`1e-6` (default) | positive scalar

Implied volatility search termination tolerance, specified as the comma-separated pair consisting of `'Tolerance'` and a positive scalar.

Data Types: `double`

Output Arguments

`Volatility` — Expected implied volatility values
vector

Expected implied volatility values, returned as a `NINST`-by-`1` vector. If no solution can be found, a `NaN` is returned.

Version History

Introduced in R2008b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `impvbybjs` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bjerksund, P. and G. Stensland. “Closed-Form Approximation of American Options.” *Scandinavian Journal of Management.* Vol. 9, 1993, Suppl., pp. S88–S99.

[2] Bjerksund, P. and G. Stensland. *“Closed Form Valuation of American Options.”* Discussion paper, 2002.

See Also

`optstockbybjs` | `optstocksensbybjs`

Topics
“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing Using the Bjerksund-Stensland Model” on page 3-84
“Supported Equity Derivative Functions” on page 3-19

impvbyblk

Determine implied volatility using Black option pricing model

Syntax

```
Volatility = impvbyblk(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,
OptPrice)
Volatility = impvbyblk( ___ ,Name,Value)
```

Description

`Volatility = impvbyblk(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike, OptPrice)` computes implied volatility using the Black option pricing model.

`Volatility = impvbyblk( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Implied Volatility Using the Black Option Pricing Model

This example shows how to compute the implied volatility using the Black option pricing model. 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. Find 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 = datetime(2008,5,1);
Maturity = datetime(2008,9,1);

% define the 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'};

Price = [1.14;0.82];
Volatility = impvbyblk(RateSpec, StockSpec, Settle, Maturity, OptSpec,...
Strike, Price,'Method','jackel2016')
```

```
Volatility = 2×1

    0.4052
    0.3021
```

The implied volatility is 41% and 30%.

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `impvbyblk` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date for the American option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `impvbyblk` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'`

Definition of the option from which the implied volatility is derived, specified as a `NINST`-by-`1` cell array of character vectors with a value of `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price

nonnegative scalar | nonnegative vector

Option strike price value, specified as a nonnegative scalar or `NINST`-by-`1` vector of strike prices. Each row is the schedule for one option.

Data Types: double

OptPrice — European option price
nonnegative scalar | nonnegative vector

European option prices from which the implied volatility of the underlying asset is derived, specified as a nonnegative scalar or NINST-by-1 vector.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: Volatility = impvbyblk(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,OptPrice,'Limit',5,'Tolerance',1e-5)

Limit — Upper bound of implied volatility search interval
10 (1000% per annum) (default) | positive value

Upper bound of implied volatility search interval, specified as the comma-separated pair consisting of 'Limit' and a positive scalar.

Note If you are using Method with a value of 'jackel2016', the Limit argument is ignored.

Data Types: double

Tolerance — Implied volatility search termination tolerance
1e-6 (default) | positive scalar

Implied volatility search termination tolerance, specified as the comma-separated pair consisting of 'Tolerance' and a positive scalar.

Note If you are using Method with a value of 'jackel2016', the Tolerance argument is ignored.

Data Types: double

Method — Method for computing implied volatility
'jackel2016' (default) | character vector with values 'search' or 'jackel2016' | string with values "search" or "jackel2016"

Method for computing implied volatility, specified as the comma-separated pair consisting of 'Method' and a character vector with a value of 'search' or 'jackel2016' or a string with a value of "search" or "jackel2016".

Data Types: char | string

Output Arguments

`Volatility` — Expected implied volatility values
vector

Expected implied volatility values, returned as a `NINST`-by-`1` vector. If no solution can be found, a `NaN` is returned.

Version History

Introduced in R2008b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `impvbyblk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Jäckel, Peter. "Let's Be Rational." *Wilmott Magazine.*, January, 2015 (https://onlinelibrary.wiley.com/doi/abs/10.1002/wilm.10395).

See Also

`optstockbyblk` | `optstocksensbyblk`

Topics
“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing Using the Black Model” on page 3-83
“Black Model” on page 3-80
“Supported Equity Derivative Functions” on page 3-19

impvbybls

Determine implied volatility using Black-Scholes option pricing model

Syntax

```
Volatility = impvbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,
OptPrice)
Volatility = impvbybls( ___ ,Name,Value)
```

Description

`Volatility = impvbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike, OptPrice)` computes implied volatility using the Black-Scholes option pricing model.

`Volatility = impvbybls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Implied Volatility Using the Black-Scholes Option Pricing Model

This example shows how to compute the implied volatility using the Black-Scholes option pricing model. 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 the RateSpec and StockSpec
RateSpec = intenvset('ValuationDate', Settlement, 'StartDates', Settlement,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', -1, 'Basis', 1);

StockSpec = stockspec(NaN, AssetPrice);

ImpvVol =  impvbybls(RateSpec, StockSpec, Settlement, Maturity, OptSpec,...
Strike, OptionPrice,'Method','jackel2016')
```

```
ImpvVol = 2×1

    0.3175
    0.4878
```

The implied volatility is 31.75% for the call and 48.78% for the put.

Input Arguments

RateSpec — Interest-rate term structure
structure

Interest-rate term structure (annualized and continuously compounded), specified by the RateSpec obtained from intenvset. For information on the interest-rate specification, see intenvset.

Data Types: struct

StockSpec — Stock specification for underlying asset
structure

Stock specification for the underlying asset. For information on the stock specification, see stockspec.

stockspec handles several types of underlying assets. For example, for physical commodities the price is StockSpec.Asset, the volatility is StockSpec.Sigma, and the convenience yield is StockSpec.DividendAmounts.

Data Types: struct

Settle — Settlement date
datetime array | string array | date character vector

Settlement date, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, impvbybls also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date for the American option, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, impvbybls also accepts serial date numbers as inputs, but they are not recommended.

OptSpec — Definition of option
character vector with values 'call' or 'put'

Definition of the option from which the implied volatility is derived, specified as a NINST-by-1 cell array of character vectors with a value of 'call' or 'put'.

Data Types: char | cell

Strike — Option strike price
nonnegative scalar | nonnegative vector

Option strike price value, specified as a nonnegative scalar or NINST-by-1 vector of strike prices. Each row is the schedule for one option.

Data Types: double

`OptPrice` — European option price

nonnegative scalar | nonnegative vector

European option prices from which the implied volatility of the underlying asset is derived, specified as a nonnegative scalar or `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Volatility = impvbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,OptPrice,'Limit',5,'Tolerance',1e-5)`

`Limit` — Upper bound of implied volatility search interval

`10` (1000% per annum) (default) | positive value

Upper bound of implied volatility search interval, specified as the comma-separated pair consisting of `'Limit'` and a positive scalar.

Note If you are using `Method` with a value of `'jackel2016'`, the `Limit` argument is ignored.

Data Types: `double`

`Tolerance` — Implied volatility search termination tolerance

`1e-6` (default) | positive scalar

Implied volatility search termination tolerance, specified as the comma-separated pair consisting of `'Tolerance'` and a positive scalar.

Note If you are using `Method` with a value of `'jackel2016'`, the `Tolerance` argument is ignored.

Data Types: `double`

`Method` — Method for computing implied volatility

`'jackel2016'` (default) | character vector with values `'search'` or `'jackel2016'` | string with values `"search"` or `"jackel2016"`

Method for computing implied volatility, specified as the comma-separated pair consisting of `'Method'` and a character vector with a value of `'search'` or `'jackel2016'` or a string with a value of `"search"` or `"jackel2016"`.

Data Types: `char` | `string`

Output Arguments

`Volatility` — Expected implied volatility values

vector

Expected implied volatility values, returned as a `NINST`-by-`1` vector. If no solution can be found, a `NaN` is returned.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `impvbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Jäckel, Peter. "Let's Be Rational." *Wilmott Magazine.*, January, 2015 (https://onlinelibrary.wiley.com/doi/abs/10.1002/wilm.10395).

See Also

`optstockbybls` | `optstocksensbybls`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing Using the Black-Scholes Model” on page 3-82
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Supported Equity Derivative Functions” on page 3-19

impvbyrgw

Determine implied volatility using Roll-Geske-Whaley option pricing model for American call option

Syntax

```
Volatility = impvbyrgw(RateSpec,StockSpec,Settle,Maturity,Strike,OptPrice)
Volatility = impvbyrgw( ___ ,Name,Value)
```

Description

`Volatility = impvbyrgw(RateSpec,StockSpec,Settle,Maturity,Strike,OptPrice)` computes implied volatility using Roll-Geske-Whaley option pricing model for American call option.

Note `impvbyrgw` computes implied volatility of American calls with a single cash dividend using the Roll-Geske-Whaley option pricing model.

`Volatility = impvbyrgw( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Implied Volatility Using the Roll-Geske-Whaley Option Pricing Model

This example shows how to compute the implied volatility using the Roll-Geske-Whaley option pricing model. 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 = datetime(2008,7,1);
Maturity = datetime(2009,7,1);

% define the RateSpec and StockSpec
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
 'EndDates', Maturity, 'Rates', Rates, 'Compounding', -1);

StockSpec = stockspec(NaN, AssetPrice, {'cash'}, 0.25, {'Nov 1,2008'});

Price = [1.346];
Volatility = impvbyrgw(RateSpec, StockSpec, Settle, Maturity, Strike, Price)

Volatility = 0.3539
```

Input Arguments

`RateSpec` — Interest-rate term structure
structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset
structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `impvbyrgw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date for the American option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `impvbyrgw` also accepts serial date numbers as inputs, but they are not recommended.

`Strike` — Option strike price value
nonnegative scalar | nonnegative vector

Option strike price value, specified as a nonnegative scalar or `NINST`-by-`1` vector of strike price values. Each row is the schedule for one option.

Data Types: `double`

`OptPrice` — American option price
nonnegative scalar | nonnegative vector

American option prices from which the implied volatility of the underlying asset is derived, specified as a nonnegative scalar or `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Volatility = impvbyrgw(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,OptPrice,'Limit',5,'Tolerance',1e-5)`

`Limit` — Upper bound of implied volatility search interval

`10` (1000% per annum) (default) | positive value

Upper bound of implied volatility search interval, specified as the comma-separated pair consisting of `'Limit'` and a positive scalar.

Data Types: `double`

`Tolerance` — Implied volatility search termination tolerance

`1e-6` (default) | positive scalar

Implied volatility search termination tolerance, specified as the comma-separated pair consisting of `'Tolerance'` and a positive scalar.

Data Types: `double`

Output Arguments

`Volatility` — Expected implied volatility values

vector

Expected implied volatility values, returned as a `NINST`-by-`1` vector. If no solution can be found, a `NaN` is returned.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `impvbyrgw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`optstockbyrgw` | `optstocksensbyrgw`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing Using the Roll-Geske-Whaley Model” on page 3-84
“Roll-Geske-Whaley Model” on page 3-80
“Supported Equity Derivative Functions” on page 3-19

instadd

Add types to instrument collection

Syntax

```
InstSetNew = instadd(InstSetOld,TypeString,Data)
```

Description

`InstSetNew = instadd(InstSetOld,TypeString,Data)` adds instruments to an existing instrument set `InstSet`. The output `InstSetNew` is a new instrument set containing the input data.

`InstSetNew` and `InstSetOld` store instruments of types `'Asian'`, `'Barrier'`, `'Bond'`, `'CBond'`, `'Cap'`, `'CashFlow'`, `'Compound'`, `'Fixed'`, `'Float'`, `'Floor'`, `'Lookback'`, `'OptBond'`, `'OptEmBond'`, `'OptEmFloat'`, `'RangeFloat'`,`'OptStock'`, `'Swap'`, or `'Swaption'`. Financial Instruments Toolbox provides pricing and sensitivity routines for these instruments.

Examples

Create a Portfolio with Two Cap Instruments and a 4% Bond

Define the bond:

```
Strike = [0.06; 0.07];
CouponRate = 0.04;
Settle = '06-Feb-2000';
Maturity = '15-Jan-2003';
```

Create a portfolio with two cap instruments and a 4% bond and then display the portfolio:

```
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    1        0     100
2     Cap  0.07   06-Feb-2000    15-Jan-2003    1        0     100

Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
3     Bond 0.04       06-Feb-2000    15-Jan-2003    2      0     1            NaN       NaN
```

Input Arguments

`InstSetOld` — Existing instrument variable

structure

Existing instrument variable, specified as an `InstSet` structure.

`InstSetOld` can contain instruments of types `'Asian'`, `'Barrier'`, `'Bond'`, `'CBond'`, `'Cap'`, `'CashFlow'`, `'Compound'`, `'Fixed'`, `'Float'`, `'Floor'`, `'Lookback'`, `'OptBond'`, `'OptEmBond'`, `'OptEmFloat'`, `'RangeFloat'`, `'OptStock'`, `'Swap'`, or `'Swaption'`.

Data Types: `struct`

`TypeString` — Instrument type
character vector

Instrument type, specified as a character vector. The supported instrument types are:

- `'Asian'`
- `'Barrier'`
- `'Bond'`
- `'CBond'`
- `'Cap'`
- `'CashFlow'`
- `'Compound'`
- `'Fixed'`
- `'Float'`
- `'Floor'`
- `'Lookback'`
- `'OptBond'`
- `'OptEmBond'`
- `'OptEmFloat'`
- `'RangeFloat'`
- `'OptStock'`
- `'Swap'`
- `'Swaption'`

Note Each instrument type can have different `Data` fields.

Data Types: `char`

`Data` — Instrument data fields
vector

Instrument data fields, specified as a row vector or character vector for each instrument.

- `'Asian'` — For example:

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

  For more information on Asian instrument arguments, see `instasian`.
- `'Barrier'` — For example:

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

For more information on the Barrier instrument arguments, see `instbarrier`.

- `'Bond'` — For example:

  ```
  InstSet = instadd('Bond',CouponRate,Settle,Maturity,Period,Basis,EndMonthRule,IssueDate,FirstC
  ```

  For more information on the Bond instrument arguments , see `instbond`.

- `'CBond'` — For example:

  ```
  InstSet = instadd('CBond',CouponRate,Settle,Maturity,ConvRatio'CallStrike',CallStrike,'CallExD
  'PutExDates',PutExDates,'AmericanPut',AmericanPut,'Period',Period,'Face',Face,'Spread',Spread,
  LastCouponDate,'StartDate',StartDate)
  ```

  For more information on the CBond instrument arguments , see `instcbond`.

- `'Cap'` — For example:

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

  For more information on the Cap instrument arguments , see `instcap`.

- `'CashFlow'` — For example:

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

  For more information on the Cash Flow instrument arguments , see `instcf`.

- `'Compound'` — For example:

  ```
  InstSet = instadd('Compound',UOptSpec,UStrike,USettle,UExerciseDates,UAmericanOpt,COptSpec,CSt
  ```

  For more information on the Compound instrument arguments , see `instcompound`.

- `'Fixed'` — For example:

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

  For more information on the Fixed instrument arguments , see `instfixed`.

- `'Float'` — For example:

  ```
  InstSet = instadd('Float',Spread,Settle,Maturity,Reset,Basis,Principal,EndMonthRule,CapRate,Fl
  ```

  For more information on the Fixed instrument arguments , see `instfloat`.

- `'Floor'` — For example:

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

  For more information on the Floor instrument arguments , see `instfloor`.

- `'Lookback'` — For example:

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

  For more information on the Lookback instrument arguments , see `instlookback`.

- `'OptBond'` — For example:

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

  For more information on the Bond Option instrument arguments , see `instoptbnd`.

- `'OptStock'` — For example:

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

  For more information on the Stock Option instrument arguments , see `instoptstock`.
- `'OptEmBond'` — For example:

  ```
  InstSet = instoptembnd(CouponRate, Settle, Maturity, OptSpec, Strike, ...
  ExerciseDates,'AmericanOpt', AmericanOpt, 'Period', Period)
  ```

  For more information on the Bond with embedded option instrument arguments , see `instoptembnd`.
- `'OptemFloat'` — For example:

  ```
  InstSet = instoptemfloat(Spread, Settle, Maturity, OptSpec, Strike, ...
  ExerciseDates,'FloatReset', Reset)
  ```

  For more information on the Floating Rate Note with an embedded option instrument arguments , see `instoptemfloat`.
- `'RangeFloat'` — For example:

  ```
  InstSet = instrangefloat(Spread, Settle, Maturity, RateSched)
  ```

  For more information on the Range Note instrument arguments , see `instrangefloat`.
- `'Swap'` — For example:

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

  For more information on the Swap instrument arguments , see `instswap`.
- `'Swaption'` — For example:

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

  For more information on the Swaption instrument arguments , see `instswaption`.

Data Types: `char`

Output Arguments

`InstSetNew` — Instrument set containing the new input data

structure

`InstSetNew` is an instrument set containing the new input data, returned as a structure.

Version History

Introduced before R2006a

See Also

`instasian` | `instbarrier` | `instbond` | `instcap` | `instcf` | `instcompound` | `instfixed` | `instfloat` | `instfloor` | `instlookback` | `instoptbnd` | `instoptembnd` | `instoptstock` | `instswap` | `instswaption` | `instaddfield` | `instdisp` | `instcbond`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Creating Instruments or Properties” on page 1-16
“Instrument Constructors” on page 1-15

instaddfield

Add new instruments to instrument collection

Syntax

```
InstSetNew = instaddfield(InstSet,Name,Value)
InstSet = instaddfield(Name,Value)
```

Description

`InstSetNew = instaddfield(InstSet,Name,Value)` adds instruments to an existing instrument set `InstSet`. The output `InstSetNew` is a new instrument set containing the input data.

`InstSet = instaddfield(Name,Value)` creates an instrument variable `InstSet`.

Examples

Add New Instruments to Instrument Collection

Build a portfolio around the following July options.

```
Strike = (95:5:105)'
```

```
Strike = 3×1

    95
   100
   105
```

```
CallP = [12.2; 9.2; 6.8]
```

```
CallP = 3×1

   12.2000
    9.2000
    6.8000
```

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')

FN = 3x1 cell
    {'Strike'}
    {'Price' }
    {'Opt'   }


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')

InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {2x1 cell}
     FieldName: {2x1 cell}
    FieldClass: {2x1 cell}
     FieldData: {2x1 cell}


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)

InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {3x1 cell}
     FieldName: {3x1 cell}
    FieldClass: {3x1 cell}
     FieldData: {3x1 cell}


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
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of instruments, specified as `InstSet` structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `InstSet = instaddfield('Type','Option','FieldName', {'Strike','Price','Opt'},'Data',{Strike,CallP,'Call'})`

`FieldName` — Name of each data field for instrument
cell array of character vectors

Name of each data field for an instrument, specified as the comma-separated pair consisting of `'FieldName'` and an `NFIELDS`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Data` — Data contents for each field
array | cell array

Data contents for each field, specified as the comma-separated pair consisting of `'Data'` and an `NINST`-by-`M` array or `NFIELDS`-by-`1` cell array.

Data Types: `double` | `cell`

`FieldClass` — Data class of each field
vector

Data class of each field, specified as the comma-separated pair consisting of `'FieldClass'` and an `NFIELDS`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Type` — Type of instrument added
character vector

Type of instrument added, specified as the comma-separated pair consisting of `'Type'` and a character vector. Instruments of different types can have different `FieldName` collections.

Data Types: `char`

Output Arguments

`InstSetNew` — Instrument set variable containing new input data added to existing `InstSet`
structure

Instrument set variable containing the new input data added to an existing `InstSet` , returned as a structure.

`InstSet` — New Instrument set variable containing input data
structure

New Instrument set variable containing input data, returned as a structure.

Version History

Introduced before R2006a

See Also

`instdisp` | `instget` | `instgetcell` | `instsetfield` | `instadd`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Creating Instruments or Properties” on page 1-16
“Instrument Constructors” on page 1-15

instasian

Construct Asian option

Syntax

```
InstSet = instasian(OptSpec,Strike,Settle,ExerciseDates)
InstSet = instasian(InstSet,OptSpec,Strike,Settle,ExerciseDates)
InstSet = instasian( ___ ,AmericanOpt,AvgType,AvgPrice,AvgDate)
[FieldList,ClassList,TypeString = instasian
```

Description

`InstSet = instasian(OptSpec,Strike,Settle,ExerciseDates)` creates a new instrument set containing Asian instruments.

`InstSet = instasian(InstSet,OptSpec,Strike,Settle,ExerciseDates)` adds Asian instruments to an existing instrument set.

`InstSet = instasian( ___ ,AmericanOpt,AvgType,AvgPrice,AvgDate)` adds optional arguments.

`[FieldList,ClassList,TypeString = instasian` lists field meta-data for the Asian instrument.

Examples

Create an Asian Option Instrument

Load the example instrument set, `deriv.mat`, and set the required values for an asian option instrument.

```
load deriv.mat
```

Create a subportfolio with barrier and lookback options.

```
CRRSubSet = instselect(CRRInstSet,'Type',{'Barrier', 'Lookback'});
```

Define the asian instrument.

```
OptSpec = 'put';
Strike = NaN;
Settle = datetime(2003,1,1);
ExerciseDates = datetime(2004,1,1);
```

Add a floating strike asian option to the instrument set.

```
InstSet = instasian(CRRSubSet, OptSpec, Strike, Settle, ExerciseDates);
instdisp(InstSet)
```

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

```
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

Index Type  OptSpec Strike Settle         ExerciseDates  AmericanOpt AvgType    AvgPrice AvgDate
4     Asian put     NaN    01-Jan-2003    01-Jan-2004    0           arithmetic NaN      NaN
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding asian instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a scalar character vector or an `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price value

nonnegative integer | vector of nonnegative integer

Option strike price value, specified with a scalar nonnegative integer or an `NINST`-by-`1` vector of strike price values.

Data Types: `double`

`Settle` — Settlement dates or trade dates

datetime array | string array | date character vector

Settlement date or trade date for the Asian option, specified as scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instasian` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instasian` also accepts serial date numbers as inputs, but they are not recommended.

For a European option (when `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 (when 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 an NINST-by-1, the option can be exercised between the valuation date of the stock tree and the single listed exercise date.

AmericanOpt — Indicator for American option
0 (default) | integer value 0 or 1

(Optional) Indicator for American option, specified as a scalar or an NINST-by-1 vector.

If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.

Data Types: double

AvgType — Averaging type
'arithmetic' (default) | character vector with value 'arithmetic' or 'geometric'

(Optional) Averaging type, specified as a character vector with a value of 'arithmetic' for arithmetic average or 'geometric' for geometric average.

Data Types: char

AvgPrice — Average price of underlying asset at the Settle date
current stock price of the underlying asset (default) | numeric

(Optional) Average price of underlying asset at the Settle date, specified as a scalar numeric.

Note Use AvgPrice when AvgDate < Settle.

Data Types: double

AvgDate — Date averaging period begins
Settle date (default) | datetime array | string array | date character vector

(Optional) Date averaging period begins, specified as a scalar datetime, string, or date character vector.

To support existing code, instasian also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

InstSet — Variable containing a collection of instruments
structure

Variable containing a collection of instruments, returned as a structure. 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 InstSet variable, see instget.

`FieldList` — Name of each data field for Asian instrument
cell array of character vectors

Name of each data field for an Asian instrument, returned as an `NFIELDS`-by-1 cell array of character vectors.

`ClassList` — Data class for each field
cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-1 cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument
character vector

Type of instrument, returned as a character vector. For an Asian option instrument, `TypeString = 'Asian'`.

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instasian` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`instdisp` | `instget` | `instgetcell` | `instsetfield` | `instadd`

Topics

“Pricing Equity Derivatives Using Trees” on page 3-64
“Creating Instruments or Properties” on page 1-16
“Asian Option” on page 3-34
“Instrument Constructors” on page 1-15
“Supported Equity Derivative Functions” on page 3-19
“Choose Instruments, Models, and Pricers” on page 1-53

instbarrier

Construct barrier option

Syntax

```
InstSet = instbarrier(OptSpec,Strike,Settle,ExerciseDates,AmericanOpt,
BarrierSpec,Barrier,Rebate)
InstSet = instbarrier(InstSetOld,OptSpec,Strike,Settle,ExerciseDates,
AmericanOpt,BarrierSpec,Barrier,Rebate)
[FieldList,ClassList,TypeString] = instbarrier
```

Description

`InstSet = instbarrier(OptSpec,Strike,Settle,ExerciseDates,AmericanOpt, BarrierSpec,Barrier,Rebate)` constructs a barrier instrument.

Data arguments are `NINST`-by-`1` vectors, scalar, or empty. Fill in unspecified entries vectors with `NaN`. Only one data argument is required to create the instrument. The others can be omitted or passed as empty matrices `[]`.

`InstSet = instbarrier(InstSetOld,OptSpec,Strike,Settle,ExerciseDates, AmericanOpt,BarrierSpec,Barrier,Rebate)` adds barrier options to an existing instrument variable `InstSetOld`).

`[FieldList,ClassList,TypeString] = instbarrier` lists field metadata for the barrier instrument.

Examples

Create Two Barrier Option Instruments

Create an instrument set of two barrier options with the following data:

```
OptSpec = {'put';'call'};
Strike = 112;
Settle = datetime(2012,1,1);
ExerciseDates = datetime(2015,1,1);
BarrierSpec = {'do';'ui'};
Barrier = [101;102];
AmericanOpt = 0;
```

Create the instrument set (`InstSet`) for the two barrier options.

```
InstSet = instbarrier(OptSpec, Strike, Settle, ExerciseDates,AmericanOpt, BarrierSpec, Barrier);
```

Display the instrument set.

```
instdisp(InstSet)
```

```
Index Type    OptSpec Strike Settle         ExerciseDates  AmericanOpt BarrierSpec Barrier Rebate
1     Barrier put     112    01-Jan-2012    01-Jan-2015    0           do          101     0
```

```
2     Barrier call    112    01-Jan-2012    01-Jan-2015    0           ui          102     0
```

Input Arguments

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'`

Definition of an option as `'call'` or `'put'`, specified as a `NINST`-by-`1` list of character vector values.

Data Types: `char`

`Strike` — Option strike price value

integer

Option strike price value, specified as an `NINST`-by-`1` vector of strike values. Each row is the schedule for one option.

Data Types: `double`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement date for the barrier option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instbarrier` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option (`AmericanOpt = 0`), specified as a `NINST`-by-`1` 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 (`AmericanOpt = 1`), specified as a `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.

To support existing code, `instbarrier` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Flag for American option

integer with values `0` or `1`

Flag for American option, specified as an integer with values `0` or `1`. If `AmericanOpt = 0`, `NaN`, or is unspecified, the option is a European option. If `AmericanOpt = 1`, the option is an American option.

Data Types: `logical`

`BarrierSpec` — Barrier option type
character vector with values: `'UI'`, `'UO'`, `'DI'`, `'DO'`

Barrier option type, specified as a character vector with the following values:

- `'UI'` — Up Knock In

  This option becomes effective when the price of the underlying asset passes above the barrier level. It gives the option holder the right, but not the obligation, to buy/sell (call/put) the underlying security at the strike price if the underlying asset goes above the barrier level during the life of the option.
- `'UO'` — Up Knock Out

  This option gives the option holder the right, but not the obligation, to buy/sell (call/put) the underlying security at the strike price as long as the underlying asset does not go above the barrier level during the life of the option. This option terminates when the price of the underlying asset passes above the barrier level. Usually, with an up-and-out option, the rebate is paid if the spot price of the underlying reaches or exceeds the barrier level.
- `'DI'` — Down Knock In

  This option becomes effective when the price of the underlying stock passes below the barrier level. It gives the option holder the right, but not the obligation, to buy/sell (call/put) the underlying security at the strike price if the underlying security goes below the barrier level during the life of the option. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option.
- `'DO'` — Down Knock Up

  This option gives the option holder the right, but not the obligation, to buy/sell (call/put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. Usually the option holder receives a rebate amount if the option expires worthless.

Option	Barrier Type	Payoff if Barrier Crossed	Payoff if Barrier not Crossed
Call/Put	Down Knock-out	Worthless	Standard Call/Put
Call/Put	Down Knock-in	Call/Put	Worthless
Call/Put	Up Knock-out	Worthless	Standard Call/Put
Call/Put	Up Knock-in	Standard Call/Put	Worthless

Data Types: `char`

`Barrier` — Barrier value
integer

Barrier value, specified as a vector of values.

Data Types: `double`

`Rebate` — Rebate value
integer

(Optional) Rebate value, specified as a vector of values.

Data Types: `double`

`InstSetOld` — Instrument variable
structure

(Optional) Instrument variable, this argument is specified only when adding barrier instruments to an existing instrument set. See `instget` for more information on the `InstSet` variable.

Data Types: `struct`

Output Arguments

`InstSet` — Instrument variable for barrier option
structure

Instrument variable for barrier option, returned as a structure. See `instget` for more information on the `InstSet` variable.

`FieldList` — Fields in `InstSet` instrument
cell array of character vectors

Fields in `InstSet` instrument are returned as a (`NFIELDS`-by-`1`) cell array of character vectors listing the name of each data field for this instrument type.

`ClassList` — Data class of each field in `InstSet` instrument
cell array of character vectors

Data class of each field in `InstSet` instrument, returned as an `NFIELDS`-by-`1` cell array of character vectors listing the data class of each field. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument added to `InstSet` instrument
character vector

Type of instrument added to `InstSet` instrument, returned as a character vector specifying the type of instrument added. For a barrier option instrument, `TypeString = 'Barrier'`.

More About

Barrier Option

A barrier option has not only a strike price but also a barrier level and sometimes a rebate.

A rebate is a fixed amount that is paid if the option cannot be exercised because the barrier level has been reached or not reached. The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by `Barrier`, during the life of the option.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `instbarrier` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`instadd` | `instdisp` | `instget` | `barrierbyls` | `barrierbyfd` | `barrierbystt` | `barrierbyitt` | `barrierbybls` | `barrierbycrr` | `barrierbyeqp`

Topics

“Pricing Equity Derivatives Using Trees” on page 3-64
“Creating Instruments or Properties” on page 1-16
“Instrument Constructors” on page 1-15
“Supported Equity Derivative Functions” on page 3-19
“Choose Instruments, Models, and Pricers” on page 1-53

instbond

Construct bond instrument

Syntax

```
InstSet = instbond(CouponRate,Settle,Maturity)
InstSet = instbond(InstSet,CouponRate,Settle,Maturity)
InstSet = instbond( ___ ,Period,Basis,EndMonthRule,IssueDate,FirstCouponDate,
LastCouponDate,StartDate,Face)
[FieldList,ClassList,TypeString] = instbond
```

Description

`InstSet = instbond(CouponRate,Settle,Maturity)` creates a new instrument set containing Bond instruments.

`InstSet = instbond(InstSet,CouponRate,Settle,Maturity)` adds Bond instruments to an existing instrument set.

`InstSet = instbond( ___ ,Period,Basis,EndMonthRule,IssueDate,FirstCouponDate, LastCouponDate,StartDate,Face)` adds optional arguments.

`[FieldList,ClassList,TypeString] = instbond` lists field meta-data for the Bond instrument.

Examples

Create a Bond Instrument

Create a new instrument variable with the following information:

```
CouponRate= [0.035;0.04];
Settle= datetime(2013,11,1);
Maturity = datetime(2014,11,1);
Period =1;

InstSet = instbond(CouponRate, Settle, Maturity, ...
Period)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Bond'}
     FieldName: {{11x1 cell}}
    FieldClass: {{11x1 cell}}
     FieldData: {{11x1 cell}}
```

Display the instrument set.

```
instdisp(InstSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.035      01-Nov-2013    01-Nov-2014    1      0     1            NaN       NaN
2     Bond 0.04       01-Nov-2013    01-Nov-2014    1      0     1            NaN       NaN
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding Bond instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`CouponRate` — Coupon rate indicating the annual percentage rate

decimal

Coupon rate indicating the annual percentage rate, specified as an `NINST`-by-`1` vector or an `NINST`-by-`1` cell array of decimal annual rates, or decimal annual rate schedules. For the latter case of a variable coupon schedule, each element of the cell array is a `NumDates`-by-`2` cell array, where the first column is dates and the second column is its associated rate. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement dates

datetime array | string array | date character vector

Settlement dates, specified as scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note `Settle` must be earlier than `Maturity`.

To support existing code, `instbond` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity dates

datetime array | string array | date character vector

Maturity dates, specified as scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instbond` also accepts serial date numbers as inputs, but they are not recommended.

`Period` — Coupons per year

2 per year (default) | vector

(Optional) Coupons per year, specified as a scalar or an `NINST`-by-`1` vector. Values for `Period` are `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis, specified as scalar or an `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

(Optional) End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as a scalar or a nonnegative integer [`0`, `1`] using an `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond's coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

(Optional) Bond issue date, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instbond` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

(Optional) Irregular first coupon date, specified as a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, instbond also accepts serial date numbers as inputs, but they are not recommended.

When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

LastCouponDate — Irregular last coupon date
datetime array | string array | date character vector

(Optional) Irregular last coupon date, specified as a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, instbond also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

StartDate — Forward starting date of payments
Settle date (default) | datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, instbond also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify StartDate, the effective start date is the Settle date.

Face — Face value
100 (default) | nonnegative value | cell array of nonnegative values

(Optional) Face or par value, specified as a scalar or an NINST-by-1 vector of nonnegative face values or an NINST-by-1 cell array of face values or face value schedules. For the latter case, each element of the cell array is a NumDates-by-2 cell array, where the first column is dates and the second column is its associated face value. The date indicates the last day that the face value is valid.

Data Types: cell | double

Output Arguments

InstSet — Variable containing a collection of instruments
structure

Variable containing a collection of instruments, returned as a structure. 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 InstSet variable, see instget.

`FieldList` — Name of each data field for Bond instrument
cell array of character vectors

Name of each data field for a Bond instrument, returned as an `NFIELDS`-by-1 cell array of character vectors.

`ClassList` — Data class for each field
cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-1 cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument
character vector

Type of instrument, returned as a character vector. For a Bond instrument, `TypeString = 'Bond'`.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `instbond` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmprice` | `instaddfield` | `instdisp` | `instget` | `intenvprice`

Topics
“Creating Instruments or Properties” on page 1-16
“Bond” on page 2-3
“Instrument Constructors” on page 1-15
“Supported Interest-Rate Instrument Functions” on page 2-3
“Choose Instruments, Models, and Pricers” on page 1-53

instcap

Construct cap instrument

Syntax

```
InstSet = instcap(Strike,Settle,Maturity)
InstSet = instcap(InstSet,Strike,Settle,Maturity)
InstSet = instcap( ___ ,CapReset,Basis,Principal)
[FieldList,ClassList,TypeString] = instcap
```

Description

`InstSet = instcap(Strike,Settle,Maturity)` creates a new instrument set containing Cap instruments.

`InstSet = instcap(InstSet,Strike,Settle,Maturity)` adds Cap instruments to an existing instrument set.

`InstSet = instcap( ___ ,CapReset,Basis,Principal)` adds optional arguments.

`[FieldList,ClassList,TypeString] = instcap` lists field meta-data for the Cap instrument.

Examples

Create Two Cap Instruments

Create a new instrument variable with the following information:

```
Strike = [0.035; 0.045];
Settle= datetime(2013,1,1);
Maturity = datetime(2043,1,1);
Reset = 1;
Basis = 1;
Principal = 1000;
```

Create the new cap instruments.

```
InstSet = instcap(Strike, Settle, Maturity, Reset, Basis, Principal)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Cap'}
     FieldName: {{6x1 cell}}
    FieldClass: {{6x1 cell}}
     FieldData: {{6x1 cell}}
```

Display the cap instruments.

```
instdisp(InstSet)
```

```
Index Type Strike Settle         Maturity       CapReset Basis Principal
1     Cap  0.035  01-Jan-2013    01-Jan-2043    1        1     1000
2     Cap  0.045  01-Jan-2013    01-Jan-2043    1        1     1000
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding Cap instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`Strike` — Rate at which cap is exercised

decimal

Rate at which the Cap is exercised, specified as a scalar or an `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement dates

datetime array | string array | date character vector

Settlement dates, specified as scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note `Settle` must be earlier than `Maturity`.

To support existing code, `instcap` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity dates

datetime array | string array | date character vector

Maturity dates, specified as scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instcap` also accepts serial date numbers as inputs, but they are not recommended.

`CapReset` — Reset frequency payment per year

`1` (default) | numeric

(Optional) Reset frequency payment per year, specified as a scalar or an `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis, specified as scalar or an `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a scalar or an `NINST`-by-`1` of notional principal amounts, or an `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing Cap.

Data Types: `double` | `cell`

Output Arguments

`InstSet` — Variable containing a collection of instruments
structure

Variable containing a collection of instruments, returned as a structure. 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 `InstSet` variable, see `instget`.

`FieldList` — Name of each data field for Cap instrument
cell array of character vectors

Name of each data field for a Cap instrument, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class for each field
cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-`1` cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument
character vector

Type of instrument, returned as a character vector. For a Cap option instrument, `TypeString = 'Cap'`.

More About

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)

For more information, see “Cap” on page 2-12.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `instcap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`instbond` | `instfloor` | `instdisp` | `instswap` | `intenvprice` | `instaddfield` | `hjmprice`

Topics
“Creating Instruments or Properties” on page 1-16
“Cap” on page 2-12
“Instrument Constructors” on page 1-15
“Supported Interest-Rate Instrument Functions” on page 2-3
“Choose Instruments, Models, and Pricers” on page 1-53

instcbond

Construct CBond instrument for convertible bond

Syntax

```
ISet = instcbond(CouponRate,Settle,Maturity,ConvRatio)
ISet = instcbond( ___ ,Name,Value)

ISet = instcbond(ISet,CouponRate,Settle,Maturity,ConvRatio)
ISet = instcbond( ___ ,Name,Value)

[FieldList,ClassList,TypeString] = instcbond
```

Description

`ISet = instcbond(CouponRate,Settle,Maturity,ConvRatio)` creates a `CBond` instrument variable from data arrays.

`ISet = instcbond( ___ ,Name,Value)` creates a `CBond` instrument variable from data arrays using optional name-value pair arguments.

`ISet = instcbond(ISet,CouponRate,Settle,Maturity,ConvRatio)` adds a `CBond` to an existing instrument set.

`ISet = instcbond( ___ ,Name,Value)` adds a `CBond` instrument to an existing instrument set using optional name-value pair arguments.

`[FieldList,ClassList,TypeString] = instcbond` lists the field metadata for the `CBond` instrument.

Examples

Create a CBond Instrument

Create a `CBond` instrument.

```
CouponRate = 0.03;
Settle = datetime(2014,1,1);
Maturity = datetime(2016,1,1);
CallStrike = 125;
CallExDates = [datetime(2015,1,1) datetime(2016,1,1)];

ConvRatio = 1.5;
Spread = 0.045;

InstSet = instcbond(CouponRate,Settle,Maturity,ConvRatio,...
'Spread',Spread,'CallExDates',CallExDates,'CallStrike',CallStrike,...
'AmericanCall', 1);
```

Display the `InstSet` for the convertible bond.

```
instdisp(InstSet)
```

```
Index Type  CouponRate Settle         Maturity       ConvRatio Period IssueDate FirstCouponDate
1     CBond 0.03       01-Jan-2014    01-Jan-2016    1.5       2      NaN       NaN
```

Add a CBond Instrument to an Existing Portfolio Set

Create a bond instrument using `instbond`.

```
CouponRate= [0.035;0.04];
Settle= 'Nov-1-2013';
Maturity = 'Nov-1-2014';
Period =1;

InstSet = instbond(CouponRate,Settle,Maturity, ...
Period);
```

Add a `CBond` instrument to the existing portfolio set.

```
ConvRatio = 1.5;
InstSet = instadd(InstSet,'CBond',CouponRate,Settle,Maturity,ConvRatio);
instdisp(InstSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCoup
1     Bond 0.035      01-Nov-2013    01-Nov-2014    1      0     1            NaN       NaN
2     Bond 0.04       01-Nov-2013    01-Nov-2014    1      0     1            NaN       NaN

Index Type  CouponRate Settle         Maturity       ConvRatio Period IssueDate FirstCouponDate
3     CBond 0.035      01-Nov-2013    01-Nov-2014    1.5       2      NaN       NaN
4     CBond 0.04       01-Nov-2013    01-Nov-2014    1.5       2      NaN       NaN
```

```
[FieldList,ClassList,TypeString] = instcbond
```

```
FieldList = 20x1 cell
    {'CouponRate'        }
    {'Settle'            }
    {'Maturity'          }
    {'ConvRatio'         }
    {'Period'            }
    {'IssueDate'         }
    {'FirstCouponDate'   }
    {'LastCouponDate'    }
    {'StartDate'         }
    {'Face'              }
    {'Spread'            }
    {'CallStrike'        }
    {'CallExDates'       }
    {'AmericanCall'      }
    {'PutStrike'         }
    {'PutExDates'        }
    {'AmericanPut'       }
    {'ConvDates'         }
    {'DefaultProbability'}
    {'RecoveryRate'      }
```

```
ClassList = 20x1 cell
    {'cell'}
    {'date'}
    {'date'}
    {'dble'}
    {'dble'}
    {'date'}
    {'date'}
    {'date'}
    {'date'}
    {'cell'}
    {'dble'}
    {'dble'}
    {'date'}
    {'dble'}
    {'dble'}
    {'date'}
    {'dble'}
    {'date'}
    {'dble'}
    {'dble'}


TypeString =
'CBond'
```

Input Arguments

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` positive decimal annual rate or an `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every convertible bond is set to the `ValuationDate` of the stock tree. The bond argument, `Settle`, is ignored.

To support existing code, `instcbond` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, instcbond also accepts serial date numbers as inputs, but they are not recommended.

ConvRatio — Number of shares convertible to one bond
nonnegative scalar

Number of shares convertible to one bond, specified as an NINST-by-1 nonnegative vector.

Data Types: double

ISet — Variable containing a collection of instruments
structure

Variable containing a collection of instruments, specified as a structure. Use thus argument to add a CBond (convertible bond) to an existing instrument set (ISet). Instruments within ISet are broken down by type, and each type can have different data fields. For more information on theISet variable, see instget.

Data Types: struct

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: InstSet = instcbond(CouponRate,Settle,Maturity,ConvRatio,'Spread',Spread,'CallExDates', CallExDates,'CallStrike',CallStrike,'AmericanCall', 1)

Period — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of 'Period' and a NINST-by-1 vector.

Data Types: double

IssueDate — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of 'IssueDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, instcbond also accepts serial date numbers as inputs, but they are not recommended.

FirstCouponDate — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-1 vector using a datetime array, string array, or date character vectors

To support existing code, `instcbond` also accepts serial date numbers as inputs, but they are not recommended.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `instcbond` also accepts serial date numbers as inputs, but they are not recommended.

`Face` — Face value
`100` (default) | scalar of nonnegative value | cell array of nonnegative values

Face value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-1 scalar of nonnegative face values or an `NINST`-by-1 cell array, where each element is a `NumDates`-by-2 cell array. The first column of the `NumDates`-by-2 cell array is dates and the second column is the associated face value. The date indicates the last day that the face value is valid.

Data Types: `cell` | `double`

`Spread` — Number of basis points over the reference rate
`0` (default) | vector

Number of basis points over the reference rate, specified as the comma-separated pair consisting of `'Spread'` and a `NINST`-by-1 vector.

Data Types: `double`

`CallStrike` — Call strike price for European, Bermuda, or American option
nonnegative integer | vector of nonnegative integers

Call strike price for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'CallStrike'` and one of the following values:

- For a European call option — `NINST`-by-1 vector of nonnegative integers
- For a Bermuda call option — `NINST`-by-`NSTRIKES` matrix of strike price values, where each row is the schedule for one call option. If a call option has fewer than `NSTRIKES` exercise opportunities, the end of the row is padded with `NaN`s.
- For an American call option — `NINST`-by-1 vector of strike price values for each call option.

Data Types: `double`

`CallExDates` — Call exercise date for European, Bermuda, or American option
datetime array | string array | date character vector

Call exercise date for European, Bermuda, or American option, specified as the comma-separated pair consisting of `'CallExDates'` and a datetime array, string array, or date character vectors for one of the following values:

- For a European option — NINST-by-1 vector of date character vectors.
- For a Bermuda option — NINST-by-NSTRIKES matrix of exercise dates, where each row is the schedule for one call option. For a European option, there is only one CallExDate on the option expiry date.
- For an American option — NINST-by-1 or NINST-by-2 matrix of exercise date boundaries. For each instrument, the call option can be exercised on any tree date between or including the pair of dates on that row. If CallExDates is NINST-by-1, the call option can be exercised between the ValuationDate of the stock tree and the single listed CallExDate.

To support existing code, instcbond also accepts serial date numbers as inputs, but they are not recommended.

AmericanCall — Call option type indicator
0 if AmericanCall is NaN or not entered (default) | scalar | vector of positive integers[0,1]

Call option type, specified as the comma-separated pair consisting of 'AmericanCall' and a NINST-by-1 vector with positive integer flags with values 0 or 1.

- For a European or Bermuda option — AmericanCall is 0 for each European or Bermuda option.
- For an American option — AmericanCall is 1 for each American option. The AmericanCall argument is required to invoke American exercise rules.

Data Types: double

PutStrike — Put strike values for European, Bermuda, or American option
scalar | vector of positive integers[0,1]

Put strike values for a European, Bermuda, or American option, specified as the comma-separated pair consisting of 'PutStrike' and one of the following values:

- For a European put option — NINST-by-1 vector of nonnegative integers
- For a Bermuda put option — NINST-by-NSTRIKES matrix of strike price values, where each row is the schedule for one put option. If a put option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- For an American put option — NINST-by-1 vector of strike price values for each put option.

Data Types: double

PutExDates — Put exercise date for European, Bermuda, or American option
datetime array | string array | date character vector

Put exercise date for a European, Bermuda, or American option, specified as the comma-separated pair consisting of 'PutExDates' and a datetime array, string array, or date character vectors for one of the following values:

- For a European option — NINST-by-1 vector date character vectors.
- For a Bermuda option — NINST-by-NSTRIKES matrix of exercise dates, where each row is the schedule for one put option. For a European option, there is only one PutExDate on the option expiry date.
- For an American option — NINST-by-1 or NINST-by-2 matrix of exercise date boundaries. For each instrument, the put option can be exercised on any tree date between or including the pair of dates on that row. If PutExDates is NINST-by-1, the put option can be exercised between the ValuationDate of the stock tree and the single listed PutExDate.

To support existing code, `instcbond` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanPut` — Put option type indicator
`0` if `AmericanPut` is `NaN` or not entered (default) | scalar | vector of positive integers`[0,1]`

Put option type, specified as the comma-separated pair consisting of `'AmericanPut'` and a `NINST`-by-`1` vector with positive integer flags with values `0` or `1`.

- For a European or Bermuda option — `AmericanPut` is `0` for each European or Bermuda option.
- For an American option — `AmericanPut` is `1` for each American option. The `AmericanPut` argument is required to invoke American exercise rules.

Data Types: `double`

`ConvDates` — Convertible dates
`MaturityDate` (default) | datetime array | string array | date character vector

Convertible dates, specified as the comma-separated pair consisting of `'ConvDates'` and a `NINST`-by-`1` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors. If `ConvDates` is not specified, the bond is always convertible until maturity.

To support existing code, `instcbond` also accepts serial date numbers as inputs, but they are not recommended.

For each instrument, the bond can be converted on any tree date between or including the pair of dates on that row.

If `ConvDates` is `NINST`-by-`1`, the bond can be converted between the `ValuationDate` of the stock tree and the single listed `ConvDates`.

Output Arguments

`ISet` — Variable containing a collection of instruments
character vector | row vector

Variable containing a collection of instruments, returned as a row vector or character vector for each instrument. Instruments are broken down by type and each type can have different data fields. For more information on the`ISet` variable, see `instget`.

`FieldList` — Name of each data field for instrument type
cell array of character vectors

Name of each data field for instrument type, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class of each field
cell array of character vectors with valid values of `'dble'`, `'date'`, and `'char'`

Data class of each field, returned as an `NFIELDS`-by-`1` cell array of character vectors with valid character vector values of `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument added
character vector

Type of instrument added, returned as character vector. When adding a `CBond`, the `TypeString = 'CBond'`.

Version History

Introduced in R2015a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instcbond` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cbondbycrr` | `instadd` | `instdisp` | `cbondbyeqp` | `crrprice` | `eqpprice` | `eqpsens` | `crrsens`

Topics

“Convertible Bond” on page 2-4

instcf

Construct cash flow instrument

Syntax

```
InstSet = instcf(CFlowAmounts,CFlowDates,Settle)
InstSet = instcf(InstSet,CFlowAmounts,CFlowDates,Settle)
InstSet = instcf( ___ ,Basis)
[FieldList,ClassList,TypeString] = instcf
```

Description

`InstSet = instcf(CFlowAmounts,CFlowDates,Settle)` creates a new instrument set containing CashFlow instruments.

`InstSet = instcf(InstSet,CFlowAmounts,CFlowDates,Settle)` adds CashFlow instruments to an existing instrument set.

`InstSet = instcf( ___ ,Basis)` adds an optional argument.

`[FieldList,ClassList,TypeString] = instcf` lists field meta-data for the CashFlow instrument.

Examples

Create Two Cash Flow Instruments

Create a new instrument variable with the following information:

```
CFlowAmounts =[5 NaN 5.5 105; 5 0 6 105];
CFlowDates = [732678, NaN, 733408,733774;
              732678, 733034, 733408, 734774];
Settle= 'Jan-1-2015';
Basis = 1;
```

Create the new cash flow instruments.

```
InstSet = instcf(CFlowAmounts,CFlowDates,Settle,Basis)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'CashFlow'}
     FieldName: {{4x1 cell}}
    FieldClass: {{4x1 cell}}
     FieldData: {{4x1 cell}}
```

Display the cash flow instruments.

```
instdisp(InstSet)
```

```
Index Type     CFlowAmounts                                CFlowDates
1     CashFlow 5           NaN           5.5           105 01-Jan-2006      NaN        01-Jan-200
2     CashFlow 5             0             6           105 01-Jan-2006   23-Dec-2006   01-Jan-200
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding CashFlow instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`CFlowAmounts` — Cash flow amounts

matrix

Cash flow amounts, specified as a 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.

Data Types: `double`

`CFlowDates` — Cash flow dates

serial date number

Cash flow dates, specified as an `NINST`-by-`MOSTCFS` vector using serial date numbers. Each entry contains the date of the corresponding cash flow in `CFlowAmounts`.

Data Types: `double`

`Settle` — Settlement date

serial date number | date character vector

Settlement date on which the cash flows are priced, specified as a scalar serial date number or date character vector.

Data Types: `double` | `char`

`Basis` — Day-count basis

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis, specified as scalar or an `NINST`-by-`1` vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

`InstSet` — Variable containing a collection of instruments

structure

Variable containing a collection of instruments, returned as a structure. 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 `InstSet` variable, see `instget`.

`FieldList` — Name of each data field for CashFlow instrument

cell array of character vectors

Name of each data field for a CashFlow instrument, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class for each field

cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-`1` cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument

character vector

Type of instrument, returned as a character vector. For a CashFlow instrument, `TypeString = 'CashFlow'`.

Version History

Introduced before R2006a

See Also

`instadd` | `instdisp` | `instget` | `intenvprice`

Topics

“Creating Instruments or Properties” on page 1-16
“Instrument Constructors” on page 1-15
“Supported Interest-Rate Instrument Functions” on page 2-3

instcompound

Construct compound option

Syntax

```
InstSet = instcompound(UOptSpec,UStrike,USettle,UExerciseDates,UAmericanOpt,
COptSpec,CStrike,CSettle,CExerciseDates,CAmericanOpt)
InstSet = instcompound(InstSet,UOptSpec,UStrike,USettle,UExerciseDates,
UAmericanOpt,COptSpec,CStrike,CSettle,CExerciseDates,CAmericanOpt)
[FieldList,ClassList,TypeString] = instcompound
```

Description

`InstSet = instcompound(UOptSpec,UStrike,USettle,UExerciseDates,UAmericanOpt, COptSpec,CStrike,CSettle,CExerciseDates,CAmericanOpt)` creates a new instrument set containing Compound option instruments.

`InstSet = instcompound(InstSet,UOptSpec,UStrike,USettle,UExerciseDates, UAmericanOpt,COptSpec,CStrike,CSettle,CExerciseDates,CAmericanOpt)` adds Compound option instruments to an existing instrument set.

`[FieldList,ClassList,TypeString] = instcompound` lists field meta-data for the Compound option instrument.

Examples

Create a Compound Option Instrument

Define a compound option instrument with the following data:

```
UOptSpec = 'Call';
UStrike = 130;
USettle = datetime(2012,1,1);
UExerciseDates = datetime(2015,1,1);
UAmericanOpt = 0;
COptSpec = 'Put';
CStrike = 5;
CSettle = datetime(2012,1,1);
CExerciseDates = datetime(2014,1,1);
CAmericanOpt = 0;

InstSet = instcompound(UOptSpec, UStrike, USettle,UExerciseDates, ...
UAmericanOpt, COptSpec, CStrike, CSettle,CExerciseDates, CAmericanOpt)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Compound'}
     FieldName: {{10x1 cell}}
    FieldClass: {{10x1 cell}}
```

```
     FieldData: {{10x1 cell}}
```

```
InstSet = instcompound(UOptSpec, UStrike, USettle,UExerciseDates, ...
UAmericanOpt, COptSpec, CStrike, CSettle,CExerciseDates)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Compound'}
     FieldName: {{10x1 cell}}
    FieldClass: {{10x1 cell}}
     FieldData: {{10x1 cell}}
```

Display the instrument set.

```
instdisp(InstSet)
```

```
Index Type     UOptSpec UStrike USettle        UExerciseDates UAmericanOpt COptSpec CStrike CSet
1     Compound Call     130     01-Jan-2012    01-Jan-2015    0            Put      5       01-Ja
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding Compound option instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`UOptSpec` — Definition of underlying option

character vector with value `'call'` or `'put'`

Definition of underlying option, specified as `'call'` or `'put'` using a character vector.

Data Types: `char`

`UStrike` — Underlying option strike price value

nonnegative integer

Underlying option strike price value, specified with a nonnegative integer using a 1-by-1 vector.

Data Types: `double`

`USettle` — Underlying option settlement date or trade date

datetime array | string array | date character vector

Underlying option settlement date or trade date, specified as a 1-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `instcompound` also accepts serial date numbers as inputs, but they are not recommended.

`UExerciseDates` — Underlying option exercise date
datetime array | string array | date character vector

Underlying option exercise date, specified as a datetime array, string array, or date character vectors:

- For a European option, use a `1`-by-`1` vector of the underlying exercise date. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `1`-by-`2` vector of the underlying 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 `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `instcompound` also accepts serial date numbers as inputs, but they are not recommended.

`UAmericanOpt` — Underlying option type
`0` European (default) | scalar with values `0` or `1`

Underlying option type, specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

If `UAmericanOpt` is a `NaN` or is unspecified, the option is a European option.

Data Types: `double`

`COptSpec` — Definition of compound option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of compound option, specified as `'call'` or `'put'` using a character vector or a cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`CStrike` — Compound option strike price values
nonnegative integers

Compound option strike price values for a European and American option, specified with a nonnegative integer using a `NINST`-by-`1` matrix. Each row is the schedule for one option.

Data Types: `double`

`CSettle` — Compound option settlement date or trade date
datetime array | string array | date character vector

Compound option settlement date or trade date, specified as a `1`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instcompound` also accepts serial date numbers as inputs, but they are not recommended.

`CExerciseDates` — Compound option exercise dates
datetime array | string array | date character vector

Compound option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a`NINST`-by-`1` matrix of the compound exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of the compound 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 `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `instcompound` also accepts serial date numbers as inputs, but they are not recommended.

`CAmericanOpt` — Compound option type
`0` European (default) | scalar with values `0` or `1`

(Optional) Compound option type, specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

If `CAmericanOpt` is a `NaN` or is unspecified, the option is a European option.

Data Types: `double`

Output Arguments

`InstSet` — Variable containing a collection of instruments
structure

Variable containing a collection of instruments, returned as a structure. 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 `InstSet` variable, see `instget`.

`FieldList` — Name of each data field for Compound option instrument
cell array of character vectors

Name of each data field for a Compound option instrument, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class for each field
cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-`1` cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument
character vector

Type of instrument, returned as a character vector. For a Compound option instrument, `TypeString = 'Compound'`.

More About

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. For more information, see “Compound Option” on page 3-23.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instcompound` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`instadd` | `instdisp` | `instget`

Topics

“Creating Instruments or Properties” on page 1-16
“Pricing Equity Derivatives Using Trees” on page 3-64
“Compound Option” on page 3-23
“Supported Equity Derivative Functions” on page 3-19

instdelete

Complement of instrument set by matching conditions

Syntax

```
ISubSet = instdelete(InstSet,Name,Value)
```

Description

`ISubSet = instdelete(InstSet,Name,Value)` deletes instruments are from `ISubSet` if all the name-value pairs `Field`, `Index`, and `Type` conditions are met. An instrument meets an individual `Field` condition if the stored data matches any of the rows listed in the `Data`.

Examples

Remove Instrument from Instrument Set

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
```

Input Arguments

`InstSet` — Instrument variable for collection of instruments

structure

Instrument variable for a collection of instruments, specified as an instrument set structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ISet = instdelete(ExampleInst,'Type','Option')`

`FieldName` — Number of fields

vector

Number of fields, specified as the comma-separated pair consisting of `'FieldName'` and an `NFIELDS`-by-`1` cell array of character vectors listing the name of each data field to match with data values.

Data Types: `char` | `cell`

`Data` — Number of values

vector

Number of values, specified as the comma-separated pair consisting of `'Data'` and a `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 `FieldName`. The number of columns is arbitrary and matching ignores trailing `NaN`s or spaces.

Data Types: `char` | `cell`

`Index` — Number of instruments

vector

Number of instruments, specified as the comma-separated pair consisting of `'Index'` and a `NINST`-by-`1` vector restricting positions of instruments to check for matches. The default is all indices available in the instrument variable.

Data Types: `char` | `cell`

`Type` — Number of types

vector

Number of types, specified as the comma-separated pair consisting of `'Type'` and a `NTYPES`-by-`1` cell array of character vectors restricting instruments to match one of types. The default is all types in the instrument variable.

Data Types: `char` | `cell`

Output Arguments

`ISubSet` — Updated variable containing a collection of instruments

structure

Updated variable containing a collection of instruments, returned as an instrument set structure. `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 data matches any of the rows listed in the `Data`. See `instfind` for more examples on matching criteria.

Version History

Introduced before R2006a

See Also

`instaddfield` | `instfind` | `instget` | `instselect`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15

instdisp

Display instruments

Syntax

```
CharTable = instdisp(InstSet)
```

Description

`CharTable = instdisp(InstSet)` creates a character array displaying the contents of an instrument collection `InstSet`. If `instdisp` is called without an output argument, the table is displayed in the Command Window.

Note When using `instdisp`, a value of `NaN` in one of the columns for an instrument indicates that the default value for that parameter will be used in the instrument's pricing function.

Examples

Retrieve Instrument from Instrument Set

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
```

Create a swap instrument and use `instdisp` to display the instrument. Notice that value of `NaN` in two columns for this instrument indicates that the default values for `LegReset` and `LegType` parameters will be used in the swap instrument's pricing function.

```
LegRate1 = [0.065, 0];
Settle1 = datetime(2007,1,1);
Maturity1 = datetime(2012,1,1);
```

```
ISet = instswap(LegRate1, Settle1, Maturity1);
instdisp(ISet)

Index Type LegRate    Settle         Maturity       LegReset Basis Principal LegType EndMonthRule
1     Swap [0.065  0] 01-Jan-2007    01-Jan-2012    [NaN]    0     100       [NaN]   1
```

Input Arguments

`InstSet` — Instrument variable for collection of instruments

structure

Instrument variable for a collection of instruments, specified as an instrument set structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

Output Arguments

`CharTable` — Table of instruments

array

Table of instruments, returned as a character array. 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.

Version History

Introduced before R2006a

See Also

`datestr` | `num2str` | `instaddfield` | `instget` | `instcbond`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15

instfields

List field names

Syntax

```
FieldList = instfields(InstSet,Name,Value)
```

Description

`FieldList = instfields(InstSet,Name,Value)` retrieves the list of fields stored in an instrument variable for the name-value pair argument `Type`.

Examples

Obtain Field Information for Instrument in Instrument Set

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
```

Get the fields listed for type `'Option'`.

```
[FieldList, ClassList] = instfields(ExampleInst, 'Type','Option')
```

```
FieldList = 4x1 cell
    {'Strike'   }
    {'Price'    }
    {'Opt'      }
    {'Contracts'}


ClassList = 4x1 cell
    {'dble'}
    {'dble'}
```

```
    {'char'}
    {'dble'}
```

Get the fields listed for types `'Option'` and `'TBill'`.

```
FieldList = instfields(ExampleInst, 'Type', {'Option', 'TBill'})
```

```
FieldList = 5x1 cell
    {'Strike'   }
    {'Opt'      }
    {'Price'    }
    {'Maturity' }
    {'Contracts'}
```

Get all the fields listed in any type in the variable.

```
FieldList = instfields(ExampleInst)
```

```
FieldList = 7x1 cell
    {'Delivery' }
    {'F'        }
    {'Strike'   }
    {'Opt'      }
    {'Price'    }
    {'Maturity' }
    {'Contracts'}
```

Input Arguments

`InstSet` — Instrument variable for collection of instruments

structure

Instrument variable for a collection of instruments, specified as an instrument set structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `FieldList = instfields(ExampleInst,'Type','Option')`

Type — Number of types

vector

Number of types, specified as the comma-separated pair consisting of `'Type'` and a `NTYPES`-by-`1` cell array of character vectors restricting instruments to match one of the types. The default is all types in the instrument variable.

Data Types: `char` | `cell`

Output Arguments

`FieldList` — Number of fields

structure

Number of fields, returned as an `NFIELDS`-by-`1` cell array of character vectors listing the name of each data field corresponding to the listed `Type`.

Version History

Introduced before R2006a

See Also

`instdisp` | `instlength` | `insttypes`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15

instfind

Search instruments for matching conditions

Syntax

```
IndexMatch = instfind(InstSet,'Field','Data')
IndexMatch = instfind( ___ ,Name,Value)
```

Description

`IndexMatch = instfind(InstSet,'Field','Data')` returns indices of instruments matching name-value pair arguments for `'Field'` and `'Data'`.

`IndexMatch = instfind( ___ ,Name,Value)` adds optional name-value pair arguments for `Index` and `Type`.

Examples

Search Instruments in Instrument Set for Matching Information

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
```

Make a vector, `Opt95`, containing the indexes within `ExampleInst` of the options struck at `95`.

```
Opt95 = instfind(ExampleInst, 'FieldName','Strike','Data','95')
```

```
Opt95 = 2×1

     1
     6
```

Locate the futures and Treasury bill instruments within `ExampleInst`.

```
Types = instfind(ExampleInst,'Type',{'Futures';'TBill'})
```

```
Types = 2×1

     4
     7
```

Input Arguments

`InstSet` — Instrument variable for collection of instruments

structure

Instrument variable for a collection of instruments, specified as an instrument set structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `IndexMatch = instfind(ExampleInst,'Type',{'Futures';'TBill'})`

`Field` — Number of fields

vector

Number of fields, specified as the comma-separated pair consisting of `'FieldName'` and an `NFIELDS`-by-`1` cell array of character vectors listing the name of each data field to match with data values.

Data Types: `char` | `cell`

`Data` — Number of values

vector

Number of values, specified as the comma-separated pair consisting of `'Data'` and a `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 `FieldName`. The number of columns is arbitrary and matching ignores trailing `NaN`s or spaces.

Data Types: `char` | `cell`

`Index` — Number of instruments

vector

Number of instruments, specified as the comma-separated pair consisting of `'Index'` and a `NINST`-by-`1` vector restricting positions of instruments to check for matches. The default is all indices available in the instrument variable.

Data Types: `char` | `cell`

Type — Number of types
vector

Number of types, specified as the comma-separated pair consisting of `'Type'` and a `NTYPES`-by-`1` cell array of character vectors restricting instruments to match one of types. The default is all types in the instrument variable.

Data Types: `char` | `cell`

Output Arguments

`IndexMatch` — Positions of instruments matching the input criteria
structure

Positions of instruments matching the input criteria, returned as 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.

Version History

Introduced before R2006a

See Also

`instaddfield` | `instget` | `instgetcell` | `instselect`

Topics
“Portfolio Creation Using Functions” on page 1-6
“Searching or Subsetting a Portfolio” on page 1-17
“Instrument Constructors” on page 1-15

instfixed

Construct fixed-rate instrument

Syntax

```
InstSet = instfixed(CouponRate,Settle,Maturity)
InstSet = instfixed(InstSet,CouponRate,Settle,Maturity)
InstSet = instfixed( ___ ,FixedReset,Basis,Principal,EndMonthRule)
[FieldList,ClassList,TypeString] = instfixed
```

Description

`InstSet = instfixed(CouponRate,Settle,Maturity)` creates a new instrument set containing Fixed-Rate instruments.

`InstSet = instfixed(InstSet,CouponRate,Settle,Maturity)` adds Fixed-Rate instruments to an existing instrument set.

`InstSet = instfixed( ___ ,FixedReset,Basis,Principal,EndMonthRule)` adds optional arguments.

`[FieldList,ClassList,TypeString] = instfixed` lists field meta-data for the Fixed-Rate instrument.

Examples

Create a Fixed-Rate Instrument

Define the characteristics of the fixed-rate instrument.

```
CouponRate = .03;
Settle = datetime(2013,3,15);
Maturity = datetime(2018,3,15);
FixedReset = 4;
Basis = 1;
Principal = 1000;
EndMonthRule = 1;
```

Create the new cap instrument.

```
ISet = instfixed(CouponRate, Settle, Maturity, FixedReset, Basis, Principal,EndMonthRule)
```

```
ISet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Fixed'}
     FieldName: {{7x1 cell}}
    FieldClass: {{7x1 cell}}
     FieldData: {{7x1 cell}}
```

Display the fixed-rate instrument.

```
instdisp(ISet)
```

```
Index Type  CouponRate Settle         Maturity       FixedReset Basis Principal EndMonthRule
1     Fixed 0.03       15-Mar-2013    15-Mar-2018    4          1     1000      1
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding Cap instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`CouponRate` — Coupon annual rate

decimal

Coupon annual rate, specified as a scalar or an `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date for the fixed-rate instrument, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instfixed` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each fixed-rate note.

To support existing code, `instfixed` also accepts serial date numbers as inputs, but they are not recommended.

`FixedReset` — Frequency of payments per year

`1` (default) | vector

(Optional) Frequency of payments per year, specified as a scalar or an `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day count basis

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day count basis, specified as a scalar or an `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

(Optional) Notional principal amounts, specified as a scalar, vector, or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array, where each element of the cell array is a `NumDates`-by-`2` cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

(Optional) End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as a nonnegative integer `0` or `1` using a scalar or an `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`InstSet` — Variable containing a collection of instruments
structure

Variable containing a collection of instruments, returned as a structure. 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 `InstSet` variable, see `instget`.

`FieldList` — Name of each data field for Fixed-Rate instrument
cell array of character vectors

Name of each data field for a Fixed-Rate instrument, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class for each field
cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-`1` cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument
character vector

Type of instrument, returned as a character vector. For a Fixed-Rate instrument, `TypeString = 'Fixed'`.

More About

Fixed-Rate Note

A fixed-rate note is a long-term debt security with a preset interest rate and maturity, by which the interest must be paid.

The principal may or may not be paid at maturity. In Financial Instruments Toolbox, the principal is always paid at maturity. For more information, see “Fixed-Rate Note” on page 2-9.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instfixed` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmprice` | `instaddfield` | `instbond` | `instcap` | `instdisp` | `instswap` | `intenvprice`

Topics

“Creating Instruments or Properties” on page 1-16
“Supported Interest-Rate Instrument Functions” on page 2-3
“Choose Instruments, Models, and Pricers” on page 1-53

instfloat

Construct floating-rate instrument

Syntax

```
InstSet = instfloat(Spread,Settle,Maturity)
InstSet = instfloat(InstSet,Spread,Settle,Maturity)
InstSet = instfloat( ___ ,FloatReset,Basis,Principal,EndMonthRule,CapRate,
FloorRate)
[FieldList,ClassList,TypeString] = instfixed
```

Description

`InstSet = instfloat(Spread,Settle,Maturity)` creates a new instrument set containing Float instruments.

`InstSet = instfloat(InstSet,Spread,Settle,Maturity)` adds Float instruments to an existing instrument set.

`InstSet = instfloat( ___ ,FloatReset,Basis,Principal,EndMonthRule,CapRate, FloorRate)` adds optional arguments.

`[FieldList,ClassList,TypeString] = instfixed` lists field meta-data for the Float instrument.

Examples

Create a Floating-Rate Instrument

Define the characteristics of the floating-rate instrument.

```
Spread = 2;
Settle = datetime(2013,3,15);
Maturity = datetime(2013,3,18);
FloatReset = 4;
Basis = 1;
Principal = 1000;
EndMonthRule = 1;
CapRate = 0.35;
FloorRate = 0.27;
```

Create the new floating-rate instrument.

```
ISet = instfloat(Spread, Settle, Maturity, FloatReset, Basis, Principal, ...
EndMonthRule, CapRate, FloorRate)
```

```
ISet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Float'}
```

```
     FieldName: {{9x1 cell}}
    FieldClass: {{9x1 cell}}
     FieldData: {{9x1 cell}}
```

Display the floating-rate instrument.

```
instdisp(ISet)
```

```
Index Type  Spread Settle         Maturity       FloatReset Basis Principal EndMonthRule CapRate
1     Float 2      15-Mar-2013    18-Mar-2013    4          1     1000      1            0.35
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding Float instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`Spread` — Number of basis points over the reference rate

vector

Number of basis points over the reference rate, specified as a scalar or an `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instfloat` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each floating-rate note.

To support existing code, `instfloat` also accepts serial date numbers as inputs, but they are not recommended.

`FloatReset` — Frequency of payments per year

`1` (default) | vector

(Optional) Frequency of payments per year, specified as a scalar or an `NINST`-by-`1` vector.

Data Types: `double`

Basis — Day count basis
0 (actual/actual) (default) | integer from 0 to 13

(Optional) Day count basis, specified as a scalar or an NINST-by-1 vector.

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

For more information, see “Basis” on page 2-228.

Data Types: double

Principal — Notional principal amounts or principal value schedules
100 (default) | vector or cell array

(Optional) Notional principal amounts, specified as a scalar, vector, or cell array.

Principal accepts a NINST-by-1 vector or NINST-by-1 cell array, where each element of the cell array is a NumDates-by-2 cell array and the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: cell | double

EndMonthRule — End-of-month rule flag for generating dates when Maturity is end-of-month date for month having 30 or fewer days
1 (in effect) (default) | nonnegative integer [0,1]

(Optional) End-of-month rule flag for generating dates when Maturity is an end-of-month date for a month having 30 or fewer days, specified a scalar nonnegative integer 0 or 1 or an NINST-by-1 vector.

- 0 = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- 1 = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: logical

CapRate — Annual cap rate
decimal

(Optional) Annual cap rate, specified as a scalar or an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array, and the cell array first column is dates, and the second column is associated cap rates. The date indicates the last day that the cap rate is valid.

Data Types: `double` | `cell`

`FloorRate` — Annual floor rate
decimal

(Optional) Annual floor rate, specified as a scalar or an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array, and the cell array first column is dates, and the second column is associated floor rates. The date indicates the last day that the floor rate is valid.

Data Types: `double` | `cell`

Output Arguments

`InstSet` — Variable containing a collection of instruments
structure

Variable containing a collection of instruments, returned as a structure. 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 `InstSet` variable, see `instget`.

`FieldList` — Name of each data field for Float instrument
cell array of character vectors

Name of each data field for a Float instrument, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class for each field
cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-`1` cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument
character vector

Type of instrument, returned as a character vector. For a Float instrument, `TypeString = 'Float'`.

More About

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.

Version History

Introduced in R2012b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instfloat` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmprice` | `instaddfield` | `instbond` | `instcap` | `instdisp` | `instswap` | `intenvprice`

Topics

“Creating Instruments or Properties” on page 1-16
“Supported Interest-Rate Instrument Functions” on page 2-3
“Choose Instruments, Models, and Pricers” on page 1-53

instfloor

Construct floor instrument

Syntax

```
InstSet = instfloor(Strike,Settle,Maturity)
InstSet = instfloor(InstSet,Strike,Settle,Maturity)
InstSet = instfloor( ___ ,FloorReset,Basis,Principal)
[FieldList,ClassList,TypeString] = instfloor
```

Description

`InstSet = instfloor(Strike,Settle,Maturity)` creates a new instrument set containing Floor instruments.

`InstSet = instfloor(InstSet,Strike,Settle,Maturity)` adds Floor instruments to an existing instrument set.

`InstSet = instfloor( ___ ,FloorReset,Basis,Principal)` adds optional arguments.

`[FieldList,ClassList,TypeString] = instfloor` lists field meta-data for the Floor instrument.

Examples

Create a Floor Instrument

Define the characteristics of the floor instrument.

```
Strike = 0.22;
Settle = datetime(2013,3,15);
Maturity = datetime(2018,3,15);
FloorReset = 4;
Basis = 1;
Principal = 1000;
```

Create the new floor instrument.

```
ISet = instfloor(Strike, Settle, Maturity, FloorReset, Basis, Principal)
```

```
ISet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Floor'}
     FieldName: {{6x1 cell}}
    FieldClass: {{6x1 cell}}
     FieldData: {{6x1 cell}}
```

Display the floor instrument.

```
instdisp(ISet)

Index Type  Strike Settle         Maturity       FloorReset Basis Principal
1     Floor 0.22   15-Mar-2013    15-Mar-2018    4          1     1000
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding Floor instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`Strike` — Rate at which floor is exercised

decimal

Rate at which the floor is exercised, specified as a scalar or an `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date for floor

datetime array | string array | date character vector

Settlement date for the floor, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instfloor` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for floor

datetime array | string array | date character vector

Maturity date for the floor, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instfloor` also accepts serial date numbers as inputs, but they are not recommended.

`FloorReset` — Reset frequency payment per year

`1` (default) | numeric

(Optional) Reset frequency payment per year, specified as a scalar or an `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis, specified as a scalar or an `NINST`-by-`1` vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a scalar or an `NINST`-by-`1` of notional principal amounts, or a `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Use `Principal` to pass a schedule to compute the price for an amortizing floor.

Data Types: `double` | `cell`

Output Arguments

`InstSet` — Variable containing a collection of instruments
structure

Variable containing a collection of instruments, returned as a structure. 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 `InstSet` variable, see `instget`.

`FieldList` — Name of each data field for Floor instrument
cell array of character vectors

Name of each data field for a Floor instrument, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class for each field
cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-`1` cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

TypeString — Type of instrument
character vector

Type of instrument, returned as a character vector. For a Floor instrument, `TypeString = 'Floor'`.

More About

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:

$\max(FloorRate - CurrentRate, 0)$

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instfloor` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmprice` | `instaddfield` | `instbond` | `instcap` | `instdisp` | `instswap` | `intenvprice`

Topics
“Creating Instruments or Properties” on page 1-16
“Supported Interest-Rate Instrument Functions” on page 2-3
“Choose Instruments, Models, and Pricers” on page 1-53

instget

Data from instrument variable

Syntax

```
Data = instget(InstSet,Name,Value)
```

Description

`Data = instget(InstSet,Name,Value)` retrieves `Data` from an instrument variable.

Examples

Retrieve Data Arrays From Instrument Variable

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 = 7×1

   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 = 7×1

   12.2000
    9.2000
    6.8000
       NaN
    7.4000
    2.9000
   99.0000


C = 7×1

           0
           0
        1000
       -1000
       -1000
           0
           6
```

Compute a value V. Create a new variable `ISet` that appends V to `ExampleInst`.

```
V = P.*C
```

```
V = 7×1

           0
           0
        6800
         NaN
       -7400
           0
         594
```

```
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 = 4×1

     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 = 4x7 char array
    'Option '
    'Futures'
    'Option '
    'TBill  '


O = 4x4 char array
    'Call'
    '    '
    'Put '
    '    '
```

Create a report of holdings `Type`, `Opt`, and `Value`.

```
rstring = [T, O, num2str(V(Ind))]
```

```
rstring = 4x16 char array
    'Option Call 6800'
    'Futures      NaN'
    'Option Put -7400'
    'TBill        594'
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of instruments, specified as `InstSet` structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Data = instget(ExampleInst,'FieldName','Price')`

`FieldName` — Name of each data field for instrument
all fields available for returned set of instruments (default) | cell array of character vectors

Name of each data field for an instrument, specified as the comma-separated pair consisting of `'FieldName'` and an `NFIELDS`-by-`1` cell array of character vectors. `FieldName` entries can also be either `Type` or `Index`; these return type strings and index numbers respectively.

Data Types: `char` | `cell`

`Index` — Number of instruments
all indices available in instrument variable (default) | vector

Number of instruments, specified as the comma-separated pair consisting of `'Index'` and an `NINST`-by-`1` vector of positions of instruments to work on. If `Type` is also entered, instruments referenced must be one of the types and contained in `Index`.

Data Types: `double`

`Type` — Number of types
all types in the instrument variable (default) | character vector

Number of types, specified as the comma-separated pair consisting of `'Type'` and a `NTYPES`-by-`1` cell array of character vectors restricting instruments worked on to match one of `Type` types.

Data Types: `char` | `cell`

Output Arguments

`Data` — Data contents
array

Data content, returned as an `NINST`-by-`M` array of data contents for the first field in `FieldName`. Each row corresponds to a separate instrument in the specified `Index`. Unavailable data is returned as `NaN` or as spaces.

Version History

Introduced before R2006a

See Also

`instaddfield` | `instdisp` | `intenvprice` | `instgetcell`

Topics
“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15

instgetcell

Data and context from instrument variable

Syntax

```
[DataList,FieldList,ClassList,IndexSet,TypeSet] = instgetcell(InstSet,
Name,Value)
```

Description

`[DataList,FieldList,ClassList,IndexSet,TypeSet] = instgetcell(InstSet, Name,Value)` retrieves data and context from an instrument variable.

Note `instgetcell` is best used for programming where the structure of the instrument variable is not known. `instget` gives more direct access to the data in a variable.

Examples

Retrieve Data and Context from Instrument Variable

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'}
```

```
FieldList = 2x1 cell
    {'Price'    }
    {'Contracts'}
```

```
DataList = instgetcell(ExampleInst, 'FieldName', FieldList )
```

```
DataList=2×1 cell array
    {7x1 double}
    {7x1 double}
```

```
P = DataList{1}
```

```
P = 7×1

   12.2000
    9.2000
    6.8000
       NaN
    7.4000
    2.9000
   99.0000
```

```
C = DataList{2}
```

```
C = 7×1

         0
         0
      1000
     -1000
     -1000
         0
         6
```

Get all the option data: `Strike`, `Price`, `Opt`, `Contracts`.

```
[DataList, FieldList, ClassList] = instgetcell(ExampleInst,'Type','Option')
```

```
DataList=4×1 cell array
    {5x1 double}
    {5x1 double}
    {5x4 char  }
    {5x1 double}


FieldList = 4x1 cell
    {'Strike'   }
    {'Price'    }
    {'Opt'      }
    {'Contracts'}


ClassList = 4x1 cell
    {'dble'}
    {'dble'}
    {'char'}
    {'dble'}
```

Look at the data as a comma-separated list.

```
DataList{:}
```

```
ans = 5×1

    95
   100
   105
   105
    95


ans = 5×1

   12.2000
    9.2000
    6.8000
    7.4000
    2.9000


ans = 5×4 char array
    'Call'
    'Call'
    'Call'
    'Put '
    'Put '


ans = 5×1

           0
           0
        1000
       -1000
           0
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of instruments, specified as `InstSet` structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `DataList = instgetcell(ExampleInst,'FieldName',FieldList)`

`FieldName` — Name of each data field for instrument
all fields available for returned set of instruments (default) | cell array of character vectors

Name of each data field for an instrument, specified as the comma-separated pair consisting of `'FieldName'` and an `NFIELDS`-by-`1` cell array of character vectors. `FieldName` entries can also be either `Type` or `Index`; these return type strings and index numbers respectively.

Data Types: `char` | `cell`

`Index` — Number of instruments
all indices available in instrument variable (default) | vector

Number of instruments, specified as the comma-separated pair consisting of `'Index'` and an `NINST`-by-`1` vector of positions of instruments to work on. If `Type` is also entered, instruments referenced must be one of the types and contained in `Index`.

Data Types: `double`

`Type` — Number of types
all types in the instrument variable (default) | character vector

Number of types, specified as the comma-separated pair consisting of `'Type'` and a `NTYPES`-by-`1` cell array of character vectors restricting instruments worked on to match one of `Type` types.

Data Types: `char` | `cell`

Output Arguments

`DataList` — Data contents for each field
cell array

Data content for each field, returned as 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` — Name of each field in `DataList`
cell array

Name of each field in `DataList`, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class of each field
array

Data class of each field, returned as an `NFIELDS`-by-`1` cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`IndexSet` — Positions of instruments
cell array

Positions of instruments, returned as an `NINST`-by-`1` vector of positions of instruments returned in `DataList`.

`TypeSet` — Type of each instrument
cell array

Type of each instrument, returned as an `NINST`-by-`1` cell array of character vectors listing the type of each instrument row returned in `DataList`.

Version History

Introduced before R2006a

See Also

`instaddfield` | `instdisp` | `instget`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15

instlength

Count instruments

Syntax

```
NInst = instlength(InstSet)
```

Description

`NInst = instlength(InstSet)` computes `NInst`, the number of instruments contained in the variable `InstSet`.

Examples

Count Instruments in Instrument Variable

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
```

Count the number of instruments contained in the variable `ExampleInst`.

```
NInst = instlength(ExampleInst)
```

```
NInst = 7
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of instruments, specified as `InstSet` structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

Output Arguments

`NInst` — Number of instruments in `InstSet`

numeric

Number of instruments in `InstSet`, returned as a numeric value.

Version History

Introduced before R2006a

See Also

`instdisp` | `instfields` | `insttypes`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15

instlookback

Construct lookback option

Syntax

```
InstSet = instlookback(OptSpec,Strike,Settle,ExerciseDates)
InstSet = instlookback(InstSet,OptSpec,Strike,Settle,ExerciseDates)
InstSet = instlookback(___,AmericanOpt)
[FieldList,ClassList,TypeString] = instlookback
```

Description

`InstSet = instlookback(OptSpec,Strike,Settle,ExerciseDates)` creates a new instrument set containing Lookback instruments.

`InstSet = instlookback(InstSet,OptSpec,Strike,Settle,ExerciseDates)` adds Lookback instruments to an existing instrument set.

`InstSet = instlookback(___,AmericanOpt)` adds an optional argument.

`[FieldList,ClassList,TypeString] = instlookback` lists field meta-data for the Lookback instrument.

Examples

Create a Lookback Option Instrument

Define a floating strike lookback instrument with the following data:

```
OptSpec = 'call';
Strike = NaN;
Settle = datetime(2012,1,1);
ExerciseDates = datetime(2015,1,1);
```

Create the instrument set.

```
InstSet = instlookback(OptSpec, Strike, Settle, ExerciseDates);
```

Display the lookback instrument.

```
instdisp(InstSet)
```

```
Index Type     OptSpec Strike Settle         ExerciseDates  AmericanOpt
1     Lookback call    NaN    01-Jan-2012    01-Jan-2015    0
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding Lookback instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a scalar `'call'` or `'put'` using a character vector or an `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price value

matrix of nonnegative integers

Option strike price value, specified as a scalar nonnegative integer or an `NINST`-by-`1` matrix of strike price values. Each row is the schedule for one option.

Data Types: `double`

`Settle` — Settlement date or trade date

datetime array | string array | date character vector

Settlement date or trade date for the lookback option, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instlookback` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors:

- For a European option, use an `NINST`-by-`1` vector of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. 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 a `NINST`-by-`1` vector of dates, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `instlookback` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type

`0` European (default) | integer with values `0` or `1`

(Optional) Option type, specified as a scalar or an `NINST`-by-`1` integer flags with values:

- `0` — European
- `1` — American

Data Types: `double`

Output Arguments

`InstSet` — Variable containing a collection of instruments

structure

Variable containing a collection of instruments, returned as a structure. 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 `InstSet` variable, see `instget`.

`FieldList` — Name of each data field for Lookback instrument

cell array of character vectors

Name of each data field for a Lookback instrument, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class for each field

cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-`1` cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument

character vector

Type of instrument, returned as a character vector. For a Lookback instrument, `TypeString = 'Lookback'`.

More About

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 Instruments 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. For more information, see “Lookback Option” on page 3-39.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instlookback` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`instadd` | `instdisp` | `instget`

Topics

“Pricing Equity Derivatives Using Trees” on page 3-64
“Creating Instruments or Properties” on page 1-16
“Supported Equity Derivative Functions” on page 3-19
“Choose Instruments, Models, and Pricers” on page 1-53

instoptbnd

Construct bond option

Syntax

```
InstSet = instoptbnd(BondIndex,OptSpec,Strike,ExerciseDates)
InstSet = instoptbnd(InstSet,BondIndex,OptSpec,Strike,ExerciseDates)
InstSet = instoptbnd(___,AmericanOpt)
[FieldList,ClassList,TypeString] = instoptbnd
```

Description

`InstSet = instoptbnd(BondIndex,OptSpec,Strike,ExerciseDates)` creates a new instrument set containing Bond option instruments.

`InstSet = instoptbnd(InstSet,BondIndex,OptSpec,Strike,ExerciseDates)` adds Bond option instruments to an existing instrument set.

`InstSet = instoptbnd(___,AmericanOpt)` adds an optional argument.

`[FieldList,ClassList,TypeString] = instoptbnd` lists field meta-data for the Bond option instrument.

Examples

Create a Bond Option Instrument

Create a new instrument variable with the following information:

```
BondIndex = 1;
OptSpec = 'call';
Strike= 85;
ExerciseDates = datetime(2014,11,1);
AmericanOpt = 1;
CouponRate= [0.035;0.04];
Settle= datetime(2013,11,1);
Maturity = datetime(2014,11,1);
Period =1;
```

Create the instrument portfolio with two bonds.

```
InstSet = instbond(CouponRate, Settle, Maturity, ...
Period)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Bond'}
     FieldName: {{11x1 cell}}
    FieldClass: {{11x1 cell}}
```

```
     FieldData: {{11x1 cell}}
```

Create an option on the first bond

```
InstSet = instoptbnd(InstSet, BondIndex, OptSpec, Strike, ExerciseDates, AmericanOpt)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {2x1 cell}
     FieldName: {2x1 cell}
    FieldClass: {2x1 cell}
     FieldData: {2x1 cell}
```

Display the instrument set.

```
instdisp(InstSet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.035      01-Nov-2013    01-Nov-2014    1      0     1            NaN       NaN
2     Bond 0.04       01-Nov-2013    01-Nov-2014    1      0     1            NaN       NaN

Index Type    UnderInd OptSpec Strike ExerciseDates  AmericanOpt
3     OptBond 1        call    85     01-Nov-2014    1
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding Bond option instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`BondIndex` — Number of Bond instruments

vector

Number of Bond instruments, specified as a scalar or an `NINST`-by-`1` vector of indices pointing to underlying instruments of Type `'Bond'` which are stored in `InstSet`. See `instbond` for information on specifying the bond data.

Data Types: `double`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a scalar `'call'` or `'put'` using a character vector or an `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `char` | `cell`

Strike — Option strike price value
matrix of nonnegative integers | vector of nonnegative integers

Option strike price value, specified as a scalar nonnegative integer or an NINST-by-1 vector of strike price values for a European option, an NINST by number of strikes (NSTRIKES) matrix of strike price values for a Bermuda option, or an NINST-by-1 vector of strike price values for each American option. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.

Note The interpretation of the Strike and ExerciseDates depends upon the setting of the AmericanOpt. If AmericanOpt = 0, NaN, or is unspecified, the option is a European or Bermuda option. If AmericanOpt = 1, the option is an American option.

Data Types: double

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a scalar or an NINST-by-2 vector using a datetime array, string array, or date character vectors 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.

To support existing code, instoptbnd also accepts serial date numbers as inputs, but they are not recommended.

AmericanOpt — Option type
0 European (default) | integer with values 0 or 1

(Optional) Option type, specified as a scalar or an NINST-by-1 integer flags with values:

- 0 — European
- 1 — American

Data Types: single | double

Output Arguments

InstSet — Variable containing a collection of instruments
structure

Variable containing a collection of instruments, returned as a structure. 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 InstSet variable, see instget.

FieldList — Name of each data field for Bond option instrument
cell array of character vectors

Name of each data field for a Bond option instrument, returned as an NFIELDS-by-1 cell array of character vectors.

`ClassList` — Data class for each field
cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-1 cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument
character vector

Type of instrument, returned as a character vector. For a Bond option instrument, `TypeString = 'OptBond'`.

More About

Bond Option

A bond option gives the holder 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.

Financial Instruments Toolbox 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.

For more information, see “Bond Options” on page 2-6.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instoptbnd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`instadd` | `instdisp` | `instget` | `hjmprice`

Topics

“Creating Instruments or Properties” on page 1-16
“Supported Interest-Rate Instrument Functions” on page 2-3
“Choose Instruments, Models, and Pricers” on page 1-53

instoptembnd

Construct bond with embedded option

Syntax

```
InstSet = instoptembnd(CouponRate,Settle,Maturity,OptSpec,Strike,
ExerciseDates)
InstSet = instoptembnd(InstSet,CouponRate,Settle,Maturity,OptSpec,Strike,
ExerciseDates)
InstSet = instoptembnd( ___ ,Name,Value)
[FieldList,ClassList,TypeString] = instoptembnd
```

Description

`InstSet = instoptembnd(CouponRate,Settle,Maturity,OptSpec,Strike, ExerciseDates)` creates a new instrument set containing Bond with embedded option instruments.

`InstSet = instoptembnd(InstSet,CouponRate,Settle,Maturity,OptSpec,Strike, ExerciseDates)` adds Bond with embedded option instruments to an existing instrument set.

`InstSet = instoptembnd( ___ ,Name,Value)` uses additional name-value pairs in addition to the required arguments in the previous syntax.

`[FieldList,ClassList,TypeString] = instoptembnd` lists field meta-data for the Bond option instrument.

Examples

Create a Bond With an Embedded Option

This example shows how to create a bond with an embedded option using the following data.

```
Settle = datetime(2007,1,1);
Maturity   = datetime(2010,1,1);
CouponRate = 0.07;
OptSpec = 'call';
Strike= 100;
ExerciseDates= [datetime(2008,1,1) datetime(2010,1,1)];
AmericanOpt=1;
Period = 1;

InstSet = instoptembnd(CouponRate, ...
Settle, Maturity, OptSpec, Strike,  ExerciseDates,'AmericanOpt', AmericanOpt, ...
'Period', Period);

% display the instrument
 instdisp(InstSet)
```

```
Index Type      CouponRate Settle         Maturity       OptSpec Strike ExerciseDates
1     OptEmBond 0.07       01-Jan-2007    01-Jan-2010    call    100    01-Jan-2008   01-Jan-2010
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding Bond embedded option instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as a scalar or an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instoptembnd` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instoptembnd` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a scalar or an `NINST`-by-`1` cell array of character vectors.

Data Types: `char`

`Strike` — Option strike price values

nonnegative integer

Option strike price value, specified as a scalar or an `NINST`-by-`1` or an `NINST`-by-`NSTRIKES` depending on the type of option:

- European option — `NINST`-by-`1` vector of strike price values.

- Bermuda option — NINST by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- American option — NINST-by-1 vector of strike price values for each option.

Data Types: double

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as scalar or an NINST-by-1, NINST-by-2, or NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row.

To support existing code, instoptembnd also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: InstSet = instoptembnd(InstSet,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates, 'Period',1,'AmericanOp',1)

AmericanOpt — Option type
0 European/Bermuda (default) | integer with values 0 or 1

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and a scalar or an NINST-by-1 positive integer flags with values:

- 0 — European/Bermuda
- 1 — American

Data Types: double

Period — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of 'Period' and a scalar or an NINST-by-1 vector.

Data Types: double

Basis — Day-count basis
0 (actual/actual) (default) | integer from 0 to 13

Day-count basis, specified as the comma-separated pair consisting of 'Basis' and a scalar or an NINST-by-1 vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: double

EndMonthRule — End-of-month rule flag
1 (in effect) (default) | nonnegative integer with values 0 or 1

End-of-month rule flag, specified as the comma-separated pair consisting of 'EndMonthRule' and a scalar nonnegative integer or an NINST-by-1 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 coupon payment date is always the same numerical day of the month.
- 1 = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: double

IssueDate — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of 'IssueDate' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, instoptembnd also accepts serial date numbers as inputs, but they are not recommended.

FirstCouponDate — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instoptembnd` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instoptembnd` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `StartDate` is the date when a bond actually starts (that is, the date from which a bond's cash flows can be considered). To make an option embedded bond instrument forward starting, specify this date as a future date.

To support existing code, `instoptembnd` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | `NINST`-by-`1` vector | `NINST`-by-`1` cell array

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a scalar or an `NINST`-by-`1` vector or an `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated face value. The date indicates the last day that the face value is valid.

Note Instruments without a `Face` schedule are treated as either vanilla bonds or stepped coupon bonds with embedded options.

Data Types: `double`

Output Arguments

`InstSet` — Variable containing a collection of instruments

structure

Variable containing a collection of instruments, returned as a structure. 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 `InstSet` variable, see `instget`.

`FieldList` — Name of each data field for Bond embedded option instrument

cell array of character vectors

Name of each data field for a Bond embedded option instrument, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class for each field

cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-`1` cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument

character vector

Type of instrument, returned as a character vector. For a Bond embedded option instrument, `TypeString = 'OptEmBond'`.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2008a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instoptembnd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`instadd` | `instdisp` | `instget`

Topics

“Creating Instruments or Properties” on page 1-16
“Supported Interest-Rate Instrument Functions” on page 2-3
“Choose Instruments, Models, and Pricers” on page 1-53

instoptfloat

Create option instrument on floating-rate note or add instrument to current portfolio

Syntax

```
InstSet = instoptfloat(FloatIndex,OptSpec,Strike,ExerciseDates)
InstSet = instoptfloat(FloatIndex,OptSpec,Strike,ExerciseDates,AmericanOpt)

InstSet = instoptfloat(InstSetOld,___)

[FieldList,ClassList,TypeString] = instoptfloat
```

Description

`InstSet = instoptfloat(FloatIndex,OptSpec,Strike,ExerciseDates)` to specify a European option for a floating-rate note.

`InstSet = instoptfloat(FloatIndex,OptSpec,Strike,ExerciseDates,AmericanOpt)` to specify an American or Bermuda option for a floating-rate note.

`InstSet = instoptfloat(InstSetOld,___)` to add instruments to an existing portfolio.

`[FieldList,ClassList,TypeString] = instoptfloat` lists the field metadata for the `'OptFloat'` instrument.

Examples

Create an Instrument Portfolio with a Call Option for a Floating-Rate Note

Define the floating-rate note:

```
Settle = datetime(2012,11,1);
Maturity   = datetime(2015,11,1);
Spread = 50;
Reset = 1;
```

Create `InstSet`:

```
InstSet = instfloat(Spread, Settle, Maturity, Reset)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Float'}
     FieldName: {{9x1 cell}}
    FieldClass: {{9x1 cell}}
     FieldData: {{9x1 cell}}
```

Display the instrument:

```
instdisp(InstSet)
```

```
Index Type  Spread Settle         Maturity       FloatReset Basis Principal EndMonthRule CapRate
1     Float 50     01-Nov-2012    01-Nov-2015    1          0     100       1            Inf
```

Add a European call option to the instrument portfolio:

```
OptSpec = 'call';
Strike = 100;
ExerciseDates = 'Nov-1-2015';
```

Create `InstSet`:

```
InstSet = instoptfloat(InstSet, 1, OptSpec, Strike, ExerciseDates)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {2x1 cell}
     FieldName: {2x1 cell}
    FieldClass: {2x1 cell}
     FieldData: {2x1 cell}
```

Display the instrument:

```
instdisp(InstSet)
```

```
Index Type  Spread Settle         Maturity       FloatReset Basis Principal EndMonthRule CapRate
1     Float 50     01-Nov-2012    01-Nov-2015    1          0     100       1            Inf

Index Type     UnderInd OptSpec Strike ExerciseDates  AmericanOpt
2     OptFloat 1        call    100    01-Nov-2015    0
```

Input Arguments

`FloatIndex` — Indices pointing to underlying instruments

vector of nonnegative integers

Indices pointing to underlying instruments of `Type 'Float'` specified by a `NINST`-by-`1` vector. The instruments of `Type 'Float'` are also stored in the `InstSet` variable. For more information, see `instfloat`.

Data Types: `double`

`OptSpec` — Definition of option

character vector | cell array of character vectors

Definition of option as `'call'` or `'put'` specified as a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price values for European, Bermuda, or American option

nonnegative integer | vector of nonnegative integers

Option strike price values for option (European, Bermuda, or American) specified as nonnegative integers using as NINST-by-NSTRIKES vector of strike price values.

- For a European or Bermuda option — 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 NaNs.
- For an American Option — NINST-by-1 vector of strike price values for each option.

Data Types: double

ExerciseDates — Exercise date for option (European, Bermuda, or American)
datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American), specified as a NINST-by-NSTRIKES or NINST-by-2 vector using a datetime array, string array, or date character vectors, depending on the option type.

- For a European or Bermuda option — NINST-by-NSTRIKES matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one 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 bond Settle date and the single listed ExerciseDate.

To support existing code, instoptfloat also accepts serial date numbers as inputs, but they are not recommended.

AmericanOpt — Option type
0 if AmericanOpt is NaN or not entered (default) | scalar | vector of positive integers[0,1]

Option type specified as NINST-by-1 positive integer scalar flags with values.

- For a European or Bermuda option — AmericanOpt is 0 for each European or Bermuda option.
- For an American option — AmericanOpt is 1 for each American option. The AmericanOpt argument is required to invoke American exercise rules.

Data Types: single | double

InstSetOld — Variable containing an existing collection of instruments
struct

Variable containing an existing collection of instruments, specified as a struct. For more information on the InstSet variable, see instget.

Data Types: struct

Output Arguments

InstSet — Variable containing a collection of instruments
scalar | vector

Variable containing a collection of instruments returned as a scalar or vector with the instruments broken down by type and each type can have different data fields. Each stored data field has a row

vector or character vector for each instrument. For more information on the `InstSet` variable, see `instget`.

`FieldList` — Data field for instrument type
character vector | cell array of character vectors

Data field for instrument type returned as a `NFIELDS`-by-`1` cell array of character vectors listing the name of each data field for this instrument type.

`ClassList` — Data class of each field
character vector with value: `'dble'`, `'date'`, `'char'` | cell array of character vectors with values: `'dble'`, `'date'`, `'char'`

Data class of each field returned as a`NFIELDS`-by-`1` cell array of character vectors listing the data class of each field. The class determines how arguments are parsed.

`TypeString` — Type of instrument added
character vector with value `'OptFloat'`

Type of instrument added returned as a character vector. The character vector for a floating-rate option instrument is `TypeString = 'OptFloat'`.

Version History

Introduced in R2013a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `instoptfloat` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`instadd` | `instoptemfloat`

Topics
“Creating Instruments or Properties” on page 1-16
“Supported Interest-Rate Instrument Functions” on page 2-3
“Choose Instruments, Models, and Pricers” on page 1-53

instoptemfloat

Create embedded option instrument on floating-rate note or add instrument to current portfolio

Syntax

```
InstSet = instopemtfloat(Spread,Settle,Maturity,OptSpec,Strike,ExerciseDates)
InstSet = instopemtfloat( ___ ,Name,Value)

InstSet = instopemtfloat(InstSetOld,Spread,Settle,Maturity,OptSpec,Strike,
ExerciseDates)
[FieldList,ClassList,TypeString] = instoptemfloat
```

Description

`InstSet = instopemtfloat(Spread,Settle,Maturity,OptSpec,Strike,ExerciseDates)` creates an embedded option instrument for a floating-rate note.

`InstSet = instopemtfloat( ___ ,Name,Value)` adds optional name-value pair arguments.

`InstSet = instopemtfloat(InstSetOld,Spread,Settle,Maturity,OptSpec,Strike,ExerciseDates)` to add `'OptEmFloat'` instruments to an instrument variable.

`[FieldList,ClassList,TypeString] = instoptemfloat` lists field metadata for the `'OptEmFloat'` instrument.

Examples

Create an Instrument Portfolio with a Embedded Option Floating-Rate Note

Define the embedded call option:

```
Settle = 'Nov-1-2012';
Maturity   = 'Nov-1-2015';
Spread = 25;
OptSpec = 'call';
Strike= 100;
ExerciseDates = 'Nov-1-2015';
Reset = 1;
```

Create `InstSet`:

```
InstSet = instoptemfloat(Spread, Settle, Maturity, OptSpec,...
Strike,  ExerciseDates,'FloatReset', Reset)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'OptEmFloat'}
     FieldName: {{13x1 cell}}
    FieldClass: {{13x1 cell}}
```

```
    FieldData: {{13x1 cell}}
```

Display the instrument:

```
instdisp(InstSet)
```

```
Index Type       Spread Settle         Maturity       OptSpec Strike ExerciseDates  FloatReset Ba
1     OptEmFloat 25     01-Nov-2012    01-Nov-2015    call    100    01-Nov-2015    1          0
```

Input Arguments

Spread — Number of basis points over the reference rate

nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (NINST)-by-1).

Data Types: double

Settle — Settlement dates of floating-rate note

ValuationDate of HW Tree (default) | datetime array | string array | date character vector | serial date number

Settlement dates of floating-rate note, specified as a NINST-by-1 vector using a datetime array, string array, date character vectors, or serial date numbers.

Data Types: double | char | string | datetime

Maturity — Floating-rate note maturity date

datetime array | string array | date character vector | serial date number

Floating-rate note maturity date, specified as a NINST-by-1 vector using a datetime array, string array, date character vectors, or serial date numbers.

Data Types: double | char | string | datetime

OptSpec — Definition of option

character vector | cell array of character vectors

Definition of option as 'call' or 'put' specified as a NINST-by-1 cell array of character vectors for 'call' or 'put'.

Data Types: char | cell

Strike — Embedded option strike price values

nonnegative integer | vector of nonnegative integers

Embedded option strike price values for option specified as nonnegative integers using as NINST-by-NSTRIKES or NINST-by-1 vector of strike price values, depending on the type of option.

- For a European or Bermuda Option — NINST-by-NSTRIKES matrix of strike price values where each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.

- For an American Option — NINST-by-1 vector of strike price values for each option.

Data Types: double

ExerciseDates — Exercise date for embedded option
datetime array | string array | date character vector | serial date number

Exercise date for embedded option, specified as a NINST-by-NSTRIKES or NINST-by-2 vector using a datetime array, string array, date character vectors, or serial date numbers, depending on the type of option.

- For a European or Bermuda Option — NINST-by-NSTRIKES of exercise dates where 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 bond Settle date and the single listed ExerciseDate.

Data Types: double | char | string | datetime

InstSetOld — Variable containing an existing collection of instruments
struct

Variable containing an existing collection of instruments, specified as a struct. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument. For more information on instrument data parameters, see the reference entries for individual instrument types. For example, see instfloat for additional information on the float instrument.

Data Types: struct

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: InstSet = instoptemfloat(Spread,Settle,Maturity,OptSpec,Strike,ExerciseDates,'FloatReset',Reset)

AmericanOpt — Embedded option type
0 if AmericanOpt is NaN or not entered (default) | scalar | vector of positive integers[0,1]

Embedded option type, specified as the comma-separated pair consisting of 'AmericanOpt' and a NINST-by-1 positive integer scalar flags with values:

- For a European or Bermuda option — AmericanOpt is 0 for each European or Bermuda option. The default is 0 if AmericanOpt is NaN or not entered.
- For an American option — AmericanOpt is 1 for each American option. The AmericanOpt argument is required to invoke American exercise rules.

Data Types: single | double

`FloatReset` — Frequency of payments per year
`1` (default) | positive integer from the set`[1,2,3,4,6, 12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of payments per year, specified as the comma-separated pair consisting of `'FloatReset'` and positive integers for the values `1,2,4,6,12]` in a `NINST`-by-`1` vector.

Data Types: `single` | `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `NINST`-by-`1` vector. The `Basis` value represents the basis used when annualizing the input forward-rate tree.

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

For more information, see “Basis” on page 2-228.

Data Types: `single` | `double`

`Principal` — Principal values
`100` (default) | nonnegative integer | vector of nonnegative integers | cell array of nonnegative integers

Principal values, specified as the comma-separated pair consisting of `'Principal'` and a nonnegative integer using a `NINST`-by-`1` vector of notional principal amounts.

Data Types: `single` | `double`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure using `derivset`.

Data Types: `struct`

EndMonthRule — End-of-month rule flag
1 (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `single` | `double`

Output Arguments

InstSet — Variable containing a collection of instruments
scalar | vector

Variable containing a collection of instruments returned as a scalar or vector with the instruments broken down by type and each type can have different data fields. Each stored data field has a row vector or character vector for each instrument. For more information on the `InstSet` variable, see `instget`.

FieldList — Name of each data field
character vector | cell array of character vectors

`NFIELDS`-by-`1` cell array of character vectors listing the name of each data field for this instrument type.

ClassList — Determines how arguments are parsed
character vector with value: `'dble'`, `'date'`, or `'char'` | cell array of character vectors with values: `'dble'`, `'date'`, or `'char'`

`NFIELDS`-by-`1` cell array of character vectors listing the data class of each field.

TypeString — Type of instrument added
character vector with value `'OptEmFloat'`

Character vector specifying the type of instrument added where `TypeString = 'OptEmFloat'`.

Version History
Introduced in R2013a

See Also
`instadd` | `instoptfloat`

Topics
“Creating Instruments or Properties” on page 1-16
“Basis” on page 2-228
“Supported Interest-Rate Instrument Functions” on page 2-3

instoptstock

Construct stock option

Syntax

```
InstSet = instoptstock(OptSpec,Strike,Settle,ExerciseDates)
InstSet = instoptstock(InstSet,OptSpec,Strike,Settle,ExerciseDates)
InstSet = instoptstock( ___ ,AmericanOpt)
[FieldList,ClassList,TypeString] = instoptstock
```

Description

`InstSet = instoptstock(OptSpec,Strike,Settle,ExerciseDates)` creates a new instrument set containing stock option instruments.

`InstSet = instoptstock(InstSet,OptSpec,Strike,Settle,ExerciseDates)` adds stock option instruments to an existing instrument set.

`InstSet = instoptstock( ___ ,AmericanOpt)` adds an optional argument for `AmericanOpt`.

`[FieldList,ClassList,TypeString] = instoptstock` lists field meta-data for the stock option instrument.

Examples

Create a Stock Option Instrument

Create an instrument set of two stock options with the following data:

```
OptSpec = {'put';'call'};
Strike = [95;98];
Settle = datetime(2012,5,1);
ExerciseDates = [datetime(2014,5,1) ; datetime(2015,5,1)];
AmericanOpt = [0;1];
```

Create the stock option instruments.

```
InstSet = instoptstock(OptSpec, Strike,Settle, ExerciseDates, AmericanOpt)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'OptStock'}
     FieldName: {{5x1 cell}}
    FieldClass: {{5x1 cell}}
     FieldData: {{5x1 cell}}
```

Display the instrument set.

```
instdisp(InstSet)
```

```
Index Type     OptSpec Strike Settle         ExerciseDates  AmericanOpt
1     OptStock put     95     01-May-2012    01-May-2014    0
2     OptStock call    98     01-May-2012    01-May-2015    1
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding stock option instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a scalar `'call'` or `'put'` or an `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values

nonnegative integer

Option strike price value, specified with a scalar or an `NINST`-by-`1` or an `NINST`-by-`NSTRIKES` depending on the option type:

- For a European option, use an `NINST`-by-`1` vector of strike prices.
- For a Bermuda option, use an `NINST`-by-`NSTRIKES` matrix of strike prices. 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, use an `NINST`-by-`1` of strike prices.

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.

Data Types: `double`

`Settle` — Settlement date or trade date

datetime array | string array | date character vector

Settlement date or trade date, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instoptstock` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a scalar or an NINST-by-1,NINST-by-2, or an NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the option type:

- For a European option, use an NINST-by-1 vector of dates. Each row is the schedule for one option. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use an NINST-by-NSTRIKES vector of dates. Each row is the schedule for one option.
- For an American option, use an NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row.

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.

To support existing code, instoptstock also accepts serial date numbers as inputs, but they are not recommended.

AmericanOpt — Option type

0 European or Bermuda (default) | integer with values of 0 or 1

(Optional) Option type, specified as a scalar or an NINST-by-1 vector of integer flags with values:

- 0 — European or Bermuda
- 1 — American

Data Types: double

Output Arguments

InstSet — Variable containing a collection of instruments

structure

Variable containing a collection of instruments, returned as a structure. 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 InstSet variable, see instget.

FieldList — Name of each data field for stock option instrument

cell array of character vectors

Name of each data field for a stock option instrument, returned as an NFIELDS-by-1 cell array of character vectors.

ClassList — Data class for each field

cell array of character vectors

Data class for each field, returned as an NFIELDS-by-1 cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are 'dble', 'date', and 'char'.

TypeString — Type of instrument

character vector

Type of instrument, returned as a character vector. For a stock option instrument, `TypeString = 'OptStock'`.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instoptstock` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`instadd` | `instdisp` | `instget`

Topics

“Pricing Equity Derivatives Using Trees” on page 3-64
“Creating Instruments or Properties” on page 1-16
“Supported Equity Derivative Functions” on page 3-19

“Choose Instruments, Models, and Pricers” on page 1-53

instrangefloat

Construct range note instrument

Syntax

```
InstSet = instrangefloat(Spread,Settle,Maturity,RateSched,Reset,Basis,
Principal,EndMonthRule)
InstSet = instrangefloat( ___ ,Reset,Basis,Principal,EndMonthRule)
InstSet = instrangefloat(InstSet, ___ )
```

Description

`InstSet = instrangefloat(Spread,Settle,Maturity,RateSched,Reset,Basis, Principal,EndMonthRule)` creates a range instrument from data arrays.

`InstSet = instrangefloat( ___ ,Reset,Basis,Principal,EndMonthRule)` creates a range instrument from data arrays using optional arguments.

`InstSet = instrangefloat(InstSet, ___ )` adds new range set instrument to an existing instrument set.

Examples

Create a Range Note Instrument

Create an instrument portfolio with a range note.

```
Spread = 100;
Settle = datetime(2011,1,1);
Maturity = datetime(2014,1,1);

RateSched.Dates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
RateSched.Rates  = [0.045 0.055; 0.0525  0.0675; 0.06 0.08];

% Create InstSet
InstSet = instrangefloat(Spread, Settle, Maturity, RateSched);

% Display the portfolio instrument
instdisp(InstSet)
```

```
Index Type       Spread Settle         Maturity       RateSched FloatReset Basis Principal EndMon
1     RangeFloat 100    01-Jan-2011    01-Jan-2014    [Struct]  1          0     100       1
```

Add a second range note instrument to the portfolio. Second Range Note:

```
Spread2 = 200;
Settle2 = datetime(2011,1,1);
Maturity2 = datetime(2013,1,1);
RateSched2.Dates = [datetime(2012,1,1) ; datetime(2013,1,1)];
RateSched2.Rates  = [0.048 0.059; 0.055  0.068];
```

```
InstSet = instrangefloat(InstSet, Spread2, Settle2, Maturity2, RateSched2);

% Display the portfolio instrument
instdisp(InstSet)

Index Type       Spread Settle         Maturity       RateSched FloatReset Basis Principal EndMo
1     RangeFloat 100    01-Jan-2011    01-Jan-2014    [Struct]  1          0     100       1
2     RangeFloat 200    01-Jan-2011    01-Jan-2013    [Struct]  1          0     100       1
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding stock option instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`Spread` — Number of basis points over the reference rate

numeric

Number of basis points over the reference rate, specified as a scalar numeric.

Data Types: `double`

`Settle` — Settlement date of floating-rate note

datetime array | string array | date character vector

Settlement date of floating-rate note, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instrangefloat` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date of floating-rate note

datetime array | string array | date character vector

Maturity date of floating-rate note, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instrangefloat` also accepts serial date numbers as inputs, but they are not recommended.

`RateSched` — Range of dates within which cash flows are nonzero

date character vector

Range of dates within which cash flows are nonzero, specified as an `NINST`-by-`1` structure where each element of the structure array contains two fields:

- `RateSched.Dates` — `NDates`-by-`1` cell array of dates corresponding to the range schedule.
- `RateSched.Rates` —`NDates`-by-`2` array with the first column containing the lower bound of the range and the second column containing the upper bound of the range. Cash flow for date

`RateSched.Dates`(*n*) is nonzero for rates in the range `RateSched.Rates`(*n*,1) < `Rate` < `RateSched.Rate` (*n*,2).

To support existing code, `instrangefloat` also accepts serial date numbers as inputs, but they are not recommended.

`Reset` — Frequency of payments per year
`1` (default) | integer

(Optional) Frequency of payments per year, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day count basis, specified as an `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amounts, specified and an `NINST`-by-`1` vector.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

(Optional) End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as an `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`InstSet` — Variable containing a collection of instruments

structure

Variable containing a collection of instruments, returned as a structure. Instruments are divided by type and each type can have different data fields. Each stored data field has a row vector or character vector for each instrument. Values are:

- `FieldList` — `NFIELDS`-by-1 cell array of character vectors listing the name of each data field for this instrument type.
- `ClassList` — `NFIELDS`-by-1 cell array of character vectors listing the data class of each field. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.
- `TypeString` — Character vector specifying the type of instrument added. `TypeString = 'RangeFloat'`.

For more information on the `InstSet` variable, see `instget`.

More About

Range Note Instrument

A range note is a structured (market-linked) security whose coupon rate is equal to the reference rate as long as the reference rate is within a certain range.

If the reference rate is outside of the range, the coupon rate is 0 for that period. This type of instrument entitles the holder to cash flows that depend on the level of some reference interest rate and are floored to be positive. The note holder gets direct exposure to the reference rate. In return for the drawback that no interest is paid for the time the range is left, they offer higher coupon rates than comparable standard products, like vanilla floating notes.

Version History

Introduced in R2012a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instrangefloat` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Jarrow, Robert. “Modelling Fixed Income Securities and Interest Rate Options.” *Stanford Economics and Finance*. 2nd Edition. 2002.

See Also

`instbond` | `instcap` | `instswap` | `instaddfield` | `instdisp` | `intenvprice` | `rangefloatbybk` | `rangefloatbybdt` | `rangefloatbyhw` | `rangefloatbyhjm`

Topics

“Creating Instruments or Properties” on page 1-16
“Supported Interest-Rate Instrument Functions” on page 2-3

instselect

Create instrument subset by matching conditions

Syntax

```
InstSubSet = instselect(InstSet,Name,Value)
```

Description

`InstSubSet = instselect(InstSet,Name,Value)` returns a variable containing a collection of instruments matching the input criteria.

Examples

Create Instrument Subset

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
```

Make a new portfolio containing only options struck at 95.

```
Opt95 = instselect(ExampleInst, 'FieldName', 'Strike','Data', '95')
```

```
Opt95 = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {3x1 cell}
     FieldName: {3x1 cell}
    FieldClass: {3x1 cell}
     FieldData: {3x1 cell}
```

```
instdisp(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'})
```

```
FutTBill = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {3x1 cell}
     FieldName: {3x1 cell}
    FieldClass: {3x1 cell}
     FieldData: {3x1 cell}
```

```
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
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of instruments, specified as `InstSet` structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Opt95 = instselect(ExampleInst,'FieldName','Strike','Data','95')`

`FieldName` — Name of each data field for instrument

cell array of character vectors

Name of each data field for an instrument, specified as the comma-separated pair consisting of `'FieldName'` and an `NFIELDS`-by-1 cell array of character vectors.

Data Types: `char` | `cell`

`Data` — Data values for field
cell array of character vectors

Data values for field, specified as the comma-separated pair consisting of `'Data'` and an `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 `FieldName`. The number of columns is arbitrary and matching ignores trailing `NaN`s or spaces.

Data Types: `double` | `cell`

`Index` — Number of instruments
all types in the instrument variable (default) | vector

Number of instruments, specified as the comma-separated pair consisting of `'Index'` and an `NINST`-by-`1` vector restricting positions of instruments to check for matches. The default is all indices available in the instrument variable.

Data Types: `double`

`Type` — Number of types
all types in the instrument variable (default) | character vector

Number of types, specified as the comma-separated pair consisting of `'Type'` and a `NTYPES`-by-`1` cell array of character vectors restricting instruments to match one of `Type` types. The default is all types in the instrument variable.

Data Types: `char` | `cell`

Output Arguments

`InstSubSet` — Variable containing a collection of instruments matching input criteria
structure

Variable containing a collection of instruments matching the input criteria, returned as a structure. Instruments are returned in `InstSubSet` if all the `FieldName`, `Index`, and `Type` conditions are met. An instrument meets an individual `FieldName` condition if the value matches any of the rows listed in the `Data` for that `FieldName`. See `instfind` for examples on matching criteria.

Version History

Introduced before R2006a

See Also

`instaddfield` | `instdelete` | `instfind` | `instget` | `instgetcell`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15

instsetfield

Add or reset data for existing instruments

Syntax

```
InstSet = instsetfield(InstSet,Name,Value)
```

Description

`InstSet = instsetfield(InstSet,Name,Value)` resets or adds fields to every instrument.

Examples

Add or Reset Data for Existing Instruments

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 field `Maturity` for the cash instrument.

```
MDate = datetime(1999,7,1);
ISet = instsetfield(ISet, 'Type', 'TBill', 'FieldName',...
'Maturity','FieldClass', 'date', 'Data', MDate);
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 Maturity
7     TBill 99    01-Jul-1999
```

Create a 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
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of instruments, specified as `InstSet` structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ISet = instsetfield(ISet,'Index',6,'FieldName',{'Strike','Price'},'Data',{ 95 , 2.9 })`

`FieldName` — Name of each data field for instrument

cell array of character vectors

Name of each data field for an instrument, specified as the comma-separated pair consisting of `'FieldName'` and an `NFIELDS`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Data` — Data values for field

cell array of character vectors

Data values for field, specified as the comma-separated pair consisting of `'Data'` and an `NVALUES`-by-`M` array or `NFIELDS`-by-`1` cell array of acceptable data values 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.

Data Types: `double` | `cell`

`Index` — Number of instruments
all types in the instrument variable (default) | vector

Number of instruments, specified as the comma-separated pair consisting of `'Index'` and an `NINST`-by-`1` vector of positions of instruments to work on. If `Type` is also entered, instruments referenced must be one of `Type` types contained in `Index`.

Data Types: `double`

`Type` — Number of types
all types in the instrument variable (default) | character vector

Number of types, specified as the comma-separated pair consisting of `'Type'` and a `NTYPES`-by-`1` cell array of character vectors restricting instruments worked on to match one of `Type` types.

Data Types: `char` | `cell`

Output Arguments

`InstSet` — Instrument set variable containing the input data
structure

Instrument set variable containing the input data, returned as a structure.

Version History

Introduced before R2006a

See Also

`instaddfield` | `instdisp` | `instget` | `instgetcell`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15

instswap

Construct swap instrument

Syntax

```
InstSet = instswap(LegRate,Settle,Maturity)
InstSet = instswap(InstSet,LegRate,Settle,Maturity)
InstSet = instswap( ___ ,LegReset,Basis,Principal,LegType,EndMonthRule,
StartDate)
[FieldList,ClassList,TypeString] = instswap
```

Description

`InstSet = instswap(LegRate,Settle,Maturity)` creates a new instrument set containing Swap instruments.

`InstSet = instswap(InstSet,LegRate,Settle,Maturity)` adds Swap instruments to an existing instrument set.

`InstSet = instswap(___ ,LegReset,Basis,Principal,LegType,EndMonthRule,
StartDate)` adds optional arguments for `LegReset`, `Basis`, `Principal`, `LegType`, `EndMonthRule`, and `StartDate`.

`[FieldList,ClassList,TypeString] = instswap` lists field meta-data for the Swap instrument.

Examples

Create a Vanilla Swap Instrument

Create a vanilla swap using market data.

Use the following market data to create a swap instrument.

```
LegRate = [0.065, 0]
```

```
LegRate = 1×2

    0.0650         0
```

```
Settle = datetime(2007,1,1);
Maturity = datetime(2012,1,1);
LegReset = [1, 1];
Basis = 0
```

```
Basis = 0
```

```
Principal = 100
```

```
Principal = 100
```

```
LegType = [1, 0]
```

```
LegType = 1×2

     1     0
```

```
InstSet = instswap(LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)
```

```
InstSet = struct with fields:
        FinObj: 'Instruments'
    IndexTable: [1x1 struct]
          Type: {'Swap'}
     FieldName: {{9x1 cell}}
    FieldClass: {{9x1 cell}}
     FieldData: {{9x1 cell}}
```

View the swap instrument using `instdisp`.

```
instdisp(InstSet)
```

```
Index Type LegRate    Settle         Maturity       LegReset Basis Principal LegType EndMonthRule
1     Swap [0.065  0] 01-Jan-2007    01-Jan-2012    [1  1]   0     100       [1  0]  1
```

Create a Float-Float Swap and Price with `intenvprice`

Use `instswap` to create a float-float swap and price the swap with `intenvprice`.

```
RateSpec = intenvset('Rates',.05,'StartDate',today,'EndDate',datemnth(today,60));
IS = instswap([40 20],today,datemnth(today,60),[], [], [], [0 0]);
intenvprice(RateSpec,IS)
```

```
ans = 0.8644
```

Create Float-Float, Fixed-Fixed, and Float-Fixed Swaps and Price with `intenvprice`

Use `instswap` to create swaps and price the swaps with `intenvprice`.

```
RateSpec = intenvset('Rates',.05,'StartDate',today,'EndDate',datemnth(today,60));
IS = instswap([.03 .02],today,datemnth(today,60),[], [], [], [1 1]);
IS = instswap(IS,[200 300],today,datemnth(today,60),[], [], [], [0 0]);
IS = instswap(IS,[300 .07],today,datemnth(today,60),[], [], [], [0 1]);
intenvprice(RateSpec,IS)
```

```
ans = 3×1

    4.3220
   -4.3220
    4.5921
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable, specified only when adding Swap instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

`LegRate` — Leg rate

matrix

Leg rate, specified as a scalar or an `NINST`-by-2 matrix, with each row defined as one of the following:

- `[CouponRate Spread]` (fixed-float)
- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

`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.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `instswap` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a scalar or an `NINST`-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each swap.

To support existing code, `instswap` also accepts serial date numbers as inputs, but they are not recommended.

`LegReset` — Reset frequency per year for each swap

`[1 1]` (default) | vector

(Optional) Reset frequency per year for each swap, specified as an `NINST`-by-2 vector.

Data Types: `double`

`Basis` — Day-count basis representing the basis for each leg

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis representing the basis for each leg, specified as an `NINST`-by-1 array (or `NINST`-by-2 if `Basis` is different for each leg).

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

(Optional) Notional principal amounts or principal value schedules, specified as a vector or cell array.

`Principal` accepts an `NINST`-by-`1` vector or an `NINST`-by-`1` cell array (or `NINST`-by-`2` if `Principal` is different for each leg) of the notional principal amounts or principal value schedules. For schedules, each element of the cell array is a `NumDates`-by-`2` array where the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`LegType` — Leg type
`[1 0]` for each instrument (default) | matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float)

(Optional) Leg type, specified as an `NINST`-by-`2` matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float). 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`. `LegType` allows `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float) swaps

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

(Optional) End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as a nonnegative integer `0` or `1`using an `NINST`-by-`1` (or `NINST`-by-`2` if `EndMonthRule` is different for each leg).

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`StartDate` — Date swap actually starts
`Settle` date (default) | datetime array | string array | date character vector

(Optional) Date swap actually starts, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `instswap` also accepts serial date numbers as inputs, but they are not recommended.

Use this argument to price forward swaps, that is, swaps that start in a future date

Output Arguments

`InstSet` — Variable containing a collection of instruments
structure

Variable containing a collection of instruments, returned as a structure. 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 `InstSet` variable, see `instget`.

`FieldList` — Name of each data field for Swap instrument
cell array of character vectors

Name of each data field for a Swap instrument, returned as an `NFIELDS`-by-`1` cell array of character vectors.

`ClassList` — Data class for each field
cell array of character vectors

Data class for each field, returned as an `NFIELDS`-by-`1` cell array of character vectors. The class determines how arguments are parsed. Valid character vectors are `'dble'`, `'date'`, and `'char'`.

`TypeString` — Type of instrument
character vector

Type of instrument, returned as a character vector. For a Swap instrument, `TypeString = 'Swap'`.

More About

Amortizing Swap

In an amortizing swap, the notional principal decreases periodically because it is tied to an underlying financial instrument with a declining (amortizing) principal balance, such as a mortgage.

Forward Swap

Agreement to enter into an interest-rate swap arrangement on a fixed date in future.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `instswap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmprice` | `instaddfield` | `instbond` | `instcap` | `instdisp` | `instfloor` | `intenvprice`

Topics

“Creating Instruments or Properties” on page 1-16
“Supported Interest-Rate Instrument Functions” on page 2-3
“Choose Instruments, Models, and Pricers” on page 1-53

instswaption

Construct swaption instrument

Syntax

```
InstSet = instswaption(OptSpec,Strike,ExerciseDates,Spread,Settle,Maturity)
InstSet = instswaption( ___ ,AmericanOpt,SwapReset,Basis,Principal)
InstSet = instswaption(InstSetOld, ___ )
[FieldList,ClassList,TypeString] = instswaption
```

Description

`InstSet = instswaption(OptSpec,Strike,ExerciseDates,Spread,Settle,Maturity)` to specify a European swaption.

Fill in unspecified entries 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 `[]`.

`InstSet = instswaption( ___ ,AmericanOpt,SwapReset,Basis,Principal)` to specify an American swaption.

`InstSet = instswaption(InstSetOld, ___ )` to add swaption instruments to an instrument variable.

`[FieldList,ClassList,TypeString] = instswaption` to list field metadata for the swaption instrument.

Examples

Create Two Swaption Instruments

This example shows how to create two European swaption instruments using the following data.

```
OptSpec = {'Call'; 'Put'};
Strike = .05;
ExerciseDates = datetime(2011,1,1);
Spread=0;
Settle = datetime(2007,1,1);
Maturity = datetime(2012,1,1);
AmericanOpt = 0;

InstSet = instswaption(OptSpec, Strike, ExerciseDates, Spread, Settle, Maturity, ...
 AmericanOpt);

% view the European swaption instruments using instdisp
instdisp(InstSet)
```

```
Index Type     OptSpec Strike ExerciseDates  Spread Settle         Maturity       AmericanOpt Swa
1     Swaption Call    0.05   01-Jan-2011    0      01-Jan-2007    01-Jan-2012    0           1
2     Swaption Put     0.05   01-Jan-2011    0      01-Jan-2007    01-Jan-2012    0           1
```

Create Two European Swaption Instruments with Receiving and Paying Legs

This example shows how to create two European swaption instruments with receiving and paying legs using the following data.

```
OptSpec = {'Call'; 'Put'};
Strike = .05;
ExerciseDates = datetime(2011,1,1);
Spread=0;
Settle = datetime(2007,1,1);
Maturity = datetime(2012,1,1);
AmericanOpt = 0;
SwapReset = [2 4]; % 1st column represents receiving leg, 2nd column represents paying leg
Basis = [1 3];     % 1st column represents receiving leg, 2nd column represents paying leg

InstSet = instswaption(OptSpec,Strike,ExerciseDates,AmericanOpt,Spread,Settle,Maturity, ...
SwapReset,Basis);
```

View the European swaption instruments using `instdisp`.

```
instdisp(InstSet)
```

```
Index Type     OptSpec Strike ExerciseDates  Spread Settle Maturity       AmericanOpt SwapReset E
1     Swaption Call    0.05   01-Jan-2011    0      0      01-Jan-2007    NaN         2  4      1
2     Swaption Put     0.05   01-Jan-2011    0      0      01-Jan-2007    NaN         2  4      1
```

Input Arguments

`OptSpec` — Definition of option

cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors with values `'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.

Data Types: `char` | `cell`

`Strike` — Strike swap rate values

vector

Strike swap rate values, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a vector using a datetime array, string array, or date character vectors, where each row is the schedule for one option and the last element of each row must be the same as the maturity of the tree.

- For a European option, use a `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, use a NINST-by-2 vector of exercise dates. 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.

To support existing code, instswaption also accepts serial date numbers as inputs, but they are not recommended.

Spread — Number of basis points over the reference rate
nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate, specified as a vector of nonnegative integers for the number of instruments (NINST)-by-1).

Data Types: double

Settle — Settle date for each swap
datetime array | string array | date character vector

Settle date for each swap, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, instswaption also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date for each swap
datetime array | string array | date character vector

Maturity date for each swap, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, instswaption also accepts serial date numbers as inputs, but they are not recommended.

AmericanOpt — Option type
0 European (default) | integer with values 0 or 1

(Optional) Option type, specified as NINST-by-1 integer flags with values:

- 0 — European
- 1 — American

The AmericanOpt argument is required to invoke American exercise rules.

Data Types: double

SwapReset — Reset frequency per year for each leg
1 (default) | numeric

(Optional) Reset frequency per year for each leg, specified as a NINST-by-1 vector or NINST-by-2 matrix. If SwapReset is NINST-by-2, the first column represents the receiving leg, while the second column represents the paying leg.

Data Types: double

Basis — Day-count basis of instrument
0 (actual/actual) (default) | integer from 0 to 13

(Optional) Day-count basis of the instrument, specified as a `NINST`-by-`1` vector or `NINST`-by-`2` matrix representing the basis for each leg. If `Basis` is `NINST`-by-`2`, the first column represents the receiving leg, while the second column represents the paying leg.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

(Optional) Notional principal amount, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`InstSetOld` — Variable containing an existing collection of instruments
struct

Variable containing an existing collection of instruments, specified as a struct. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument. The `InstSetOld` argument is specified only when adding swaption instruments to an existing instrument set. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

Output Arguments

`InstSet` — Variable containing collection of instruments
vector

(Optional) 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 character vector for each instrument. For more information on the `InstSet` variable, see `instget`.

Data Types: `double`

FieldList — Name of each data field for this instrument type
cell array of character vectors

Name of each data field for this instrument type, returned as a NFIELDS-by-1 cell array of character vectors.

Data Types: char | cell

ClassList — Data class of each field
character vector with value: 'dble', 'date', or 'char'.

Data class of each field, returned as a NFIELDS-by-1 cell array of character vectors. Valid character vectors are 'dble', 'date', and 'char'.

Data Types: char | cell

TypeString — Type of instrument added
character vector

Type of instrument added, returned as a character vector (for a swaption, TypeString = 'Swaption').

Data Types: char

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although instswaption supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

instadd | instget | instdisp

Topics
“Creating Instruments or Properties” on page 1-16
“Supported Interest-Rate Instrument Functions” on page 2-3
“Choose Instruments, Models, and Pricers” on page 1-53

insttypes

List types

Syntax

```
TypeList = insttypes(InstSet)
```

Description

`TypeList = insttypes(InstSet)` retrieves a list of types stored in an instrument variable.

Examples

List Instrument Types

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 = 3x1 cell
    {'Futures'}
    {'Option' }
    {'TBill'  }
```

Input Arguments

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of instruments, specified as `InstSet` structure. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument. For more information on the `InstSet` variable, see `instget`.

Data Types: `struct`

Output Arguments

`TypeList` — Types of instrument

cell array of character vectors

Types of instruments, returned as an `NTYPES`-by-`1` cell array of character vectors listing the `Type` of instruments contained in the `InstSet` variable.

Version History

Introduced before R2006a

See Also

`instdisp` | `instfields` | `instlength`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15

intenvget

Properties of interest-rate structure

Syntax

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

Description

`ParameterValue = intenvget(RateSpec,ParameterName)` obtains the value of the named parameter `ParameterName` extracted from the `RateSpec`.

Examples

Extract Values from a RateSpec

Use `intenvset` to set the interest-rate structure.

```
RateSpec = intenvset('Rates',0.05,'StartDates',...
'20-Jan-2000','EndDates','20-Jan-2001')
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: 0.9518
            Rates: 0.0500
         EndTimes: 2
       StartTimes: 0
         EndDates: 730871
       StartDates: 730505
    ValuationDate: 730505
            Basis: 0
     EndMonthRule: 1
```

Use `intenvget` to extract values from the `RateSpec`.

```
[R, RateSpec] = intenvget(RateSpec,'Rates')
```

```
R = 0.0500
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: 0.9518
            Rates: 0.0500
         EndTimes: 2
       StartTimes: 0
         EndDates: 730871
       StartDates: 730505
    ValuationDate: 730505
            Basis: 0
```

```
    EndMonthRule: 1
```

Input Arguments

RateSpec — Interest-rate specification

structure

Interest-rate specification, specified by the `RateSpec` obtained previously from `intenvset` or `toRateSpec` for an `IRDataCurve` or `toRateSpec` for an `IRFunctionCurve`.

Data Types: `struct`

ParameterName — Parameter name to be accessed

character vector

Parameter name to be accessed, specified as a character vector. 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.

Data Types: `char`

Output Arguments

ParameterValue — Value of the named parameter ParameterName extracted from RateSpec

scalar

Value of the named parameter `'ParameterName'` extracted from `RateSpec`, returned as a scalar value.

Version History

Introduced before R2006a

See Also

`intenvset`

Topics

“Modeling the Interest-Rate Term Structure” on page 2-57
“Understanding the Interest-Rate Term Structure” on page 2-48

intenvprice

Price instruments from set of zero curves

Syntax

```
Price = intenvprice(RateSpecInstSet)
```

Description

`Price = intenvprice(RateSpecInstSet)` computes arbitrage-free prices for instruments against a set of zero coupon bond rate curves.

`intenvprice` handles the following instrument types: `'Bond'`, `'CashFlow'`, `'Fixed'`, `'Float'`, `'Swap'`. See `instadd` for information about constructing defined types.

Examples

Load Zero Curves and Instruments from Data File

Load the zero curves and instruments.

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

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.04       01-Jan-2000    01-Jan-2003    1      NaN   NaN          NaN       NaN
2     Bond 0.04       01-Jan-2000    01-Jan-2004    2      NaN   NaN          NaN       NaN

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

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

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

Price the instruments.

```
Price = intenvprice(ZeroRateSpec, ZeroInstSet)
```

```
Price = 5×1

   98.7159
   97.5334
   98.7159
  100.5529
    3.6923
```

Create a Float-Float Swap and Price with intenvprice

Use instswap to create a float-float swap and price the swap with intenvprice.

```
RateSpec = intenvset('Rates',.05,'StartDate',today,'EndDate',datemnth(today,60));
IS = instswap([400 200],today,datemnth(today,60),[], [], [], [0 0]);
intenvprice(RateSpec,IS)
```

```
ans = 8.6440
```

Create Float-Float, Fixed-Fixed, and Float-Fixed Swaps and Price with intenvprice

Use instswap to create swaps and price the swaps with intenvprice.

```
RateSpec = intenvset('Rates',.05,'StartDate',today,'EndDate',datemnth(today,60));
IS = instswap([.03 .02],today,datemnth(today,60),[], [], [], [1 1]);
IS = instswap(IS,[200 300],today,datemnth(today,60),[], [], [], [0 0]);
IS = instswap(IS,[300 .07],today,datemnth(today,60),[], [], [], [0 1]);
intenvprice(RateSpec,IS)
```

```
ans = 3×1

    4.3220
   -4.3220
    4.5921
```

Input Arguments

RateSpec — Interest-rate specification

structure

(Optional) Interest-rate specification, specified by the RateSpec obtained previously from intenvset or toRateSpec for an IRDataCurve or toRateSpec for an IRFunctionCurve.

Data Types: struct

InstSet — Instrument variable containing a collection of instruments

structure

Instrument variable containing a collection of instruments, specified using instadd. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: struct

Output Arguments

Price — Prices of each instrument

matrix

Prices of each instrument, returned as a number of instruments (NINST) by number of curves (NUMCURVES) matrix. If an instrument cannot be priced, a NaN is returned in that entry.

For single-type pricing functions to retrieve pricing information, see the following:

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.

Version History

Introduced before R2006a

See Also

hjmprice | hjmsens | instswap | instadd | intenvsens | intenvset | bondbyzero | cfbyzero | floatbyzero | swapbyzero | fixedbyzero

Topics

“Pricing Using Interest-Rate Term Structure” on page 2-61
“Understanding the Interest-Rate Term Structure” on page 2-48

intenvsens

Instrument price and sensitivities from set of zero curves

Syntax

```
[Delta,Gamma,Price] = intenvprice(RateSpecInstSet)
```

Description

`[Delta,Gamma,Price] = intenvprice(RateSpecInstSet)` computes dollar prices and price sensitivities for instruments that use a zero coupon bond rate structure.

`intenvsens` handles the following instrument types: `'Bond'`, `'CashFlow'`, `'Fixed'`, `'Float'`, `'Swap'`. See `instadd` for information about constructing defined types.

Examples

Compute Prices Sensitivities for Instruments Using a Zero Coupon Bond Rate Structure

Load the tree and instruments from the `deriv.mat` data file and use `intenvprice` to compute dollar prices and sensitivities for instruments that use a zero coupon bond rate structure.

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

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCou
1     Bond 0.04       01-Jan-2000    01-Jan-2003    1      NaN   NaN          NaN       NaN
2     Bond 0.04       01-Jan-2000    01-Jan-2004    2      NaN   NaN          NaN       NaN

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

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

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

```
[Delta,Gamma] = intenvsens(ZeroRateSpec,ZeroInstSet)
```

```
Delta = 5×1

 -272.6403
 -347.4386
 -272.6403
   -1.0445
 -282.0405


Gamma = 5×1
10^3 ×
```

```
    1.0298
    1.6227
    1.0298
    0.0033
    1.0596
```

Input Arguments

`RateSpec` — Interest-rate specification

structure

(Optional) Interest-rate specification, specified by the `RateSpec` obtained previously from `intenvset` or `toRateSpec` for an `IRDataCurve` or `toRateSpec` for an `IRFunctionCurve`.

Data Types: `struct`

`InstSet` — Instrument variable containing a collection of instruments

structure

Instrument variable containing a collection of instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

Output Arguments

`Delta` — Rate of change of instrument prices with respect to shifts in the observed zero curve

vector

Rate of change of instrument prices with respect to shifts in the observed zero curve, returned as a number of instruments (`NINST`) by number of curves (`NUMCURVES`) matrix. `Delta` is computed by finite differences.

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

`Gamma` — Rate of change of instrument deltas with respect to shifts in the observed zero curve

vector

Rate of change of instrument deltas with respect to shifts in the observed zero curve, returned as a number of instruments (`NINST`) by number of curves (`NUMCURVES`) matrix. `Gamma` is computed by finite differences.

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

Price — Prices of each instrument
vector

Prices of each instrument, returned as a number of instruments (NINST) by number of curves (NUMCURVES) matrix. If an instrument cannot be priced, a NaN is returned in that entry.

Version History

Introduced before R2006a

See Also

hjmprice | hjmsens | instadd | intenvprice | intenvset

Topics

“Pricing Using Interest-Rate Term Structure” on page 2-61
“Understanding the Interest-Rate Term Structure” on page 2-48

intenvset

Set properties of interest-rate structure

Syntax

```
RateSpec = intenvset(Name,Value)
[RateSpec,RateSpecOld] = intenvset(RateSpec,Name,Value)
[RateSpec,RateSpecOld] = intenvset
```

Description

`RateSpec = intenvset(Name,Value)` creates an interest-rate term structure (`RateSpec`) where the input argument list is specified as name-value pairs.

Note When creating a new `RateSpec`, the set of arguments passed to `intenvset` must include `StartDates`, `EndDates`, and either `Rates` or `Disc`.

Alternatively, you can create a `RateSpec` using the Financial Instruments Toolbox object framework to construct a `ratecurve` object. For more information on converting a `RateSpec` object to a `ratecurve` object, see “Convert RateSpec to a ratecurve Object” on page 1-49.

`[RateSpec,RateSpecOld] = intenvset(RateSpec,Name,Value)` creates an interest-rate term structure (`RateSpec`) where the input argument list is specified as name-value pairs along with the optional argument `RateSpec`. If the optional argument `RateSpec` is specified, `intenvset` modifies the existing interest-rate 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-rate term structure `RateSpec` with all fields set to `[ ]`.

Examples

Create a RateSpec for a Zero Curve

Use `intenvset` to create a `RateSpec` for a zero curve.

```
RateSpec = intenvset('Rates', 0.05, 'StartDates',...
datetime(2000,1,20), 'EndDates', datetime(2021,1,20))
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: 0.3545
            Rates: 0.0500
         EndTimes: 42
       StartTimes: 0
         EndDates: 738176
       StartDates: 730505
```

```
    ValuationDate: 730505
            Basis: 0
     EndMonthRule: 1
```

Now change the `Compounding` argument to `1` (annual).

```
RateSpec = intenvset(RateSpec, 'Compounding', 1)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.3545
            Rates: 0.0506
          EndTimes: 21
        StartTimes: 0
          EndDates: 738176
        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: [ 0 | 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 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 ]
           EndMonthRule: [ 0 | {1} ]
```

Create a RateSpec for a Forward Curve

Use `intenvset` to create a `RateSpec` for a forward curve.

```
RateSpec = intenvset('Rates', 0.05, 'StartDates',...
datetime(2001,1,20), 'EndDates', datetime(2022,1,20), 'ValuationDate',datetime(2000,1,20))
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: 0.3545
            Rates: 0.0500
          EndTimes: 44
        StartTimes: 2
          EndDates: 738541
        StartDates: 730871
    ValuationDate: 730505
            Basis: 0
```

```
     EndMonthRule: 1
```

Now change the `Compounding` argument to `1` (annual).

```
RateSpec = intenvset(RateSpec, 'Compounding', 1)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.3545
            Rates: 0.0506
         EndTimes: 22
       StartTimes: 1
         EndDates: 738541
       StartDates: 730871
    ValuationDate: 730505
            Basis: 0
     EndMonthRule: 1
```

Create a RateSpec Using Two Curves

Define data for the interest-rate term structure and use `intenvset` to create a `RateSpec`.

```
StartDates = datetime(2011,11,1);
EndDates = [datetime(2012,11,1) ; datetime(2013,11,1) ; datetime(2014,11,1) ; datetime(2015,11,1
Rates = [[0.0356;0.041185;0.04489;0.047741],[0.0325;0.0423;0.0437;0.0465]];
RateSpec = intenvset('Rates', Rates, 'StartDates',StartDates,...
'EndDates', EndDates, 'Compounding', 1)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x2 double]
            Rates: [4x2 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
       StartDates: 734808
    ValuationDate: 734808
            Basis: 0
     EndMonthRule: 1
```

To look at the `Rates` for the two interest-rate curves:

```
RateSpec.Rates
```

```
ans = 4×2

    0.0356    0.0325
    0.0412    0.0423
    0.0449    0.0437
    0.0477    0.0465
```

Create a RateSpec to Price Multi-Stepped Coupon Bonds

Price the following multi-stepped coupon bonds using the following data:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2010,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
Compounding = 1;

% Create RateSpec using intenvset
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);

% Create a portfolio of stepped coupon bonds with different maturities
Settle = datetime(2010,1,1);
Maturity = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
CouponRate = {{datetime(2011,1,1) .042;datetime(2012,1,1) .05;datetime(2013,1,1) .06;datetime(201

% Display the instrument portfolio
ISet = instbond(CouponRate, Settle, Maturity, 1);
instdisp(ISet)
```

```
Index Type CouponRate Settle         Maturity       Period Basis EndMonthRule IssueDate FirstCoup
1     Bond [Cell]     01-Jan-2010    01-Jan-2011    1      0     1            NaN       NaN
2     Bond [Cell]     01-Jan-2010    01-Jan-2012    1      0     1            NaN       NaN
3     Bond [Cell]     01-Jan-2010    01-Jan-2013    1      0     1            NaN       NaN
4     Bond [Cell]     01-Jan-2010    01-Jan-2014    1      0     1            NaN       NaN
```

Build a `BDTTree` to price the stepped coupon bonds. Assume the volatility to be 10%

```
Sigma = 0.1;
BDTTimeSpec = bdttimespec(ValuationDate, EndDates, Compounding);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVolSpec, RS, BDTTimeSpec);

% Compute the price of the stepped coupon bonds
PBDT = bdtprice(BDTT, ISet)
```

```
PBDT = 4×1

  100.6763
  100.7368
  100.9266
  101.0115
```

Input Arguments

RateSpec — Interest-rate specification for initial risk-free rate curve
structure

(Optional) Interest-rate specification for initial rate curve, specified by the `RateSpec` obtained previously from `intenvset` or `toRateSpec` for an `IRDataCurve` or `toRateSpec` for an `IRFunctionCurve`.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `RateSpec = intenvset('Rates',0.05,'StartDates','20-Jan-2001','EndDates','20-Jan-2002','ValuationDate','20-Jan-2000')`

`Compounding` — Rate at which the input zero rates were compounded when annualized
`2` (default) | integer with value of `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, or `-1`

Rate at which the input zero rates were compounded when annualized, specified as the comma-separated pair consisting of `'Compounding'` and a scalar integer value. The `Compounding` argument determines the formula for the discount factors (`Disc`):

- `Compounding` = `0` for simple interest
  - `Disc = 1/(1 + Z * T)`, where `T` is time in years and simple interest assumes annual times `F = 1`.
- `Compounding` = `1`, `2`, `3`, `4`, `6`, `12`
  - `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units, for example, `T = F` is one year.
- `Compounding` = `365`
  - `Disc = (1 + Z/F)^(-T)`, where `F` is the number of days in the basis year and `T` is a number of days elapsed computed by basis.
- `Compounding` = `-1`
  - `Disc = exp(-T*Z)`, where `T` is time in years.

Data Types: `double`

`Disc` — Unit bond prices over investment intervals
`[ ]` (default) | matrix

Unit bond prices over investment intervals from `StartDates` (when the cash flow is valued) to `EndDates` (when the cash flow is received), specified as the comma-separated pair consisting of `'Disc'` and a number of points (`NPOINTS`) by number of curves (`NCURVES`) matrix.

Data Types: `double`

`Rates` — Interest rates
matrix of decimal values

Interest rates, specified as the comma-separated pair consisting of `'Rates'` and a number of points (`NPOINTS`) by number of curves (`NCURVES`) matrix of decimal values. `Rates` can only contain

negative decimal values if the resulting `RateSpec` is used with a Normal (Bachelier) model, shifted Black model, or a shifted SABR model.

Data Types: `double`

`EndDates` — Maturity dates ending the interval to discount over
datetime array | string array | date character vector

Maturity dates ending the interval to discount over, specified as the comma-separated pair consisting of `'EndDates'` and a scalar or a `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `intenvset` also accepts serial date numbers as inputs, but they are not recommended.

`StartDates` — Dates starting the interval to discount over
datetime array | string array | date character vector

Dates starting the interval to discount over, specified as the comma-separated pair consisting of `'StartDates'` and a scalar or a `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors. `StartDates` must be earlier than `EndDates`.

To support existing code, `intenvset` also accepts serial date numbers as inputs, but they are not recommended.

`ValuationDate` — Observation date of the investment horizons entered in `StartDates` and `EndDates`
`min(StartDates)` (default) | datetime array | string array | date character vector

Observation date of the investment horizons entered in `StartDates` and `EndDates`, specified as the comma-separated pair consisting of `'ValuationDate'` and a specified as a scalar datetime, string, or date character vector.

To support existing code, `intenvset` also accepts serial date numbers as inputs, but they are not recommended.

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer value.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar integer with a value of `0` or `1`. This rule applies only when `EndDates` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

Output Arguments

`RateSpec` — Interest-rate specification for initial rate curve
structure

Interest-rate specification for initial rate curve, returned as a structure.

`RateSpecOld` — Properties of an interest-rate structure before the changes introduced by the call to `intenvset`
structure

Properties of an interest-rate structure before the changes introduced by the call to `intenvset`, returned as a structure.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `intenvset` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`intenvget` | `intenvprice` | `ratecurve`

Topics

“Pricing Using Interest-Rate Term Structure” on page 2-61
“Understanding the Interest-Rate Term Structure” on page 2-48
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Convert RateSpec to a ratecurve Object” on page 1-49

isafin

True if input argument is financial structure type or financial object class

Syntax

```
IsFinObj = isafin(Obj,ClassName)
```

Description

`IsFinObj = isafin(Obj,ClassName)` returns true (`1`) if input argument is a financial structure type or financial object class, otherwise false (`0`) is returned.

Examples

Determine if Input is Financial Structure Type or Financial Object Class

`isafin` returns true (`1`) if input argument is a financial structure type or financial object class, otherwise false (`0`) is returned.

```
load deriv.mat
IsFinObj = isafin(HJMTree, 'HJMFwdTree')
```

```
IsFinObj = logical
   1
```

Input Arguments

`Obj` — Name of a financial structure

object

Name of a financial structure, specified as an object.

Data Types: `object`

`ClassName` — Name of financial structure class

character vector

Name of a financial structure class, specified as a character vector.

Data Types: `char`

Output Arguments

`IsFinObj` — Is input argument is financial structure type or financial object class

logical

Is input argument is financial structure type or financial object class, returned as a logical. `isafin` returns true (`1`) if input argument is a financial structure type or financial object class, otherwise false (`0`).

Version History

Introduced before R2006a

See Also

`classfin`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Instrument Constructors” on page 1-15

ittprice

Price instruments using implied trinomial tree (ITT)

Syntax

```
[Price,PriceTree] = ittprice(ITTTree,InstSet)
[Price,PriceTree] = ittprice( ___ ,Options)
```

Description

`[Price,PriceTree] = ittprice(ITTTree,InstSet)` price instruments using an implied trinomial tree (ITT) created with `itttree`. All instruments contained in a financial instrument variable, `InstSet`, are priced.

`ittprice` handles the following instrument types: optstock, barrier, Asian, lookback, and compound. Use `instadd` to construct the defined types.

`[Price,PriceTree] = ittprice( ___ ,Options)` adds an optional input argument for `Options`.

Examples

Price Instruments Using Implied Trinomial Tree (ITT)

Load the ITT tree and instruments from the data file `deriv.mat`.

```
load deriv.mat
```

Display the barrier and Asian options contained in the instrument set.

```
ITTSubSet = instselect(ITTInstSet,'Type', {'Barrier', 'Asian'});

instdisp(ITTSubSet)
```

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

Index Type  OptSpec Strike Settle         ExerciseDates  AmericanOpt AvgType    AvgPrice AvgDate
2     Asian call    55     01-Jan-2006    01-Jan-2008    0           arithmetic NaN      NaN
3     Asian call    55     01-Jan-2006    01-Jan-2010    0           arithmetic NaN      NaN
```

Price the barrier and Asian options contained in the instrument set.

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

```
Price = 3×1

    2.4074
    3.2052
    6.6074
```

```
PriceTree = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {1x5 cell}
      tObs: [0 1 2 3 4]
      dObs: [732678 733043 733408 733773 734139]
```

Input Arguments

`ITTTree` — Implied trinomial stock tree structure

structure

Implied trinomial stock tree structure, specified by using `itttree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

(Optional) Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Price for each instrument

vector

Price for each instrument, returned as a `NINST`-by-`1` vector. 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.

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
- `cbondbyitt` for pricing convertible bonds using an ITT tree

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as 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.tObs` contains the observation times.
- `PriceTree.dObs` contains the observation dates.

Version History

Introduced in R2007a

See Also

`ittsens` | `itttree`

Topics

“Computing Prices Using ITT” on page 3-68
“Examining Output from the Pricing Functions” on page 3-70
“Computing Equity Instrument Sensitivities” on page 3-75
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

ittsens

Instrument sensitivities and prices using implied trinomial tree (ITT)

Syntax

```
[Delta,Gamma,Vega,Price] = ittsens(ITTTree,InstSet)
[Delta,Gamma,Vega,Price] = ittsens( ___ ,Options)
```

Description

`[Delta,Gamma,Vega,Price] = ittsens(ITTTree,InstSet)` calculates instrument sensitivities and prices using an implied trinomial tree (ITT) that is created with the `itttree` function. All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide by the respective instrument price.

`ittsens` handles the following instrument types: optstock, barrier, Asian, lookback, and compound. Use `instadd` to construct the defined types.

`[Delta,Gamma,Vega,Price] = ittsens( ___ ,Options)` adds an optional input argument for `Options`.

Examples

Compute Instrument Sensitivities Using an Implied Trinomial Tree (ITT)

Load the ITT tree and instruments from the data file `deriv.mat` and display the vanilla options and barrier option instruments.

```
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         ExerciseDates  AmericanOpt BarrierSpec Barrier Rebate
3     Barrier call    85     01-Jan-2006    31-Dec-2008    1           ui          115     0
```

Compute the `Delta` and `Gamma` sensitivities of vanilla options and barrier option contained in the instrument set.

```
[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


Delta = 3×1

    0.2387
   -0.4283
    0.3482


Gamma = 3×1

    0.0260
    0.0188
    0.0380
```

Input Arguments

`ITTTree` — Stock tree structure

structure

Stock tree structure, specified by using `itttree`.

Data Types: `struct`

`InstSet` — Instrument variable

structure

Instrument variable containing a collection of `NINST` instruments, specified using `instadd`. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or character vector for each instrument.

Data Types: `struct`

`Options` — Derivatives pricing options structure

structure

Derivatives pricing options structure, created using `derivset`.

Data Types: `struct`

Output Arguments

`Delta` — Rate of change of instruments prices with respect to changes in the stock price

vector

Rate of change of instruments prices with respect to changes in the stock price, returned as a `NINST`-by-`1` vector of deltas.

For path-dependent options (`'Lookback'` and `'Asian'`), `Delta` and `Gamma` are computed by finite differences in calls to `ittprice`. For the rest of the options (`'OptStock'`, `'Barrier'`, `'CBond'`,

and 'Compound'), Delta and Gamma are computed from the ITTTree and the corresponding option price tree.

Gamma — Rate of change of instruments deltas with respect to changes in stock price
vector

Rate of change of instruments deltas with respect to changes in the stock price, returned as a NINST-by-1 vector of gammas.

For path-dependent options ('Lookback' and 'Asian'), Delta and Gamma are computed by finite differences in calls to ittprice. For the rest of the options ('OptStock', 'Barrier', 'CBond', and 'Compound'), Delta and Gamma are computed from the ITTTree and the corresponding option price tree.

Vega — Rate of change of instruments prices with respect to changes in volatility of the stock
vector

Rate of change of instruments prices with respect to changes in the volatility of the stock, returned as a NINST-by-1 vector of vegas. Vega is computed by finite differences in calls to itttree.

Price — Price of each instrument
vector

Price of each instrument, returned as a NINST-by-1 vector. 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.

Version History

Introduced in R2007a

References

[1] Chriss, Neil. *Black-Scholes and Beyond: Option Pricing Models.* McGraw-Hill, 1996, pp 308-312.

See Also

ittprice | itttree

Topics

“Computing Prices Using ITT” on page 3-68
“Examining Output from the Pricing Functions” on page 3-70
“Computing Equity Instrument Sensitivities” on page 3-75
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

itttimespec

Specify time structure using implied trinomial tree (ITT)

Syntax

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

Description

`TimeSpec = itttimespec(ValuationDate,Maturity,NumPeriods)` creates the structure specifying the time layout for an ITT tree (`itttree`).

Examples

Creates the Structure Specifying the Time Layout for an ITT Tree

This example shows how to specify a four-period tree with time steps of 1 year.

```
ValuationDate = datetime(2006,7,1);
Maturity = datetime(2010,7,1);
TimeSpec = itttimespec(ValuationDate, Maturity, 4)
```

```
TimeSpec = struct with fields:
           FinObj: 'ITTTimeSpec'
    ValuationDate: 732859
         Maturity: 734320
       NumPeriods: 4
            Basis: 0
     EndMonthRule: 1
             tObs: [0 1 2 3 4]
             dObs: [732859 733224 733589 733954 734320]
```

Input Arguments

`ValuationDate` — Pricing date and first observation in the tree

datetime scalar | string scalar | date character vector

Pricing date and first observation in the `itttree`, specified as a scalar datetime, string, or date character vector.

To support existing code, `itttimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Date marking the depth of the EQP stock tree

datetime scalar | string scalar | date character vector

Date marking the depth of the `itttree` trinomial tree, specified as scalar datetime, string, or date character vector.

To support existing code, `itttimespec` also accepts serial date numbers as inputs, but they are not recommended.

`NumPeriods` — Number of time steps in the ITT tree
integer

Number of time steps in the `itttree` trinomial tree, specified as scalar integer value.

Data Types: `double`

Output Arguments

`TimeSpec` — Specification for the time layout for `itttree`
structure

Specification for the time layout for `itttree`, returned as a structure.

Version History

Introduced in R2007a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `itttimespec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`ittprice` | `itttree` | `stockspec`

Topics
“Building Implied Trinomial Trees” on page 3-6
“Examining Equity Trees” on page 3-14
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Understanding Equity Trees” on page 3-2
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

itttree

Build implied trinomial stock tree

Syntax

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

Description

`ITTTree = itttree(StockSpec,RateSpec,TimeSpec,StockOptSpec)` builds an implied trinomial (ITT) stock tree.

Examples

Create an ITT Tree

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 the `StockSpec`, `TimeSpec`, and `StockOptSpec` structures.

```
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);

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

TimeSpec = itttimespec(ValuationDate, EndDate, NumPeriods);
```

Build a `StockOptSpec` structure.

```
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);
```

Use `itttree` to build the `ITTTree` structure. Note, in this example, the extrapolation warnings are turned on. 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.

```
warning('on', 'fininst:itttree:Extrapolation');
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]}
```

Input Arguments

`StockSpec` — Stock specification

structure

Stock specification, specified by the `StockSpec` obtained from `stockspec`. See `stockspec` for information on creating a stock specification.

Data Types: `struct`

`RateSpec` — Interest-rate specification for initial risk-free rate curve

structure

Interest-rate specification for initial risk-free rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`TimeSpec` — Tree time layout specification

structure

Tree time layout specification, specified by the `TimeSpec` obtained from `itttimespec`. The `TimeSpec` defines the observation dates of the ITT tree. See `itttimespec` for information on the tree structure.

Data Types: `struct`

`StockOptSpec` — Option stock specification

structure

Option stock specification, specified by the `StockOptSpec` obtained from `stockoptspec`. See `stockoptspec` for information on creating a stock specification.

Data Types: `struct`

Output Arguments

`ITTTree` — ITT trinomial tree

structure

ITT trinomial tree, returned as a structure specifying the time layout for the tree.

Version History

Introduced in R2007a

See Also

`intenvset` | `ittprice` | `itttree` | `stockspec` | `itttimespec` | `stockoptspec`

Topics

“Building Implied Trinomial Trees” on page 3-6

“Examining Equity Trees” on page 3-14
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Understanding Equity Trees” on page 3-2
“Pricing Options Structure” on page A-2
“Supported Equity Derivative Functions” on page 3-19

LiborMarketModel

Create LIBOR Market Model

Description

The LIBOR Market Model (LMM) is an interest-rate model that differs from short rate models in that it evolves a set of discrete forward rates.

Specifically, the lognormal LMM specifies the following diffusion equation for each forward rate

$$\frac{dF_i(t)}{F_i} = -\mu_i dt + \sigma_i(t) dW_i$$

where:

W is an N-dimensional geometric Brownian motion with

$$dW_i(t) dW_j(t) = \rho_{ij}$$

The LMM relates drifts of the forward rates based on no-arbitrage arguments. Specifically, under the Spot LIBOR measure, drifts are expressed as

$$\mu_i(t) = -\sigma_i(t) \sum_{j=q(t)}^{i} \frac{\tau_j \rho_{i,j} \sigma_j(t) F_j(t)}{1 + \tau_j F_j(t)}$$

where:

$\rho_{i,j}$ represents the input argument `Correlation`.

$\sigma_j(t)$ represents the input argument `VolFunc`.

$F_j(t)$ represents the computation of the input argument for `ZeroCurve`.

τ_i is the time fraction associated with the i th forward rate

$q(t)$ is an index defined by the relation

$$T_{q(t)-1} < t < T_{q(t)}$$

and the Spot LIBOR numeraire is defined as

$$B(t) = P(t, T_{q(t)}) \prod_{n=0}^{q(t)-1} (1 + \tau_n F_n(T_n))$$

Creation

Syntax

```
LMM = LiborMarketModel(ZeroCurve,VolFunc,Correlation)
LMM = LiborMarketModel( ___ ,Name,Value)
```

Description

`LMM = LiborMarketModel(ZeroCurve,VolFunc,Correlation)` creates a `LiborMarketModel` (`LMM`) object using the required arguments for `ZeroCurve`, `VolFunc`, `Correlation`.

Note Alternatively, you can use the `SABRBraceGatarekMusiela` or `BraceGatarekMusiela` model objects to create a LIBOR market model. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`LMM = LiborMarketModel( ___ ,Name,Value)` sets Properties on page 11-964 using name-value pairs. For example, `LMM = LiborMarketModel(irdc,VolFunc,Correlation,'Period',1)`. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Input Arguments

`ZeroCurve` — Zero curve used to evolve path of future interest rates

`IRDataCurve` object | `RateSpec`

Zero curve used to evolve the path of future interest rates, specified as an output from `IRDataCurve` or a `RateSpec` that is obtained from `intenvset`. The `ZeroCurve` input sets the ZeroCurve on page 11-0 property.

Data Types: `object` | `struct`

`VolFunc` — Volatility function

cell array of function handles

Volatility function, specified using a `NumRates`-by-`1` cell array of function handles and sets the VolFunc on page 11-0 property. Each function handle must take time as an input and, return a scalar volatility.

Note The number of rates to simulate using the `simTermStructs` function is determined by the size of the `VolFunc` and `Correlation` inputs which must be consistent. These can be any value and, together with the `Period` property, determines the kinds and number of rates being simulated. For example, if the `Period` is set to `4` (quarterly) and `VolFunc` has length of `120` and `Correlation` has size `120`-by-`120`, then `120` quarterly rates are simulated. In other words, 30 years of the yield curve are simulated (0-3mos, 3mos-6mos, 6mos-9mos, and so on, all the way up to 30 years). Therefore, if `VolFunc` and `Correlation` have size `120`, the output of a call to `simTermStructs` is (`nPeriods` +`1`) -by-`121`-by-`nTrials`.

Data Types: `cell`

`Correlation` — Correlation matrix

matrix

Correlation matrix, specified using a NumRates-by-NumRates correlation matrix and sets the Correlation on page 11-0 property.

Note The number of rates to simulate using the simTermStructs function is determined by the size of the VolFunc and Correlation inputs which must be consistent. These can be any value and, together with the Period property, determines the kinds and number of rates being simulated. For example, if the Period is set to 4 (quarterly) and VolFunc has length of 120 and Correlation has size 120-by-120, then 120 quarterly rates are simulated. In other words, 30 years of the yield curve are simulated (0-3mos, 3mos-6mos, 6mos-9mos, and so on, all the way up to 30 years). Therefore, if VolFunc and Correlation have size 120, the output of a call to simTermStructs is (nPeriods +1) -by-121-by-nTrials.

Data Types: double

Properties

ZeroCurve — Zero curve

IRDataCurve object | RateSpec

Zero curve, specified as an output from IRDataCurve or a RateSpec that is obtained from intenvset.

Data Types: object | struct

VolFunc — Volatility function

cell array of function handles

Volatility function, specified using a NumRates-by-1 cell array of function handles. Each function handle must take time as an input and, return a scalar volatility.

Data Types: cell

Correlation — Correlation matrix

matrix

Correlation matrix, specified using a NumRates-by-NumRates correlation matrix.

Data Types: double

NumFactors — Number of Brownian factors

NaN (default) | numeric

Number of Brownian factors, specified as a numeric value. The default is NaN, where the number of factors is equal to the number of rates.

Data Types: double

Period — Period of forward rates, specifically number of rates per year

2 (default) | numeric with value 1, 2, 4, or 12

Period of the forward rates, specifically the number of rates per year, specified as a numeric value of 1, 2, 4, or 12. The default value is 2, meaning forward rates are spaced at 0, .5, 1, 1.5, and so on.

Data Types: double

Object Functions

simTermStructs Simulate term structures for LIBOR Market Model

Examples

Create a LIBOR Market Model Using an IRDataCurve

Create a LMM object using an IRDataCurve.

```
Settle = datetime(2007,12,15);
CurveTimes = [1:5 7 10 20]';
ZeroRates = [.01 .018 .024 .029 .033 .034 .035 .034]';
CurveDates = daysadd(Settle,360*CurveTimes,1);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

LMMVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
LMMVolParams = [.3 -.02 .7 .14];

numRates = 20;
VolFunc(1:numRates,1) = {@(t) LMMVolFunc(LMMVolParams,t)};

Beta = .08;
CorrFunc = @(i,j,Beta) exp(-Beta*abs(i-j));
Correlation = CorrFunc(meshgrid(1:numRates)',meshgrid(1:numRates),Beta);

LMM = LiborMarketModel(irdc,VolFunc,Correlation,'Period',1)
```

```
LMM =
  LiborMarketModel with properties:

       ZeroCurve: [1x1 IRDataCurve]
    VolFunctions: {20x1 cell}
     Correlation: [20x20 double]
      NumFactors: NaN
          Period: 1
```

Simulate the term structures for the specified LMM object.

```
[ZeroRates, ForwardRates] = simTermStructs(LMM, 10,'nTrials',100);
```

Create a LIBOR Market Model Using a RateSpec

Create a LMM object using a RateSpec.

```
Settle = datetime(2007,12,15);
CurveTimes = [1:5 7 10 20]';
ZeroRates = [.01 .018 .024 .029 .033 .034 .035 .034]';
CurveDates = daysadd(Settle,360*CurveTimes,1);

RateSpec = intenvset('Rates',ZeroRates,'EndDates',CurveDates,'StartDate',Settle);
```

```
LMMVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
LMMVolParams = [.3 -.02 .7 .14];

numRates = 20;
VolFunc(1:numRates,1) = {@(t) LMMVolFunc(LMMVolParams,t)};

Beta = .08;
CorrFunc = @(i,j,Beta) exp(-Beta*abs(i-j));
Correlation = CorrFunc(meshgrid(1:numRates)',meshgrid(1:numRates),Beta);

LMM = LiborMarketModel(RateSpec,VolFunc,Correlation,'Period',1)

LMM =
  LiborMarketModel with properties:

       ZeroCurve: [1x1 IRDataCurve]
    VolFunctions: {20x1 cell}
     Correlation: [20x20 double]
      NumFactors: NaN
          Period: 1
```

Simulate the term structures for the specified `LMM` object.

```
[ZeroRates, ForwardRates] = simTermStructs(LMM, 10,'nTrials',100);
```

More About

LIBOR Market Model

The LIBOR Market Model, also called the BGM Model (Brace, Gatarek, Musiela Model) is a financial model of interest rates.

The quantities that are modeled are a set of forward rates (also called forward LIBORs) which have the advantage of being directly observable in the market, and whose volatilities are naturally linked to traded contracts.

Version History

Introduced in R2013a

References

[1] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

See Also

`HullWhite1F` | `intenvset` | `IRDataCurve` | `LinearGaussian2F` | `simTermStructs` | `SABRBraceGatarekMusiela` | `BraceGatarekMusiela`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Supported Interest-Rate Instrument Functions” on page 2-3

“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

simTermStructs

Simulate term structures for LIBOR Market Model

Syntax

```
[ZeroRates,ForwardRates] = simTermStructs(LMM,nPeriods)
[ZeroRates,ForwardRates] = simTermStructs(___,Name,Value)
```

Description

`[ZeroRates,ForwardRates] = simTermStructs(LMM,nPeriods)` simulates future zero curve paths using a specified `LiborMarketModel` object.

`[ZeroRates,ForwardRates] = simTermStructs(___,Name,Value)` adds optional name-value pair arguments.

Examples

Simulate Term Structures for a LIBOR Market Model

Create a `LMM` object.

```
Settle = datetime(2007,12,15);
CurveTimes = [1:5 7 10 20]';
ZeroRates = [.01 .018 .024 .029 .033 .034 .035 .034]';
CurveDates = daysadd(Settle,360*CurveTimes,1);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

LMMVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
LMMVolParams = [.3 -.02 .7 .14];

numRates = 20;
VolFunc(1:numRates-1) = {@(t) LMMVolFunc(LMMVolParams,t)};

Beta = .08;
CorrFunc = @(i,j,Beta) exp(-Beta*abs(i-j));
Correlation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),Beta);

LMM = LiborMarketModel(irdc,VolFunc,Correlation,'Period',1)
```

```
LMM =
  LiborMarketModel with properties:

       ZeroCurve: [1x1 IRDataCurve]
    VolFunctions: {1x19 cell}
     Correlation: [19x19 double]
      NumFactors: NaN
          Period: 1
```

Simulate the term structures for the specified LMM object.

```
[ZeroRates, ForwardRates] = simTermStructs(LMM, 20,'nTrials',100);
```

Input Arguments

LMM — LiborMarketModel object

object

LiborMarketModel object, specified using the LMM object created using LiborMarketModel.

Data Types: object

nPeriods — Number of simulation periods

numeric

Number of simulation periods, specified as a numeric value. The nPeriods value is determined by the swaption expiry and the periodicity of the rates of the model. For example, if you were to price a swaption expiring in 5 years with a semiannual LIBOR Market Model (LMM), then nPeriods would be 10.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [ZeroRates, ForwardRates] = simTermStructs(LMM, 20,'nTrials',100)

nTrials — Number of simulated trials

1 (default) | positive integer

Number of simulated trials (sample paths), specified as the comma-separated pair consisting of 'nTrials' and a positive scalar integer value of nPeriods observations each. If you do not specify a value for this argument, the default is 1, indicating a single path of correlated state variables.

Data Types: double

antithetic — Flag indicating whether antithetic sampling is used to generate Gaussian random variates

false (default) | positive integer

Flag indicating whether antithetic sampling is used to generate the Gaussian random variates that drive the zero-drift, unit-variance rate Brownian vector $dW(t)$, specified as the comma-separated pair consisting of 'antithetic' and a Boolean scalar flag. For details on the Brownian vector, see simBySolution.

Data Types: logical

Z — Direct specification of dependent random noise process

generated by simBySolution function (default) | numeric

Direct specification of the dependent random noise process, specified as the comma-separated pair consisting of `'Z'` and a numeric value. The `Z` value is used to generate the zero-drift, unit-variance rate Brownian vector $dW(t)$ that drives the simulation. For details, see `simBySolution` for the GBM model.

Data Types: `double`

`Tenor` — Maturities to compute at each time step
number of rates in `LiborMarketModel` object (default) | numeric vector

Maturities to compute at each time step, specified as the comma-separated pair consisting of `'Tenor'` and a numeric vector.

`Tenor` enables you to choose a different set of rates to output than the underlying rates. For example, you may want to simulate quarterly data but only report annual rates; this can be done by specifying the optional input `Tenor`.

The default for `tenor` is the number of rates in the `LiborMarketModel` object as specified by the `Correlation` and `VolFunc` input arguments for the `LiborMarketModel` object.

Data Types: `double`

Output Arguments

`ZeroRates` — Simulated zero-rate term structures
matrix

Simulated zero-rate term structures, returned as a `nPeriods+1`-by-`nTenors`-by-`nTrials` matrix.

`ForwardRates` — Simulated forward-rate term structures
matrix

Simulated zero-rate term structures, returned as a `nPeriods+1`-by-`nTenors`-by-`nTrials` matrix.

Version History

Introduced in R2013a

See Also

`LiborMarketModel` | `blackvolbyrebonato`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Supported Interest-Rate Instrument Functions” on page 2-3

LinearGaussian2F

Create two-factor additive Gaussian interest-rate model

Description

The two-factor additive Gaussian interest rate-model is specified using the zero curve, a, b, sigma, eta, and rho parameters.

Specifically, the LinearGaussian2F model is defined using the following equations:

$$r(t) = x(t) + y(t) + \phi(t)$$

$$dx(t) = -a(t)x(t)dt + \sigma(t)dW_1(t), x(0) = 0$$

$$dy(t) = -b(t)y(t)dt + \eta(t)dW_2(t), y(0) = 0$$

where $dW_1(t)dW_2(t) = \rho dt$ is a two-dimensional Brownian motion with correlation ρ, and ϕ is a function chosen to match the initial zero curve.

Creation

Syntax

```
G2PP = LinearGaussian2F(ZeroCurve,a,b,sigma,eta,rho)
```

Description

`G2PP = LinearGaussian2F(ZeroCurve,a,b,sigma,eta,rho)` creates a `LinearGaussian2F` (`G2PP`) object using the required arguments to set the Properties on page 11-971.

Note Alternatively, you can use the `LinearGaussian2F` model object to create a two-factor additive Gaussian interest-rate model. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

Properties

ZeroCurve — Zero curve

`IRDataCurve` object | `RateSpec`

Zero curve, specified as an output from `IRDataCurve` or a `RateSpec` that is obtained from `intenvset`. This is the zero curve used to evolve the path of future interest rates.

Data Types: `object` | `struct`

a — Mean reversion for first factor

numeric

Mean reversion for the first factor, specified either as a scalar or function handle which takes time as input and returns a scalar mean reversion value.

Data Types: `double`

b — Mean reversion for second factor
numeric

Mean reversion for the second factor, specified either as a scalar or as a function handle which takes time as input and returns a scalar mean reversion value.

Data Types: `double`

`sigma` — Volatility for first factor
numeric

Volatility for the first factor, specified either as a scalar or function handle which takes time as input and returns a scalar mean volatility.

Data Types: `double`

`eta` — Volatility for second factor
numeric

Volatility for the second factor, specified either as a scalar or function handle which takes time as input and returns a scalar mean volatility.

Data Types: `double`

`rho` — Scalar correlation of factors
numeric

Scalar correlation of the factors, specified as a numeric value.

Data Types: `double`

Object Functions

simTermStructs Simulate term structures for two-factor additive Gaussian interest-rate model

Examples

Create a Two-Factor Additive Gaussian Interest-Rate Model Using an `IRDataCurve`

Create a two-factor additive Gaussian interest-rate model using an `IRdataCurve`.

```
Settle = datetime(2007,12,15);
CurveTimes = [1:5 7 10 20]';
ZeroRates = [.01 .018 .024 .029 .033 .034 .035 .034]';
CurveDates = daysadd(Settle,360*CurveTimes,1);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

a = .07;
b = .5;
sigma = .01;
```

```
eta = .006;
rho = -.7;

G2PP = LinearGaussian2F(irdc,a,b,sigma,eta,rho)
```

```
G2PP =
  LinearGaussian2F with properties:

    ZeroCurve: [1x1 IRDataCurve]
            a: @(t,V)ina
            b: @(t,V)inb
        sigma: @(t,V)insigma
          eta: @(t,V)ineta
          rho: -0.7000
```

Use the `simTermStructs` method to simulate term structures based on the `LinearGaussian2F` model.

```
 SimPaths = simTermStructs(G2PP, 10,'nTrials',100);
```

Create a Two-Factor Additive Gaussian Interest-Rate Model Using a RateSpec

Create a two-factor additive Gaussian interest-rate model using a `RateSpec`.

```
Settle = datetime(2007,12,15);
CurveTimes = [1:5 7 10 20]';
ZeroRates = [.01 .018 .024 .029 .033 .034 .035 .034]';
CurveDates = daysadd(Settle,360*CurveTimes,1);

RateSpec = intenvset('Rates',ZeroRates,'EndDates',CurveDates,'StartDate',Settle);

a = .07;
b = .5;
sigma = .01;
eta = .006;
rho = -.7;

G2PP = LinearGaussian2F(RateSpec,a,b,sigma,eta,rho)
```

```
G2PP =
  LinearGaussian2F with properties:

    ZeroCurve: [1x1 IRDataCurve]
            a: @(t,V)ina
            b: @(t,V)inb
        sigma: @(t,V)insigma
          eta: @(t,V)ineta
          rho: -0.7000
```

Use the `simTermStructs` method to simulate term structures based on the `LinearGaussian2F` model.

```
 SimPaths = simTermStructs(G2PP, 10,'nTrials',100);
```

More About

Two-Factor Additive Gaussian Interest-Rate Model

Short-rate model based on two factors where the short rate is the sum of the two factors and a deterministic function.

In this case $\phi(t)$, which is chosen to match the initial term structure.

Version History

Introduced in R2013a

References

[1] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

See Also

`HullWhite1F` | `LiborMarketModel` | `simTermStructs` | `capbylg2f` | `floorbylg2f` | `swaptionbylg2f` | `LinearGaussian2F`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

simTermStructs

Simulate term structures for two-factor additive Gaussian interest-rate model

Syntax

```
[ZeroRates,ForwardRates] = simTermStructs(G2PP,nPeriods)
[ZeroRates,ForwardRates] = simTermStructs(___,Name,Value)
```

Description

`[ZeroRates,ForwardRates] = simTermStructs(G2PP,nPeriods)` simulates future zero curve paths using a specified `LinearGaussian2F` object.

`[ZeroRates,ForwardRates] = simTermStructs(___,Name,Value)` adds optional name-value pair arguments.

Examples

Simulate Term Structures for the LinearGaussian2F Model

Create a two-factor additive Gaussian interest-rate model.

```
Settle = datetime(2007,12,15);
CurveTimes = [1:5 7 10 20]';
ZeroRates = [.01 .018 .024 .029 .033 .034 .035 .034]';
CurveDates = daysadd(Settle,360*CurveTimes,1);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

a = .07;
b = .5;
sigma = .01;
eta = .006;
rho = -.7;

G2PP = LinearGaussian2F(irdc,a,b,sigma,eta,rho)
```

```
G2PP =
  LinearGaussian2F with properties:

    ZeroCurve: [1x1 IRDataCurve]
            a: @(t,V)ina
            b: @(t,V)inb
        sigma: @(t,V)insigma
          eta: @(t,V)ineta
          rho: -0.7000
```

Use the `simTermStructs` method to simulate term structures based on the `LinearGaussian2F` model.

```
SimPaths = simTermStructs(G2PP, 10,'nTrials',100);
```

Simulate Term Structures for the LinearGaussian2F Model Using a Vector for deltaTime

Create a two-factor additive Gaussian interest-rate model.

```
Settle = datetime(2007,12,15);
CurveTimes = [1:5 7 10 20]';
ZeroRates = [.01 .018 .024 .029 .033 .034 .035 .034]';
CurveDates = daysadd(Settle,360*CurveTimes,1);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);
a = .07;
b = .5;
sigma = .01;
eta = .006;
rho = -.7;
G2PP = LinearGaussian2F(irdc,a,b,sigma,eta,rho)
```

```
G2PP =
  LinearGaussian2F with properties:

    ZeroCurve: [1x1 IRDataCurve]
            a: @(t,V)ina
            b: @(t,V)inb
        sigma: @(t,V)insigma
          eta: @(t,V)ineta
          rho: -0.7000
```

Use the `simTermStructs` method to simulate term structures based on the `LinearGaussian2F` object, where uneven simulation tenors are specified using the optional name-value argument `deltaTime` as a vector of length `NPeriods`.

```
NPeriods = 10;
dt = rand(NPeriods,1);
SimPaths = G2PP.simTermStructs(NPeriods,'nTrials',100,'DeltaTime',dt);
```

Input Arguments

G2PP — LinearGaussian2F object

object

`LinearGaussian2F` object, specified using the `G2PP` object created using `LinearGaussian2F`.

Data Types: `object`

nPeriods — Number of simulation periods

numeric

Number of simulation periods, specified as a numeric value. For example, to simulate 12 years with an annual spacing, specify `12` as the `nPeriods` input and `1` as the optional `deltaTime` input (note that the default value for `deltaTime` is `1`).

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[ZeroRates,ForwardRates] = simTermStructs(G2PP,NPeriods,'nTrials',100,'deltaTime',dt)`

`deltaTime` — Time step between `nPeriods`

1 (default) | numeric

Time step between `nPeriods`, specified as the comma-separated pair consisting of `'deltaTime'` and a numeric scalar or vector. For example, to simulate 12 years with an annual spacing, specify `12` as the `nPeriods` input and `1` as the optional `deltaTime` input (note that the default value for `deltaTime` is `1`).

Data Types: `double`

`nTrials` — Number of simulated trials

1 (default) | positive integer

Number of simulated trials (sample paths), specified as the comma-separated pair consisting of `'nTrials'` and a positive scalar integer value of `nPeriods` observations each. If you do not specify a value for this argument, the default is `1`, indicating a single path of correlated state variables.

Data Types: `double`

`antithetic` — Flag indicating whether antithetic sampling is used to generate Gaussian random variates

`false` (default) | positive integer

Flag indicating whether antithetic sampling is used to generate the Gaussian random variates that drive the zero-drift, unit-variance rate Brownian vector $dW(t)$, specified as the comma-separated pair consisting of `'antithetic'` and a Boolean scalar flag. For details, see `simBySolution` for the HWV model.

Data Types: `logical`

`Z` — Direct specification of dependent random noise process

Gaussian variates generated by `simBySolution` function (default) | numeric

Direct specification of the dependent random noise process, specified as the comma-separated pair consisting of `'Z'` and a numeric value. The `Z` value is used to generate the zero-drift, unit-variance rate Brownian vector $dW(t)$ that drives the simulation. For details, see `simBySolution` for the HWV model. If you do not specify a value for `Z`, `simBySolution` generates Gaussian variates.

Data Types: `double`

`Tenor` — Maturities to compute at each time step

tenor of `LinearGaussian2F` object zero curve (default) | numeric vector

Maturities to compute at each time step, specified as the comma-separated pair consisting of `'Tenor'` and a numeric vector.

`Tenor` enables you to choose a different set of rates to output than the underlying rates. For example, you may want to simulate quarterly data but only report annual rates; this can be done by specifying the optional input `Tenor`.

Data Types: `double`

Output Arguments

`ZeroRates` — Simulated zero-rate term structures

matrix

Simulated zero-rate term structures, returned as a `nPeriods+1`-by-`nTenors`-by-`nTrials` matrix.

`ForwardRates` — Simulated forward-rate term structures

matrix

Simulated zero-rate term structures, returned as a `nPeriods+1`-by-`nTenors`-by-`nTrials` matrix. The `ForwardRates` output is computed using the simulated short rates and by using the model definition to recover the entire yield curve at each simulation date.

Version History

Introduced in R2013a

See Also

`LinearGaussian2F`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Pricing Bermudan Swaptions with Monte Carlo Simulation” on page 2-114
“Supported Interest-Rate Instrument Functions” on page 2-3

lookbackbycrr

Price lookback option from Cox-Ross-Rubinstein binomial tree

Syntax

```
Price = lookbackbycrr(CRRTree,OptSpec,Strike,Settle,ExerciseDates)
Price = lookbackbycrr( ___ ,AmericanOpt)
```

Description

`Price = lookbackbycrr(CRRTree,OptSpec,Strike,Settle,ExerciseDates)` prices lookback options using a Cox-Ross-Rubinstein binomial tree.

Note Alternatively, you can use the `Lookback` object to price lookback options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = lookbackbycrr( ___ ,AmericanOpt)` adds an optional argument for `AmericanOpt`.

Examples

Price a Lookback Option Using a CRR Binomial Tree

This example shows how to price a lookback option using a CRR binomial tree by loading 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;

OptSpec = 'Call';
Strike = 115;
Settle = datetime(2003,1,1);
ExerciseDates = datetime(2006,1,1);

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

```
Price = 7.6015
```

Input Arguments

`CRRTree` — Stock tree structure

structure

Stock tree structure for a Cox-Ross-Rubinstein binomial tree, specified by using `crrtree`.

Data Types: `struct`

OptSpec — Definition of option
character vector with value 'call' or 'put' | cell array of character vectors with values 'call' or 'put'

Definition of option, specified as 'call' or 'put' using a character vector or an NINST-by-1 cell array of character vectors for 'call' or 'put'.

Data Types: char | cell

Strike — Option strike price value
matrix of nonnegative integers

Option strike price value, specified with a nonnegative integer using an NINST-by-1 matrix of strike price values. Each row is the schedule for one option.

To compute the value of a floating-strike lookback option, Strike must be specified as NaN. Floating-strike lookback options are also known as average strike options.

Data Types: double

Settle — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date for the lookback option, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

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

To support existing code, lookbackbycrr also accepts serial date numbers as inputs, but they are not recommended.

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use an NINST-by-1 matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one ExerciseDates on the option expiry date.
- For an American option, use an NINST-by-2 vector of exercise date boundaries. 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 an NINST-by-1 vector of dates, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

To support existing code, lookbackbycrr also accepts serial date numbers as inputs, but they are not recommended.

AmericanOpt — Option type
0 European (default) | integer with values 0 or 1

(Optional) Option type, specified as an NINST-by-1 integer flags with values:

- 0 — European

- `1` — American

Data Types: `single` | `double`

Output Arguments

`Price` — Expected prices for lookback options at time `0`

vector

Expected prices for lookback options at time 0, returned as an `NINST`-by-`1` vector. Pricing of lookback options is done using Hull-White (1993). Therefore, for these options there are no unique prices on the tree nodes except for the root node.

More About

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 Instruments 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. For more information, see “Lookback Option” on page 3-39.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `lookbackbycrr` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] 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` | `Lookback`

Topics

lookbackbycvgsg

Calculate prices of European lookback options using Conze-Viswanathan and Goldman-Sosin-Gatto models

Syntax

```
Price = lookbackbycvgsg(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates)
Price = lookbackbycvgsg( ___ ,Name,Value)
```

Description

`Price = lookbackbycvgsg(RateSpec,StockSpec,OptSpec,Strike,Settle, ExerciseDates)` returns prices of European lookback options using Conze-Viswanathan and Goldman-Sosin-Gatto models. `lookbackbycvgsg` calculates prices of European fixed- and floating-strike lookback options. To compute the value of a floating-strike lookback option, `Strike` must be specified as `NaN`. The Goldman-Sosin-Gatto model is used for floating-strike lookback options. The Conze-Viswanathan model is used for fixed-strike lookback options.

Note Alternatively, you can use the `Lookback` object to price lookback options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = lookbackbycvgsg( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of a Floating Lookback Option Using the Goldman-Sosin-Gatto Model

Define the `RateSpec`.

```
StartDates = datetime(2013,1,1);
EndDates = datetime(2014,1,1);
Rates = 0.042;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9589
            Rates: 0.0420
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
```

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

Define the StockSpec.

```
AssetPrice = 50;
Sigma = 0.36;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3600
         AssetPrice: 50
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the floating lookback options.

```
Settle   = datetime(2013,1,1);
Maturity = datetime(2013,4,1);
OptSpec = {'put';'call'};
Strike = NaN;
```

Compute the price of the European floating lookback options.

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

```
Price = 2×1

    7.2581
    6.9777
```

Compute the Price of a Fixed Lookback Option Using the Conze-Viswanathan Model

Define the RateSpec.

```
StartDates = 'Jan-1-2013';
EndDates = 'Jan-1-2014';
Rates = 0.045;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9560
            Rates: 0.0450
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
```

```
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec`.

```
AssetPrice = 102;
Sigma = 0.45;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.4500
         AssetPrice: 102
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the fixed lookback options.

```
Settle   = 'Jan-1-2013';
Maturity = 'July-1-2013';
OptSpec = {'put';'call'};
Strike = [98;101];
```

Price the European fixed lookback options.

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

```
Price = 2×1

   18.3130
   30.4021
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
integer | vector of integers

Option strike price values, specified as an integer using a `NINST`-by-`1` vector of strike price values.

Data Types: `single` | `double`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the lookback option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `lookbackbycvgsg` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — European option expiry date
datetime array | string array | date character vector

European option expiry date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `lookbackbycvgsg` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = lookbackbycvgsg(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'AssetMinMax',AssetMinMax)`

`AssetMinMax` — Maximum or minimum underlying asset price
if unspecified, the lookback option is newly issued, and `AssetMinMax = StockSpec.AssetPrice` (default) | nonnegative integer

Maximum or minimum underlying asset price, specified as the comma-separated pair consisting of `'AssetMinMax'` and a `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected prices of lookback option

vector

Expected prices of the lookback option, returned as a `NINST`-by-`1` vector.

More About

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 Instruments 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. For more information, see “Lookback Option” on page 3-39.

Version History

Introduced in R2014a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `lookbackbycvgsg` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. C. *Options, Futures, and Other Derivatives* 5th Edition. Englewood Cliffs, NJ: Prentice Hall, 2002.

See Also

`lookbacksensbycvgsg` | `lookbackbyls` | `lookbacksensbyls` | `stockspec` | `intenvset` | `Lookback`

Topics

“Lookback Option” on page 3-39
“Supported Equity Derivative Functions” on page 3-19

“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

lookbacksensbycvgsg

Calculate prices or sensitivities of European lookback options using Conze-Viswanathan and Goldman-Sosin-Gatto models

Syntax

```
PriceSens = lookbacksensbycvgsg(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates)
PriceSens = lookbacksensbycvgsg( ___ ,Name,Value)
```

Description

`PriceSens = lookbacksensbycvgsg(RateSpec,StockSpec,OptSpec,Strike,Settle, ExerciseDates)` returns prices or sensitivities of European lookback options using Conze-Viswanathan and Goldman-Sosin-Gatto models. `lookbacksensbycvgsg` calculates prices of European fixed- and floating-strike lookback options. To compute the value of a floating-strike lookback option, `Strike` must be specified as `NaN`. The Goldman-Sosin-Gatto model is used for floating-strike lookback options. The Conze-Viswanathan model is used for fixed-strike lookback options.

Note Alternatively, you can use the `Lookback` object to calculate price or sensitivities for lookback options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = lookbacksensbycvgsg( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price and Delta of a Floating Lookback Option Using the Goldman-Sosin-Gatto Model

Define the `RateSpec`.

```
StartDates = datetime(2013,1,1);
EndDates = datetime(2014,1,1);
Rates = 0.41;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
 'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.6637
            Rates: 0.4100
         EndTimes: 1
       StartTimes: 0
```

```
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec` with continuous dividend yield.

```
AssetPrice = 120;
Sigma = 0.3;
Yield = 0.045;
StockSpec = stockspec(Sigma, AssetPrice, 'Continuous', Yield)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3000
         AssetPrice: 120
       DividendType: {'continuous'}
    DividendAmounts: 0.0450
    ExDividendDates: []
```

Define the floating lookback option.

```
Settle   = datetime(2013,1,1);
Maturity = datetime(2013,7,1);
OptSpec = 'call';
Strike = NaN;
SMinMax = 100;
```

Compute the price and delta of the European floating lookback option.

```
OutSpec = {'price', 'delta'};
[Price, Delta] = lookbacksensbycvgsg(RateSpec, StockSpec, OptSpec, Strike,...
Settle, Maturity,'AssetMinMax', SMinMax, 'OutSpec', OutSpec)
```

```
Price = 36.9926
```

```
Delta = 0.8659
```

Compute the Price and Delta of a Fixed Lookback Option Using the Conze-Viswanathan Model

Define the `RateSpec`.

```
StartDates = datetime(2013,1,1);
EndDates = datetime(2015,1,1);
Rates = 0.1;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
```

```
             Disc: 0.8187
            Rates: 0.1000
          EndTimes: 2
        StartTimes: 0
          EndDates: 735965
        StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec`.

```
AssetPrice = 103;
Sigma = 0.30;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3000
          AssetPrice: 103
        DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the fixed lookback option.

```
Settle   = datetime(2013,1,1);
Maturity = datetime(2013,7,1);
OptSpec = 'call';
Strike = 99;
```

Price and delta for the European fixed lookback option.

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

```
Price = 22.7227
```

```
Delta = 1.1349
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
integer | vector of integers

Option strike price values, specified as an integer using a `NINST`-by-`1` vector of strike price values.

Data Types: `double`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the lookback option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `lookbacksensbycvgsg` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — European option expiry date
datetime array | string array | date character vector

European option expiry date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `lookbacksensbycvgsg` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = lookbacksensbycvgsg(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'AssetMinMax',AssetMinMax,'OutSpec',{'All'})`

`AssetMinMax` — Maximum or minimum underlying asset price
if unspecified, the lookback option is newly issued, and `AssetMinMax = StockSpec.AssetPrice` (default) | nonnegative integer

Maximum or minimum underlying asset price, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity.

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities of lookback option
vector

Expected prices or sensitivities (defined by `OutSpec`) of the lookback option, returned as a `NINST`-by-`1` vector.

More About

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 Instruments 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. For more information, see “Lookback Option” on page 3-39.

Version History

Introduced in R2014a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `lookbacksensbycvgsg` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. C. *Options, Futures, and Other Derivatives* 5th Edition. Englewood Cliffs, NJ, Prentice Hall, 2002.

See Also

`lookbackbycvgsg` | `lookbackbyls` | `lookbacksensbyls` | `stockspec` | `intenvset` | `Lookback`

Topics

“Lookback Option” on page 3-39
“Supported Equity Derivative Functions” on page 3-19

lookbackbyeqp

Price lookback option from Equal Probabilities binomial tree

Syntax

```
Price = lookbackbyeqp(EQPTree,OptSpec,Strike,Settle,ExerciseDates)
Price = lookbackbyeqp( ___ ,AmericanOpt)
```

Description

`Price = lookbackbyeqp(EQPTree,OptSpec,Strike,Settle,ExerciseDates)` prices lookback options using an Equal Probabilities binomial tree.

Note Alternatively, you can use the `Lookback` object to price lookback options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = lookbackbyeqp( ___ ,AmericanOpt)` adds an optional argument for `AmericanOpt`.

Examples

Price a Lookback Option Using an EQP Equity Tree

This example shows how to price a lookback option using an EQP equity tree by loading 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

OptSpec = 'Call';
Strike = 115;
Settle = datetime(2003,1,1);
ExerciseDates = datetime(2006,1,1);

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

```
Price = 8.7941
```

Input Arguments

EQPTree — Stock tree structure

structure

Stock tree structure for an Equal Probabilities binomial tree, specified by using `eqptree`.

Data Types: `struct`

OptSpec — Definition of option
character vector with value 'call' or 'put' | cell array of character vectors with values 'call' or 'put'

Definition of option, specified as 'call' or 'put' using a character vector or a NINST-by-1 cell array of character vectors for 'call' or 'put'.

Data Types: char | cell

Strike — Option strike price value
matrix of nonnegative integers

Option strike price value, specified with a nonnegative integer using a NINST-by-1 matrix of strike price values. Each row is the schedule for one option.

To compute the value of a floating-strike lookback option, Strike must be specified as NaN. Floating-strike lookback options are also known as average strike options.

Data Types: double

Settle — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date for the lookback option, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

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

To support existing code, lookbackbyeqp also accepts serial date numbers as inputs, but they are not recommended.

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use aNINST-by-1 matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one ExerciseDates on the option expiry date.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. 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 a NINST-by-1 cell array of character vectors, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

To support existing code, lookbackbyeqp also accepts serial date numbers as inputs, but they are not recommended.

AmericanOpt — Option type
0 European (default) | integer with values 0 or 1

(Optional) Option type, specified as NINST-by-1 integer flags with values:

- 0 — European

- `1` — American

Data Types: `double`

Output Arguments

`Price` — Expected prices for lookback options at time `0`

vector

Expected prices for lookback options at time 0, returned as a `NINST`-by-`1` vector. Pricing of lookback options is done using Hull-White (1993). Therefore, for these options there are no unique prices on the tree nodes except for the root node.

More About

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 Instruments 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. For more information, see “Lookback Option” on page 3-39.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `lookbackbyeqp` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] 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` | `Lookback`

Topics

“Computing Prices Using EQP” on page 3-66
“Examining Output from the Pricing Functions” on page 3-70
“Computing Equity Instrument Sensitivities” on page 3-75
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Lookback Option” on page 3-39
“Computing Instrument Prices” on page 3-64
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

lookbackbyitt

Price lookback option using implied trinomial tree (ITT)

Syntax

```
Price = lookbackbyitt(ITTTree,OptSpec,Strike,Settle,ExerciseDates)
Price = lookbackbyitt( ___ ,AmericanOpt)
```

Description

`Price = lookbackbyitt(ITTTree,OptSpec,Strike,Settle,ExerciseDates)` prices lookback options using an implied trinomial tree (ITT).

`Price = lookbackbyitt( ___ ,AmericanOpt)` adds an optional argument for `AmericanOpt`.

Examples

Price a Lookback Option Using an ITT Equity Tree

This example shows how to price a lookback option using an ITT equity tree by loading 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

OptSpec = 'Call';
Strike = 85;
Settle = datetime(2006,1,1);
ExerciseDates = datetime(2008,1,1);

Price = lookbackbyitt(ITTTree, OptSpec, Strike, Settle, ExerciseDates)
```

```
Price = 0.5426
```

Input Arguments

`ITTTree` — Stock tree structure

structure

Stock tree structure for an implied trinomial tree (ITT), specified by using `itttree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a character vector or a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: char | cell

Strike — Option strike price value
matrix of nonnegative integers

Option strike price value, specified with a nonnegative integer using a NINST-by-1 matrix of strike price values. Each row is the schedule for one option.

To compute the value of a floating-strike lookback option, Strike must be specified as NaN. Floating-strike lookback options are also known as average strike options.

Data Types: double

Settle — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date for the lookback option, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

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

To support existing code, lookbackbyitt also accepts serial date numbers as inputs, but they are not recommended.

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use aNINST-by-1 matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one ExerciseDates on the option expiry date.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. 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 a NINST-by-1 cell array of character vectors, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

To support existing code, lookbackbyitt also accepts serial date numbers as inputs, but they are not recommended.

AmericanOpt — Option type
0 European (default) | integer with values 0 or 1

(Optional) Option type, specified as NINST-by-1 integer flags with values:

- 0 — European
- 1 — American

Data Types: double

Output Arguments

`Price` — Expected prices for lookback options at time `0`

vector

Expected prices for lookback options at time 0, returned as a `NINST`-by-`1` vector. Pricing of lookback options is done using Hull-White (1993). Therefore, for these options there are no unique prices on the tree nodes except for the root node.

More About

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 Instruments 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. For more information, see “Lookback Option” on page 3-39.

Version History

Introduced in R2007a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `lookbackbyitt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] 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

`itttree` | `instlookback`

Topics

“Computing Prices Using ITT” on page 3-68

“Examining Output from the Pricing Functions” on page 3-70
“Computing Equity Instrument Sensitivities” on page 3-75
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Lookback Option” on page 3-39
“Computing Instrument Prices” on page 3-64
“Supported Equity Derivative Functions” on page 3-19

lookbackbyls

Price European or American lookback options using Monte Carlo simulations

Syntax

```
[Price,Paths,Times,Z] = lookbackbyls(RateSpec,StockSpec,OptSpec,Strike,
Settle,ExerciseDates)
[Price,Paths,Times,Z] = lookbackbyls( ___ ,Name,Value)
```

Description

`[Price,Paths,Times,Z] = lookbackbyls(RateSpec,StockSpec,OptSpec,Strike, Settle,ExerciseDates)` returns prices of lookback options using the Longstaff-Schwartz model for Monte Carlo simulations. `lookbackbyls` computes prices of European and American lookback options.

For American options, the Longstaff-Schwartz least squares method calculates the early exercise premium.

`lookbackbyls` calculates values of fixed- and floating-strike lookback options. To compute the value of a floating-strike lookback option, `Strike` must be specified as `NaN`.

Note Alternatively, you can use the `Lookback` object to price lookback options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,Paths,Times,Z] = lookbackbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price for a Floating Lookback Option Using Monte Carlo Simulation

Define the `RateSpec`.

```
StartDates = datetime(2013,1,1);
EndDates = datetime(2014,1,1);
Rates = 0.042;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates, 'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9589
            Rates: 0.0420
         EndTimes: 1
```

```
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the StockSpec.

```
AssetPrice = 50;
Sigma = 0.36;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3600
         AssetPrice: 50
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the floating lookback option.

```
Settle   = datetime(2013,1,1);
Maturity = datetime(2013,4,1);
OptSpec = 'put';
Strike = NaN;
```

Compute the price of the European floating lookback option.

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

```
Price = 6.6471
```

Compute the Price of a Fixed Lookback Option Using Monte Carlo Simulation

Define the RateSpec.

```
StartDates = 'Jan-1-2013';
EndDates = 'Jan-1-2014';
Rates = 0.045;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9560
            Rates: 0.0450
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
```

```
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec`.

```
AssetPrice = 102;
Sigma = 0.45;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.4500
         AssetPrice: 102
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the fixed lookback option.

```
Settle   = 'Jan-1-2013';
Maturity = 'July-1-2013';
OptSpec = 'call';
Strike = 98;
```

Compute the price of the European fixed lookback option.

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

```
Price = 30.2368
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
integer | vector of integers

Option strike price values, specified as an integer using a `NINST`-by-`1` vector of strike price values.

Data Types: `single` | `double`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the lookback option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `lookbackbyls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Exercise callable or puttable dates for European or American options
datetime array | string array | date character vector

Exercise callable or puttable dates for European or American options, specified as a datetime array, string array, or date character vectors as follows:

- European option — `NINST`-by-`1` vector of exercise dates. For a European option, there is only one exercise date which is the option expiry date.
- American option — `NINST`-by-`2` vector of exercise date boundaries. For each instrument, the option is 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 a `NINST`-by-`1` vector of dates, the option is exercised between `Settle` and the single listed exercise date.

To support existing code, `lookbackbyls` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = lookbackbyls(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr,'AmericanOpt',1)`

`AmericanOpt` — Option type
`0` European (default) | scalar with value `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and an integer scalar flag with these values:

- `0` — European

- 1 — American

Note For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `double`

`NumTrials` — Scalar number of independent sample paths
`1000` (default) | nonnegative scalar integer

Scalar number of independent sample paths (simulation trials), specified as the comma-separated pair consisting of `'NumTrials'` and a nonnegative integer.

Data Types: `double`

`NumPeriods` — Scalar number of simulation periods per trial
`100` (default) | nonnegative scalar integer

Scalar number of simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` and a nonnegative integer. `NumPeriods` is considered only when pricing European lookback options. For American lookback options, `NumPeriods` is equal to the number of exercise days during the life of the option.

Data Types: `double`

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-1-by-`NumTrials` 3-D array. The `Z` value generates the Brownian motion vector (that is, Wiener processes) that drives the simulation.

Data Types: `double`

`Antithetic` — Indicator for antithetic sampling
`false` (default) | scalar logical flag with value of `true` or `false`

Indicator for antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a value of `true` or `false`.

Data Types: `logical`

Output Arguments

`Price` — Expected price of lookback option
scalar

Expected price of the lookback option, returned as a 1-by-1 scalar.

`Paths` — Simulated paths of correlated state variables
vector

Simulated paths of correlated state variables, returned as a `NumPeriods + 1`-by-`1`-by-`NumTrials` 3-D time series array. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths
vector

Observation times associated with simulated paths, returned as a `NumPeriods + 1`-by-`1` column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, returned as a `NumPeriods`-by-`1`-by-`NumTrials` 3-D array when `Z` is specified as an input argument. If the `Z` input argument is not specified, then the `Z` output argument contains the random variates generated internally.

More About

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 Instruments 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. For more information, see “Lookback Option” on page 3-39.

Version History

Introduced in R2014a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `lookbackbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. C. *Options, Futures, and Other Derivatives* 5th Edition. Englewood Cliffs, NJ: Prentice Hall, 2002.

See Also

`lookbackbycvgsg` | `lookbacksensbycvgsg` | `lookbacksensbyls` | `stockspec` | `intenvset` | `Lookback`

Topics

“Lookback Option” on page 3-39
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

lookbacksensbyls

Calculate price and sensitivities for European or American lookback options using Monte Carlo simulations

Syntax

```
[PriceSens,Paths,Times,Z] = lookbacksensbyls(RateSpec,StockSpec,OptSpec,
Strike,Settle,ExerciseDates)
[PriceSens,Paths,Times,Z] = lookbacksensbyls( ___ ,Name,Value)
```

Description

`[PriceSens,Paths,Times,Z] = lookbacksensbyls(RateSpec,StockSpec,OptSpec, Strike,Settle,ExerciseDates)` returns prices or sensitivities of lookback options using the Longstaff-Schwartz model for Monte Carlo simulations. `lookbacksensbyls` computes prices of European and American lookback options.

For American options, the Longstaff-Schwartz least squares method calculates the early exercise premium.

`lookbacksensbyls` calculates values of fixed- and floating-strike lookback options. To compute the value of a floating-strike lookback option, `Strike` must be specified as `NaN`.

Note Alternatively, you can use the `Lookback` object to calculate price or sensitivities for lookback options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,Paths,Times,Z] = lookbacksensbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price and Delta of a European Floating Lookback Option Using Monte Carlo Simulation

Define the `RateSpec`.

```
StartDates = datetime(2013,1,1);
EndDates = datetime(2014,1,1);
Rates = 0.41;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.6637
```

```
           Rates: 0.4100
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
           Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec` with continuous dividend yield.

```
AssetPrice = 120;
Sigma = 0.3;
Yield = 0.045;
StockSpec = stockspec(Sigma, AssetPrice, 'Continuous', Yield)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3000
         AssetPrice: 120
       DividendType: {'continuous'}
    DividendAmounts: 0.0450
    ExDividendDates: []
```

Define the floating lookback option.

```
Settle   = datetime(2013,1,1);
Maturity = datetime(2013,7,1);
OptSpec = 'call';
Strike = NaN;
```

Compute the price and delta of the European floating lookback option.

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

```
Price = 27.0768
```

```
Delta = 0.2256
```

Compute the Price and Delta of a European Fixed Lookback Option Using Monte Carlo Simulation

Define the `RateSpec`.

```
StartDates = datetime(2013,1,1);
EndDates = datetime(2015,1,1);
Rates = 0.1;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
```

```
      Compounding: -1
             Disc: 0.8187
            Rates: 0.1000
          EndTimes: 2
        StartTimes: 0
          EndDates: 735965
        StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec`.

```
AssetPrice = 103;
Sigma = 0.30;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.3000
          AssetPrice: 103
        DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the fixed lookback option.

```
Settle   = datetime(2013,1,1);
Maturity = datetime(2013,7,1);
OptSpec = 'call';
Strike = 99;
```

Compute the price and delta of the European fixed lookback option.

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

```
Price = 22.7227
```

```
Delta = 1.1349
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` can handle several types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
integer | vector of integers

Option strike price values, specified as an integer using a `NINST`-by-`1` vector of strike price values.

Data Types: `single` | `double`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the lookback option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `lookbacksensbyls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Matrix of exercise callable or puttable dates for European or American options
datetime array | string array | date character vector

Matrix of exercise callable or puttable dates for European or American options, specified as a datetime array, string array, or date character vectors as follows:

- European option — `NINST`-by-`1` vector of exercise dates. For a European option, there is only one exercise date which is the option expiry date.
- American option — `NINST`-by-`2` vector of exercise date boundaries. For each instrument, the option is 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 a `NINST`-by-`1` vector of dates, the option is exercised between `Settle` and the single listed exercise date.

To support existing code, `lookbacksensbyls` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = lookbacksensbyls(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr,'AmericanOpt',1,'OutSpec',{'All'})`

`AmericanOpt` — Option type
`0` European (default) | scalar with value `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and an integer scalar flag with these values:

- `0` — European
- `1` — American

Note For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `double`

`NumTrials` — Scalar number of independent sample paths
`1000` (default) | nonnegative scalar integer

Scalar number of independent sample paths (simulation trials), specified as the comma-separated pair consisting of `'NumTrials'` and a nonnegative integer.

Data Types: `double`

`NumPeriods` — Scalar number of simulation periods per trial
`100` (default) | nonnegative scalar integer

Scalar number of simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` and a nonnegative integer. `NumPeriods` is considered only when pricing European lookback options. For American lookback options, `NumPeriod` is equal to the number of exercise days during the life of the option.

Data Types: `double`

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-`1`-by-`NumTrials` 3-D array. The `Z` value generates the Brownian motion vector (that is, Wiener processes) that drives the simulation.

Data Types: `double`

`Antithetic` — Indicator for antithetic sampling
`false` (default) | scalar logical flag with value `true` or `false`

Indicator for antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a value of `true` or `false`.

Data Types: `logical`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity.

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected price or sensitivities of lookback option
scalar

Expected price or sensitivities (defined by `OutSpec`) of the lookback option, returned as a `1`-by-`1` array.

`Paths` — Simulated paths of correlated state variables
vector

Simulated paths of correlated state variables, returned as a `NumPeriods + 1`-by-`1`-by-`NumTrials` 3-D time series array. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths
vector

Observation times associated with simulated paths, returned as a `NumPeriods + 1`-by-`1` column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, returned as a `NumPeriods`-by-`1`-by-`NumTrials` 3-D array when `Z` is specified as an input argument. If the `Z` input argument is not specified, then the `Z` output argument contains the random variates generated internally.

More About

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 Instruments 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. For more information, see “Lookback Option” on page 3-39.

Version History

Introduced in R2014a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `lookbacksensbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. C. *Options, Futures, and Other Derivatives* 5th Edition. Englewood Cliffs, NJ: Prentice Hall, 2002.

See Also

`lookbackbycvgsg` | `lookbacksensbycvgsg` | `lookbackbyls` | `intenvset` | `stockspec` | `Lookback`

Topics

“Lookback Option” on page 3-39
“Supported Equity Derivative Functions” on page 3-19

lookbackbystt

Price lookback options using standard trinomial tree

Syntax

```
Price = lookbackbystt(STTTree,OptSpec,Strike,Settle,ExerciseDates)
Price = lookbackbystt( ___ ,AmericanOpt)
```

Description

`Price = lookbackbystt(STTTree,OptSpec,Strike,Settle,ExerciseDates)` prices lookback options using a standard trinomial (STT) tree.

Note Alternatively, you can use the `Lookback` object to price lookback options. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`Price = lookbackbystt( ___ ,AmericanOpt)` prices lookback options using a standard trinomial (STT) tree with an optional argument for `AmericanOpt`.

Examples

Price a Lookback Option Using the Standard Trinomial Tree Model

Create a `RateSpec`.

```
StartDates = datetime(2009,1,1);
EndDates = datetime(2013,1,1);
Rates = 0.035;
Basis = 1;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8694
            Rates: 0.0350
         EndTimes: 4
       StartTimes: 0
         EndDates: 735235
       StartDates: 733774
    ValuationDate: 733774
            Basis: 1
     EndMonthRule: 1
```

Create a `StockSpec`.

```
AssetPrice = 85;
Sigma = 0.15;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1500
          AssetPrice: 85
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Create an STTTree.

```
NumPeriods = 4;
TimeSpec = stttimespec(StartDates, EndDates, 4);
STTTree = stttree(StockSpec, RateSpec, TimeSpec)
```

```
STTTree = struct with fields:
       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 1 2 3 4]
         dObs: [733774 734139 734504 734869 735235]
        STree: {1x5 cell}
        Probs: {[3x1 double]  [3x3 double]  [3x5 double]  [3x7 double]}
```

Define the lookback option and compute the price.

```
Settle = datetime(2009,1,1);
ExerciseDates = [datetime(2012,1,1) ; datetime(2013,1,1)];
OptSpec = 'call';
Strike = [90;95];

Price= lookbackbystt(STTTree, OptSpec, Strike, Settle, ExerciseDates)
```

```
Price = 2×1

   11.7296
   12.9120
```

Input Arguments

STTTree — Stock tree structure for standard trinomial tree

structure

Stock tree structure for a standard trinomial tree, specified by using `stttree`.

Data Types: `struct`

OptSpec — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a character vector or a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price value
matrix of nonnegative integers

Option strike price value, specified with a nonnegative integer using a `NINST`-by-`1` matrix of strike price values. Each row is the schedule for one option. To compute the value of a floating-strike lookback option, `Strike` should be specified as `NaN`. Floating-strike lookback options are also known as average strike options.

Data Types: `double`

`Settle` — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date for the lookback option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `lookbackbystt` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a`NINST`-by-`1` matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. 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 a `NINST`-by-`1` cell array of character vectors, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `lookbackbystt` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
`0` European (default) | scalar with values `[0,1]`

Option type, specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

Data Types: `double`

Output Arguments

`Price` — Expected prices for lookback options at time `0`

matrix

Expected prices for lookback options at time 0, returned as a `NINST`-by-`1` matrix. Pricing of lookback options is done using Hull-White (1993). Consequently, for these options there are no unique prices on the tree nodes with the exception of the root node.

More About

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 Instruments 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. For more information, see “Lookback Option” on page 3-39.

Version History

Introduced in R2015b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `lookbackbystt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] 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

`stttimespec` | `stttree` | `sttprice` | `sttsens` | `Lookback`

Topics

“Lookback Option” on page 3-39

“Supported Equity Derivative Functions” on page 3-19

lrtimespec

Specify time structure for Leisen-Reimer binomial tree

Syntax

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

Description

`TimeSpec = lrtimespec(ValuationDate,Maturity,NumPeriods)` specifies a time structure for a Leisen-Reimer stock tree (`lrtree`).

Examples

Specify the Time Structure for Leisen-Reimer Binomial Tree

This example shows how to specify a 5-period tree with time steps of 1 year.

```
ValuationDate = datetime(2010,7,1);
Maturity = datetime(2015,7,1);
TimeSpec = lrtimespec(ValuationDate, Maturity, 5)
```

```
TimeSpec = struct with fields:
           FinObj: 'BinTimeSpec'
    ValuationDate: 734320
         Maturity: 736146
       NumPeriods: 5
            Basis: 0
     EndMonthRule: 1
             tObs: [0 1 2 3 4 5]
             dObs: [734320 734685 735050 735415 735780 736146]
```

Input Arguments

`ValuationDate` — Pricing date and first observation in Leisen-Reimer stock tree
datetime scalar | string scalar | date character vector

Pricing date and first observation in the `lrtree`, specified as a scalar datetime, string, or data character vector.

To support existing code, `lrtimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Date marking the depth of the Leisen-Reimer stock tree
date character vector

Date marking the depth of the Leisen-Reimer stock tree, specified as scalar datetime, string, or data character vector.

To support existing code, `lrtimespec` also accepts serial date numbers as inputs, but they are not recommended.

`NumPeriods` — Number of time steps in the Leisen-Reimer stock tree
odd integer value

Number of time steps in the Leisen-Reimer stock tree, specified as scalar odd integer value.

Note Leisen-Reimer requires the number of steps to be an odd number.

Data Types: `double`

Output Arguments

`TimeSpec` — Specification for the time layout for `lrtree`
structure

Specification for the time layout for `lrtree`, returned as a structure.

Version History

Introduced in R2010b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `lrtimespec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Leisen D.P., M. Reimer. “Binomial Models for Option Valuation – Examining and Improving Convergence.” *Applied Mathematical Finance.* Number 3, 1996, pp. 319–346.

See Also

`stockspec` | `lrtree`

Topics
“Building Equity Binary Trees” on page 3-3

“Understanding Equity Trees” on page 3-2
“Supported Equity Derivative Functions” on page 3-19

lrtree

Build Leisen-Reimer stock tree

Syntax

```
LRTree = lrtree(StockSpec,RateSpec,TimeSpec,Strike)
LRTree = lrtree( ___ ,Name,Value)
```

Description

`LRTree = lrtree(StockSpec,RateSpec,TimeSpec,Strike)` builds a Leisen-Reimer stock tree.

`LRTree = lrtree( ___ ,Name,Value)` adds a name-value pair argument.

Examples

Build a Leisen-Reimer Stock Tree

This example shows how to build Leisen-Reimer stock tree. Consider a European put option with an exercise price of $30 that expires on June 1, 2010. The underlying stock is trading at $30 on January 1, 2010 and has a volatility of 30% per annum. The annualized continuously compounded risk-free rate is 5% per annum. Using this data, create a Leisen-Reimer tree with 101 steps using the `PP1` method.

```
AssetPrice = 30;
Strike = 30;

ValuationDate = 'Jan-1-2010';
Maturity = 'June-1-2010';

% define StockSpec
Sigma = 0.3;
StockSpec = stockspec(Sigma, AssetPrice);

% define RateSpec
Rates = 0.05;
Settle = ValuationDate;
Basis = 1;
Compounding = -1;

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

% build the Leisen-Reimer (LR) tree with 101 steps
LRTimeSpec = lrtimespec(ValuationDate, Maturity, 101);

% use the PP1 method
LRMethod  = 'PP1';

LRTree = lrtree(StockSpec, RateSpec, LRTimeSpec, Strike, ...
'method', LRMethod)
```

```
LRTree = struct with fields:
       FinObj: 'BinStockTree'
       Method: 'LR'
    Submethod: 'PP1'
       Strike: 30
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.0041 0.0082 0.0123 0.0164 0.0205 0.0246 0.0288 ... ]
         dObs: [734139 734140 734141 734143 734144 734146 734147 734149 ... ]
        STree: {1x102 cell}
      UpProbs: [101x1 double]
```

Input Arguments

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in `StockSpec`, dividends are assumed to be `0`.

Data Types: `struct`

`RateSpec` — Interest-rate specification for initial risk-free rate curve

structure

Interest-rate specification for initial rate curve, specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`TimeSpec` — Time tree layout specification

structure

Time tree layout specification, specified using the `TimeSpec` output obtained from `lrtimespec`.

Data Types: `struct`

`Strike` — Option strike price value

nonnegative integer

Option strike price value, specified as a scalar nonnegative integer.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `LRTree = lrtree(StockSpec,RateSpec,LRTimeSpec,Strike,'Method','PP2')`

`Method` — Computation method
`'PP1'` (default) | character vector with value `'PP1'` or `'PP2'`

Computation method, specified as the comma-separated pair consisting of `'Method'` and a character vector with a value of `'PP1'` or `'PP2'`. `'PP1'` is for Peizer-Pratt method 1 inversion and `'PP2'` is for Peizer-Pratt method 2 inversion. For more information on `'PP1'` and `'PP2'` methods, see “Leisen-Reimer Tree (LR) Modeling” on page B-3.

Data Types: `char`

Output Arguments

`LRTree` — Stock and time information for a Leisen-Reimer tree
structure

Stock and time information for a Leisen-Reimer tree, returned as a structure.

Version History

Introduced in R2010b

References

[1] Leisen D.P., M. Reimer. “Binomial Models for Option Valuation – Examining and Improving Convergence.” *Applied Mathematical Finance.* Number 3, 1996, pp. 319–346.

See Also

`stockspec` | `lrtimespec` | `intenvset` | `optstockbylr` | `optstocksensbylr`

Topics
“Building Equity Binary Trees” on page 3-3
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Understanding Equity Trees” on page 3-2
“Supported Equity Derivative Functions” on page 3-19

maxassetbystulz

Determine 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)
```

Description

`Price = maxassetbystulz(RateSpec,StockSpec1,StockSpec2,Settle,Maturity, OptSpec,Strike,Corr)` computes option prices using the Stulz option pricing model.

Examples

Compute Rainbow Option Prices Using the Stulz Option Pricing Model

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). First, 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)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
```

```
             Disc: 0.9512
            Rates: 0.0500
          EndTimes: 1
        StartTimes: 0
          EndDates: 733788
        StartDates: 733422
    ValuationDate: 733422
            Basis: 1
     EndMonthRule: 1
```

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)
```

```
StockSpec1 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2200
         AssetPrice: 950
       DividendType: {'continuous'}
    DividendAmounts: 0.0200
    ExDividendDates: []
```

```
StockSpec2 = stockspec(Sigma2, AssetPrice2, 'continuous', Div2)
```

```
StockSpec2 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1500
         AssetPrice: 980
       DividendType: {'continuous'}
    DividendAmounts: 0.0600
    ExDividendDates: []
```

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 = 3×1

  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).

Input Arguments

`RateSpec` — Annualized, continuously compounded rate term structure
structure

Annualized, continuously compounded rate term structure, specified using `intenvset`.

Data Types: `structure`

`StockSpec1` — Stock specification for asset 1
structure

Stock specification for asset 1, specified using `stockspec`.

Data Types: `structure`

`StockSpec2` — Stock specification for asset 2
structure

Stock specification for asset 2, specified using `stockspec`.

Data Types: `structure`

`Settle` — Settlement or trade dates
vector

Settlement or trade dates, specified as an `NINST`-by-`1` vector of numeric dates.

Data Types: `double`

`Maturity` — Maturity dates
vector

Maturity dates, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`OptSpec` — Option type
cell array of character vectors with a value of `'call'` or `'put'`

Option type, specified as an `NINST`-by-`1` cell array of character vectors with a value of `'call'` or `'put'`.

Data Types: `cell`

`Strike` — Strike prices
vector

Strike prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Corr` — Correlation between the underlying asset prices
vector

Correlation between the underlying asset prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected option prices

vector

Expected option prices, returned as an `NINST`-by-`1` vector.

More About

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 Instruments 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.

For more information, see “Rainbow Option” on page 3-27.

Version History

Introduced in R2009a

See Also

`intenvset` | `maxassetsensbystulz` | `minassetbystulz` | `stockspec`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Rainbow Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19

maxassetsensbystulz

Determine European rainbow option prices or sensitivities on maximum of two risky assets using Stulz pricing model

Syntax

```
PriceSens = maxassetsensbystulz(RateSpec,StockSpec1,StockSpec2,Settle,
Maturity,OptSpec,Strike,Corr)
PriceSens = maxassetsensbystulz( ___ ,Name,Value)
```

Description

`PriceSens = maxassetsensbystulz(RateSpec,StockSpec1,StockSpec2,Settle, Maturity,OptSpec,Strike,Corr)` computes option prices using the Stulz option pricing model.

`PriceSens = maxassetsensbystulz( ___ ,Name,Value)` specifies options using one or more optional name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute Rainbow Option Prices and Sensitivities Using the Stulz Option Pricing Model

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), 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). First, 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)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0500
         EndTimes: 1
       StartTimes: 0
         EndDates: 733788
       StartDates: 733422
    ValuationDate: 733422
            Basis: 1
     EndMonthRule: 1
```

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)
```

```
StockSpec1 = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.2200
          AssetPrice: 950
        DividendType: {'continuous'}
    DividendAmounts: 0.0200
    ExDividendDates: []
```

```
StockSpec2 = stockspec(Sigma2, AssetPrice2, 'continuous', Div2)
```

```
StockSpec2 = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.1500
          AssetPrice: 980
        DividendType: {'continuous'}
    DividendAmounts: 0.0600
    ExDividendDates: []
```

Calculate the price and delta for different correlation levels.

```
Strike = 1000 ;
Corr = [-0.5; 0; 0.5];
OutSpec = {'price'; 'delta'};
OptSpec = 'call';
[Price, Delta] = maxassetsensbystulz(RateSpec, StockSpec1, StockSpec2,...
Settle, Maturity, OptSpec, Strike, Corr,'OutSpec', OutSpec)
```

```
Price = 3×1
```

```
  111.6683
  103.7715
   92.4412


Delta = 3×2

    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).

Input Arguments

`RateSpec` — Annualized, continuously compounded rate term structure
structure

Annualized, continuously compounded rate term structure, specified using `intenvset`.

Data Types: `structure`

`StockSpec1` — Stock specification for asset 1
structure

Stock specification for asset 1, specified using `stockspec`.

Data Types: `structure`

`StockSpec2` — Stock specification for asset 2
structure

Stock specification for asset 2, specified using `stockspec`.

Data Types: `structure`

`Settle` — Settlement or trade dates
vector

Settlement or trade dates, specified as an `NINST`-by-`1` vector of numeric dates.

Data Types: `double`

`Maturity` — Maturity dates
vector

Maturity dates, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`OptSpec` — Option type
cell array of character vectors with a value of `'call'` or `'put'`

Option type, specified as an `NINST`-by-`1` cell array of character vectors with a value of `'call'` or `'put'`.

Data Types: `cell`

`Strike` — Strike prices
vector

Strike prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Corr` — Correlation between the underlying asset prices
vector

Correlation between the underlying asset prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PriceSens] = maxassetsensbystulz(RateSpec, StockSpecA,StockSpecB,Settle,Maturity,OptSpec,Strike,Corr,'OutSpec',OutSpec)`

`OutSpec` — Define outputs
`{'Price'}` (default) | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors or string array with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`OutSpec = {'All'}` specifies that the output is `Delta`, `Gamma`, `Vega`, `Lambda`, `Rho`, `Theta`, and `Price`, in that order. This is the same as specifying `OutSpec` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities
vector

Expected prices or sensitivities, returned as an `NINST`-by-`1` or `NINST`-by-`2` vector.

More About

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 Instruments 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.

For more information, see “Rainbow Option” on page 3-27.

Version History

Introduced in R2009a

See Also

`intenvset` | `maxassetsensbystulz` | `stockspec`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Rainbow Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19

minassetbystulz

Determine 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)
```

Description

`Price = minassetbystulz(RateSpec,StockSpec1,StockSpec2,Settle,Maturity, OptSpec,Strike,Corr)` computes option prices using the Stulz option pricing model.

Examples

Compute Rainbow Option Prices Using the Stulz Option Pricing Model

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. First, create the `RateSpec`:

```
Settle = datetime(2000,11,15);
Maturity = datetime(2009,5,15);
Rates = 0.045;
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.6822
            Rates: 0.0450
         EndTimes: 8.5000
       StartTimes: 0
         EndDates: 733908
       StartDates: 730805
    ValuationDate: 730805
            Basis: 1
     EndMonthRule: 1
```

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)
```

```
StockSpecA = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1100
         AssetPrice: 49.7500
       DividendType: {'continuous'}
    DividendAmounts: 0.0450
    ExDividendDates: []
```

```
StockSpecB = stockspec(SigmaB, AssetPriceB, 'continuous', DivB)
```

```
StockSpecB = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1600
         AssetPrice: 51
       DividendType: {'continuous'}
    DividendAmounts: 0.0500
    ExDividendDates: []
```

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 = 3×1

   10.0002
    9.1433
    8.1622
```

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.

Input Arguments

RateSpec — Annualized, continuously compounded rate term structure

structure

Annualized, continuously compounded rate term structure, specified using `intenvset`.

Data Types: `structure`

StockSpec1 — Stock specification for asset 1

structure

Stock specification for asset 1, specified using `stockspec`.

Data Types: `structure`

`StockSpec2` — Stock specification for asset 2
structure

Stock specification for asset 2, specified using `stockspec`.

Data Types: `structure`

`Settle` — Settlement or trade dates
datetime array | string array | date character vector

Settlement or trade dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `minassetbystulz` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity dates
datetime array | string array | date character vector

Maturity dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `minassetbystulz` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Option type
cell array of character vectors with a value of `'call'` or `'put'`

Option type, specified as an `NINST`-by-`1` cell array of character vectors with a value of `'call'` or `'put'`.

Data Types: `cell`

`Strike` — Strike prices
vector

Strike prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Corr` — Correlation between the underlying asset prices
vector

Correlation between the underlying asset prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected option prices
vector

Expected option prices, returned as an `NINST`-by-`1` vector.

More About

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 Instruments 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.

For more information, see “Rainbow Option” on page 3-27.

Version History

Introduced in R2009a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `minassetbystulz` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`intenvset` | `maxassetsensbystulz` | `stockspec` | `minassetsensbystulz`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Rainbow Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19

minassetsensbystulz

Determine European rainbow option prices or sensitivities on minimum of two risky assets using Stulz option pricing model

Syntax

```
PriceSens = minassetsensbystulz(RateSpec,StockSpec1,StockSpec2,Settle,
Maturity,OptSpec,Strike,Corr)
PriceSens = minassetsensbystulz( ___ ,Name,Value)
```

Description

`PriceSens = minassetsensbystulz(RateSpec,StockSpec1,StockSpec2,Settle, Maturity,OptSpec,Strike,Corr)` computes option prices using the Stulz option pricing model.

`PriceSens = minassetsensbystulz( ___ ,Name,Value)` specifies options using one or more optional name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute Rainbow Option Prices and Sensitivities Using the Stulz Option Pricing Model

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. First, create the `RateSpec`:

```
Settle = datetime(2000,11,15);
Maturity = datetime(2009,5,15);
Rates = 0.045;
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.6822
            Rates: 0.0450
         EndTimes: 8.5000
       StartTimes: 0
         EndDates: 733908
       StartDates: 730805
    ValuationDate: 730805
            Basis: 1
     EndMonthRule: 1
```

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)
```

```
StockSpecA = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1100
         AssetPrice: 49.7500
       DividendType: {'continuous'}
    DividendAmounts: 0.0450
    ExDividendDates: []
```

```
StockSpecB = stockspec(SigmaB, AssetPriceB, 'continuous', DivB)
```

```
StockSpecB = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1600
         AssetPrice: 51
       DividendType: {'continuous'}
    DividendAmounts: 0.0500
    ExDividendDates: []
```

Calculate price and delta for different correlation levels.

```
Strike = 50.25;
Corr = [-0.5;0;0.5];
OptSpec = 'put';
OutSpec = {'Price'; 'delta'};
[P, D] = minassetsensbystulz(RateSpec, StockSpecA, StockSpecB,...
Settle, Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec)
```

```
P = 3×1

   10.0002
    9.1433
    8.1622


D = 3×2

   -0.2037   -0.2192
   -0.1774   -0.2101
   -0.1452   -0.2075
```

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.

Input Arguments

`RateSpec` — Annualized, continuously compounded rate term structure
structure

Annualized, continuously compounded rate term structure, specified using `intenvset`.

Data Types: `structure`

`StockSpec1` — Stock specification for asset 1
structure

Stock specification for asset 1, specified using `stockspec`.

Data Types: `structure`

`StockSpec2` — Stock specification for asset 2
structure

Stock specification for asset 2, specified using `stockspec`.

Data Types: `structure`

`Settle` — Settlement or trade dates
datetime array | string array | date character vector

Settlement or trade dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `minassetsensbystulz` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity dates
datetime array | string array | date character vector

Maturity dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `minassetsensbystulz` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Option type
cell array of character vectors with a value of `'call'` or `'put'`

Option type, specified as an `NINST`-by-`1` cell array of character vectors with a value of `'call'` or `'put'`.

Data Types: `cell`

`Strike` — Strike prices
vector

Strike prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Corr` — Correlation between the underlying asset prices
vector

Correlation between the underlying asset prices, specified as an NINST-by-1 vector.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [PriceSens] = minassetsensbystulz(RateSpec, StockSpecA,StockSpecB,Settle,Maturity,OptSpec,Strike,Corr,'OutSpec',OutSpec)

OutSpec — Define outputs

{'Price'} (default) | cell array of character vectors with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and a NOUT- by-1 or 1-by-NOUT cell array of character vectors or string array with possible values of 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'.

OutSpec = {'All'} specifies that the output is Delta, Gamma, Vega, Lambda, Rho, Theta, and Price, in that order. This is the same as specifying OutSpec to include each sensitivity:

Example: OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}

Data Types: cell

Output Arguments

PriceSens — Expected prices or sensitivities

vector

Expected prices or sensitivities, returned as an NINST-by-1 or NINST-by-2 vector.

More About

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 Instruments 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.

For more information, see “Rainbow Option” on page 3-27.

Version History

Introduced in R2009a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `minassetsensbystulz` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`intenvset` | `stockspec` | `minassetsensbystulz`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Rainbow Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19

mkbush

Create bushy tree

Syntax

```
[Tree,NumStates = mkbush(NumLevels,NumChild,NumPos)
[Tree,NumStates = mkbush( ___ ,Trim,NodeVal)
```

Description

`[Tree,NumStates = mkbush(NumLevels,NumChild,NumPos)` creates a bushy tree `Tree` with initial values `NodeVal` at each node.

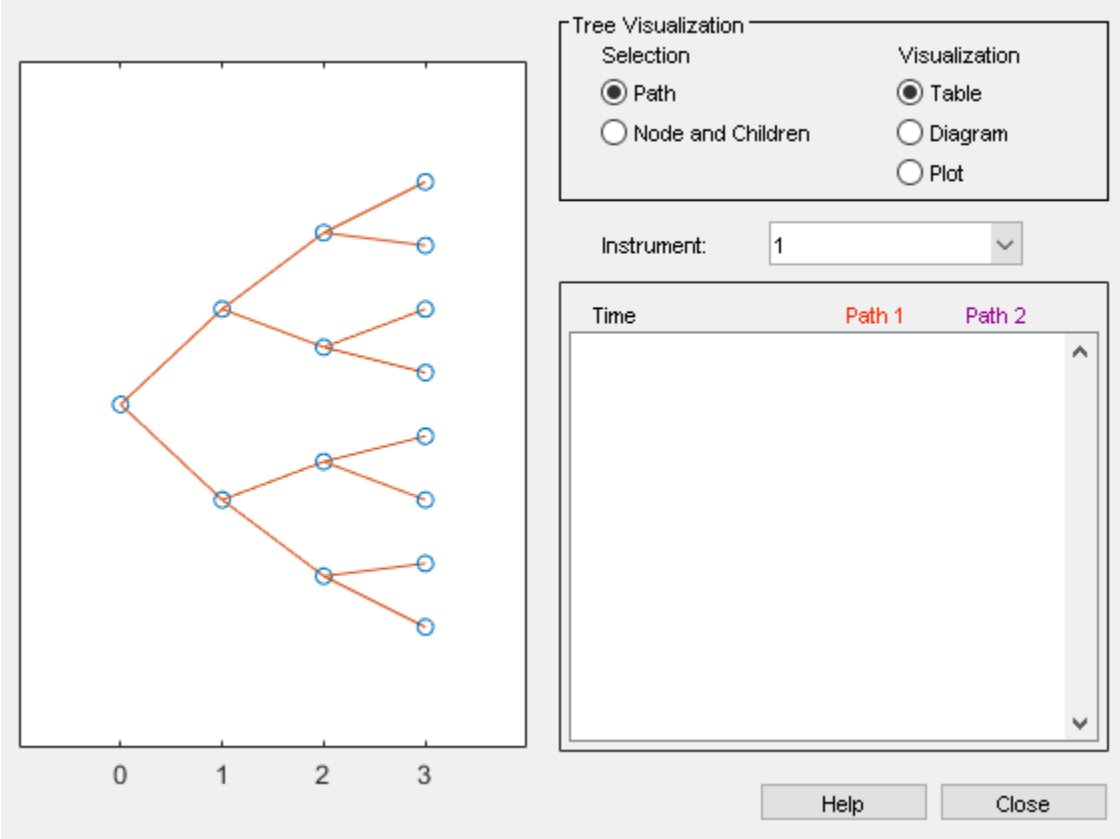
`[Tree,NumStates = mkbush( ___ ,Trim,NodeVal)` adds optional arguments for `Trim` and `NodeVal`.

Examples

Create Bushy Tree

This example shows how to 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)
```



Input Arguments

`NumLevels` — Number of time levels of the tree

numeric

Number of time levels of the tree, specified as a scalar numeric.

Data Types: `double`

`NumChild` — Number of branches (children) of the nodes in each level

vector

Number of branches (children) of the nodes in each level, specified as a 1-by- number of levels (`NUMLEVELS`) vector.

Data Types: `double`

`NumPos` — Length of the state vectors in each time level

vector

Length of the state vectors in each time level, specified as a 1-by-`NUMLEVELS` vector.

Data Types: `double`

`Trim` — Indicates movement between nodes

`0` (default) | logical

(Optional) Indicates movement between nodes, specified as a scalar `0` or `1`. If `Trim = 1`, `NumPos` decreases by `1` when moving from one time level to the next. Otherwise, if `Trim = 0`, `NumPos` does not decrease.

Data Types: `logical`

`NodeVal` — Initial value at each node of the tree
`NaN` (default) | numeric

(Optional) Initial value at each node of the tree, specified as a scalar numeric.

Data Types: `double`

Output Arguments

`Tree` — Bushy tree
struct

Bushy tree, returned as a tree struct with initial values `NodeVal` at each node.

`NumStates` — Number of state vectors in each level
vector

Number of state vectors in each level, returned as a `1`-by-`NUMLEVELS` vector.

Version History

Introduced before R2006a

See Also

`bushpath` | `bushshape`

Topics
“Graphical Representation of Trees” on page 2-219
“Overview of Interest-Rate Tree Models” on page 2-44

mktree

Create recombining binomial tree

Syntax

```
Tree = mktree(NumLevels,NumPos)
Tree = mktree( ___ ,NodeVal,IsPriceTree)
```

Description

`Tree = mktree(NumLevels,NumPos)` creates a recombining tree `Tree` with initial values `NodeVal` at each node.

`Tree = mktree( ___ ,NodeVal,IsPriceTree)` adds optional arguments for `NodeVal` and `IsPriceTree`.

Examples

Create Recombining Binomial Tree

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)
```

```
Tree=1×4 cell array
    {2x1 double}    {2x2 double}    {2x3 double}    {2x4 double}
```

Input Arguments

`NumLevels` — Number of time levels of the tree

numeric

Number of time levels of the tree, specified as a scalar numeric.

Data Types: `double`

`NumPos` — Length of the state vectors in each time level

vector

Length of the state vectors in each time level, specified as a 1-by-`NUMLEVELS` vector.

Data Types: `double`

`NodeVal` — Initial value at each node of the tree

`NaN` (default) | numeric

(Optional) Initial value at each node of the tree, specified as a scalar numeric.

Data Types: `double`

`IsPriceTree` — Indicator if final horizontal branch is added to tree
`0` (default) | logical

(Optional) Indicator if final horizontal branch is added to tree, specified as a scalar logical value.

Data Types: `logical`

Output Arguments

`Tree` — Bushy tree
struct

Bushy tree, returned as a tree struct with initial values `NodeVal` at each node.

Version History

Introduced before R2006a

See Also

`treepath` | `treeshape`

Topics
“Graphical Representation of Trees” on page 2-219
“Overview of Interest-Rate Tree Models” on page 2-44

mktrintree

Create recombining trinomial tree

Syntax

```
TrinTree = mktrintree(NumLevels,NumPos,NumStates)
TrinTree = mktrintree( ___ ,NodeVal)
```

Description

`TrinTree = mktrintree(NumLevels,NumPos,NumStates)` creates a recombining tree `TrinTree` with initial values `NodeVal` at each node.

`TrinTree = mktrintree( ___ ,NodeVal)` adds an optional argument for `NodeVal`.

Examples

Create Recombining Trinomial Tree

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])
```

```
TrinTree=1×4 cell array
    {2x1 double}    {2x3 double}    {2x5 double}    {2x7 double}
```

Input Arguments

`NumLevels` — Number of time levels of the tree

numeric

Number of time levels of the tree, specified as a scalar numeric.

Data Types: `double`

`NumPos` — Length of the state vectors in each time level

vector

Length of the state vectors in each time level, specified as a 1-by-`NUMLEVELS` vector.

Data Types: `double`

`NumStates` — Number of state vectors in each level

vector

Number of state vectors in each level, specified as a 1-by-`NUMLEVELS` vector.

Data Types: `double`

`NodeVal` — Initial value at each node of the tree
`NaN` (default) | numeric

(Optional) Initial value at each node of the tree, specified as a scalar numeric.

Data Types: `double`

Output Arguments

`TrinTree` — Recombining trinomial tree
struct

Recombining trinomial tree, returned as a tree struct with initial values `NodeVal` at each node.

Version History

Introduced before R2006a

See Also

`trintreepath` | `trintreeshape`

Topics
“Graphical Representation of Trees” on page 2-219
“Overview of Interest-Rate Tree Models” on page 2-44

mmktbybdt

Create money-market tree from Black-Derman-Toy interest-rate tree

Syntax

```
MMktTree = mmktbybdt(BDTTree)
MMktTree = mmktbybdt(BDTTree)
```

Description

`MMktTree = mmktbybdt(BDTTree)` `MMktTree = mmktbybdt(BDTTree)` creates a money-market tree from an interest-rate tree structure created by `bdttree`.

Examples

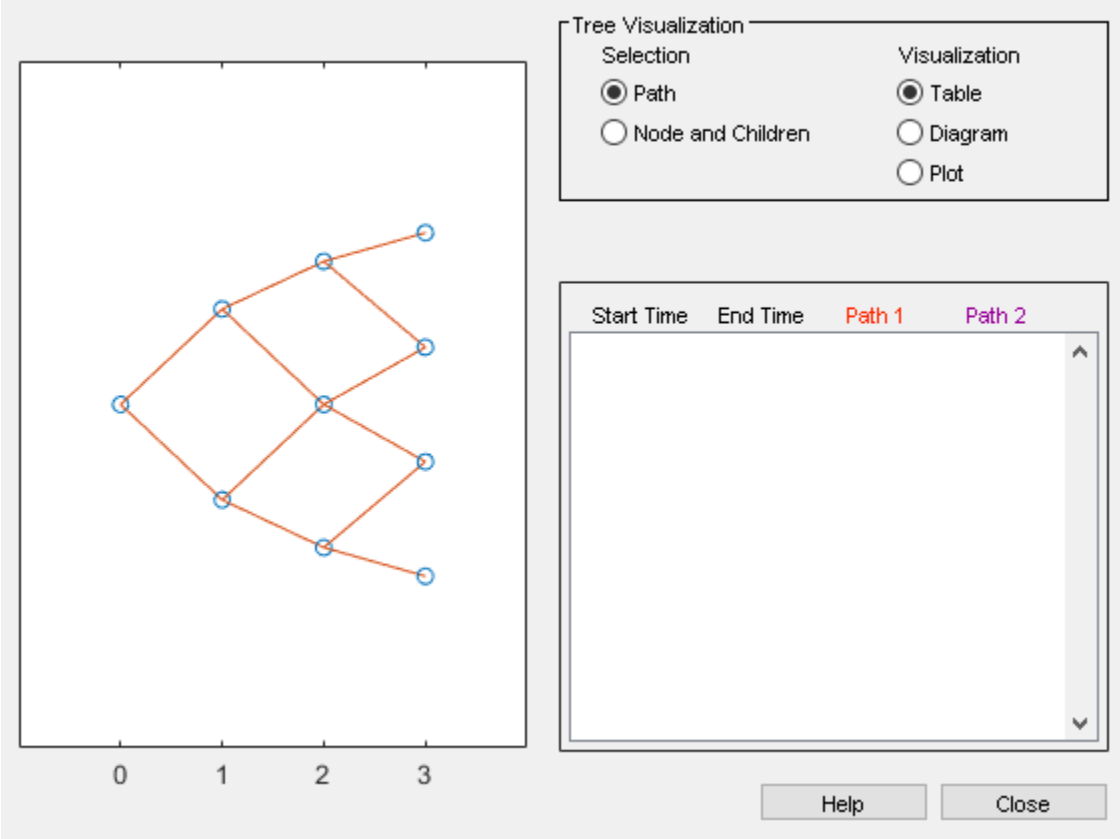
Create Money-Market Tree from Black-Derman-Toy Interest-Rate Tree

Use a `bdttree` from the `deriv.mat` and create a money-market tree from the BDT interest-rate tree.

```
load deriv.mat;
MMktTree = mmktbybdt(BDTTree)
```

```
MMktTree = struct with fields:
      FinObj: 'BDTMmktTree'
        tObs: [0 1 2 3 4]
    MMktTree: {[1]  [1.1000 1.1000]  [1.2077 1.2324 1.2575]  [1.3256 1.3633 ... ]}
```

```
treeviewer(MMktTree)
```



Input Arguments

`BDTTree` — BDT interest-rate tree structure

struct

BDT interest-rate tree structure, specified by `bdttree`.

Data Types: `struct`

Output Arguments

`MMktTree` — Money-market tree

struct

Money-market tree, returned as a tree structure.

Version History

Introduced before R2006a

See Also

`bdttree`

Topics

mmktbyhjm

Create money-market tree from Heath-Jarrow-Morton interest-rate tree

Syntax

```
MMktTree = mmktbyhjm(HJMTree)
```

Description

`MMktTree = mmktbyhjm(HJMTree)` creates a money-market tree from an interest-rate tree structure created by `hjmtree`.

Examples

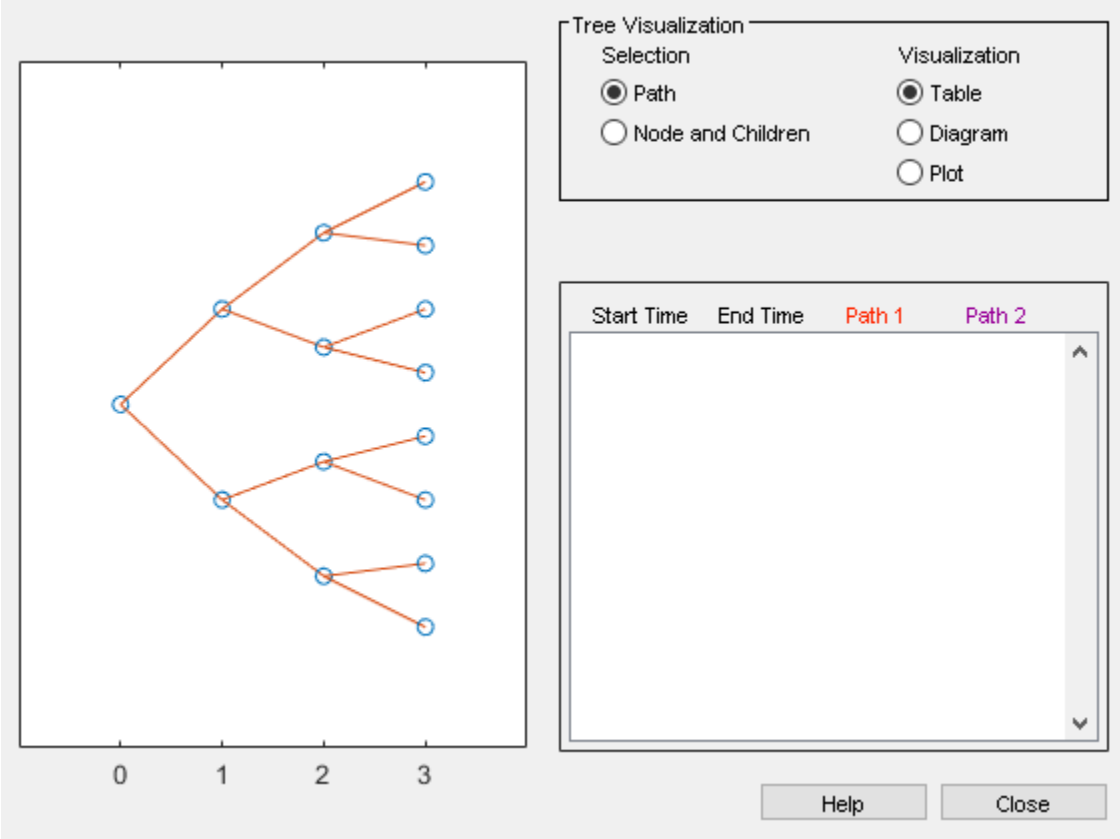
Create Money-Market Tree from Heath-Jarrow-Morton Interest-Rate Tree

Use a `hjmtree` from the `deriv.mat` and create a money-market tree from the HJM interest-rate tree.

```
load deriv.mat;
MMktTree = mmktbyhjm(HJMTree)
```

```
MMktTree = struct with fields:
      FinObj: 'HJMMmktTree'
        tObs: [0 1 2 3 4]
    MMktTree: {[1]  [1x1x2 double]  [1x2x2 double]  [1x4x2 double]}
```

```
treeviewer(MMktTree)
```



Input Arguments

`HJMTree` — HJM interest-rate tree structure

struct

HJM interest-rate tree structure, specified by `hjmtree`.

Data Types: `struct`

Output Arguments

`MMktTree` — Money-market tree

struct

Money-market tree, returned as a tree structure.

Version History

Introduced before R2006a

See Also

`hjmtree`

Topics

“Computing Instrument Prices” on page 2-81
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

normalvolbysabr

Implied Normal (Bachelier) volatility by SABR model

Syntax

```
outVol = normalvolbysabr(Alpha,Beta,Rho,Nu,Settle,ExerciseDate,ForwardValue,
Strike)
outVol = normalvolbysabr( ___ ,Name,Value)
```

Description

`outVol = normalvolbysabr(Alpha,Beta,Rho,Nu,Settle,ExerciseDate,ForwardValue, Strike)` calculates the implied Normal (Bachelier) volatility by using the SABR stochastic volatility model.

`outVol = normalvolbysabr( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Implied Normal (Bachelier) Volatility Using the SABR Model

Define the model parameters and option data.

```
ForwardValue = 0.0209;
Strike = 0.02;
Alpha = 0.041;
Beta = 0.5;
Rho = -0.2;
Nu = 0.33;

Settle = datetime(2018,2,15);
ExerciseDate = datetime(2020,2,15);
```

Compute the Normal (Bachelier) volatility using the SABR model.

```
ComputedVols = normalvolbysabr(Alpha,Beta,Rho,Nu,Settle,ExerciseDate,ForwardValue,Strike)
```

```
ComputedVols = 0.0059
```

Compute the Implied Normal (Bachelier) Volatility Using the Normal SABR Model

To use the Normal SABR model, set the `Beta` parameter to zero. Negative interest rates are allowed when the Normal SABR model is used in combination with Normal (Bachelier) implied volatility.

Define the model parameters and option data.

```
ForwardValue = -0.00383;
Strike = -0.003;
```

```
Alpha = 0.007;
Beta = 0;  % Set the Beta parameter to zero to use the Normal SABR model
Rho = -0.18;
Nu = 0.29;

Settle = datetime(2018,1,17);
ExerciseDate = datetime(2018,4,17);
```

Compute the Normal (Bachelier) volatility using the Normal SABR model.

```
ComputedVols = normalvolbysabr(Alpha,Beta,Rho,Nu,Settle,ExerciseDate,ForwardValue,Strike)
```

```
ComputedVols = 0.0070
```

Input Arguments

`Alpha` — Current SABR volatility

scalar numeric

Current SABR volatility, specified as a scalar numeric.

Data Types: `double`

`Beta` — SABR constant elasticity of variance (CEV) exponent

scalar numeric

SABR CEV exponent, specified as a scalar numeric.

Note Set the `Beta` parameter to `0` to allow a negative `ForwardValue` and `Strike`.

Data Types: `double`

`Rho` — Correlation between forward value and volatility

scalar numeric

Correlation between forward value and volatility, specified as a scalar numeric.

Data Types: `double`

`Nu` — Volatility of volatility

scalar numeric

Volatility of volatility, specified as a scalar numeric.

Data Types: `double`

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `normalvolbysabr` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDate` — Option exercise date
datetime scalar | string scalar | date character vector

Option exercise date, specified as a scalar datetime, string, or date character vector.

To support existing code, `normalvolbysabr` also accepts serial date numbers as inputs, but they are not recommended.

`ForwardValue` — Current forward value of underlying asset
scalar numeric | vector

Current forward value of the underlying asset, specified as a scalar numeric or vector of size `NumVols`-by-`1`.

Data Types: `double`

`Strike` — Option strike price value
scalar numeric | vector

Option strike price value, specified as a scalar numeric or a vector of size `NumVols`-by-`1`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `outVol = normalvolbysabr(Alpha,Beta,Rho,Nu,Settle,ExerciseDate,ForwardValue,Strike,'Basis',2)`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer from the set `[1...13]`.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

`outVol` — Normal (Bachelier) volatility computed by SABR model

scalar numeric | vector

Normal (Bachelier) volatility computed by the SABR model, returned as a scalar numeric or vector of size `NumVols`-by-`1`.

Algorithms

The two general case algorithms for `normalvolbysabr` are not At-The-Money (ATM) and ATM.

For not ATM ($F \neq K$):

$$\sigma_N(\alpha,\beta,\rho,\upsilon,F,K,T) = \alpha\frac{(F-K)(1-\beta)}{F^{(1-\beta)}-K^{(1-\beta)}}\left(\frac{z}{x(z)}\right)\left\{1+\left[\frac{\beta(\beta-2)\alpha^2}{24F_{mid}^{2-2\beta}}+\frac{\rho\beta\upsilon\alpha}{4F_{mid}^{1-\beta}}+\frac{2-3\rho^2}{24}\upsilon^2\right]T\right\}$$

$$=\upsilon\frac{(F-K)}{x(z)}\left\{1+\left[\frac{\beta(\beta-2)\alpha^2}{24F_{mid}^{2-2\beta}}+\frac{\rho\beta\upsilon\alpha}{4F_{mid}^{1-\beta}}+\frac{2-3\rho^2}{24}\upsilon^2\right]T\right\}$$

$$F_{mid}=\frac{F+K}{2}$$

$$z=\frac{\upsilon}{\alpha}\left(\frac{F^{1-\beta}-K^{1-\beta}}{1-\beta}\right)\qquad x(z)=\ln\left(\frac{\sqrt{1-2\rho z+z^2}+z-\rho}{1-\rho}\right)$$

For ATM ($F = K$):

$$\sigma_{N,ATM}(\alpha,\beta,\rho,\upsilon,F,T)=\alpha F^{\beta}\left\{1+\left[\frac{\beta(\beta-2)\alpha^2}{24F^{2-2\beta}}+\frac{\rho\beta\upsilon\alpha}{4F^{1-\beta}}+\frac{2-3\rho^2}{24}\upsilon^2\right]T\right\}$$

The special case for `normalvolbysabr` where β = 0 for not ATM ($F \neq K$) is:

$$\sigma_N(\alpha,\rho,\upsilon,F,K,T)=\upsilon\frac{(F-K)}{\hat{x}(\zeta)}\left(1+\frac{2-3\rho^2}{24}\upsilon^2T\right)$$

$$\zeta=\frac{\upsilon}{\alpha}(F-K)$$

$$\hat{x}(\zeta)=\ln\left(\frac{\sqrt{1-2\rho\zeta+\zeta^2}+\zeta-\rho}{1-\rho}\right)$$

For ATM ($F = K$):

$$\sigma_{N,ATM}(\alpha,\rho,\upsilon,T)=\alpha\left(1+\frac{2-3\rho^2}{24}\upsilon^2T\right)$$

The special case for `normalvolbysabr` where β = 1 for not ATM ($F \neq K$) is:

$$\sigma_N(\alpha,\rho,\upsilon,F,K,T) = \upsilon\frac{(F-K)}{\hat{x}(\zeta)}\left\{1+\left[\frac{-\alpha^2}{24}+\frac{\rho\upsilon\alpha}{4}+\frac{2-3\rho^2}{24}\upsilon^2\right]T\right\}$$

$$\zeta = \frac{\upsilon}{\alpha}\ln\frac{F}{K}$$

$$\hat{x}(\zeta) = \ln\left(\frac{\sqrt{1-2\rho\zeta+\zeta^2}+\zeta-\rho}{1-\rho}\right)$$

For ATM ($F = K$):

$$\sigma_{N,\,ATM}(\alpha,\rho,\upsilon,F,T) = \alpha F\left\{1+\left[\frac{-\alpha^2}{24}+\frac{\rho\upsilon\alpha}{4}+\frac{2-3\rho^2}{24}\upsilon^2\right]T\right\}$$

Version History

Introduced in R2018b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `normalvolbysabr` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hagan, P. S., D. Kumar, A.S. Lesniewski, and D.E. Woodward. "Managing Smile Risk." *Wilmott Magazine*. September 2002, pp. 84–108.

See Also

`swaptionbyblk` | `swaptionbynormal` | `optsensbysabr`

Topics

“Calibrate the SABR Model Using Normal (Bachelier) Volatilities with Negative Strikes” on page 2-163
“Price Swaptions with Negative Strikes Using the Shifted SABR Model” on page 2-26
“Calibrate the SABR Model” on page 2-33
“Price a Swaption Using the SABR Model” on page 2-38
“Work with Negative Interest Rates Using Functions” on page 2-18

“Supported Interest-Rate Instrument Functions” on page 2-3

External Websites
How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

numerix class

Create numerix object to set up Numerix CAIL environment

Description

The `numerix` object makes the Numerix engine directly accessible from MATLAB. To use the capabilities of Numerix CAIL, you must have CAIL client software installed on your local desktop.

In addition, you must add the Numerix library file to MATLAB path to use the documentation examples:

- Add <*Numerix software package installation root*>`/lib` to <*matlabroot*>`/toolbox/local/librarypath.txt`

  or

- Place <*Numerix software package installation root*>`/lib/NxProjava.dll` in the folder <*matlabroot*>`/bin/win64`

Construction

`N = numerix(DATADIRECTORYPATH)` constructs the `numerix` object and sets up the Numerix CrossAsset Integration Layer (CAIL) environment given the path, `DATADIRECTORYPATH`.

Properties

The following properties are from the `numerix` class.

`Path` — Path for `DATADIRECTORYPATH`

character vector

Defines the path for `DATADIRECTORYPATH`, specified as a character vector. This path is the location of the templates and is created by the client installation of CAIL. A template defines the interface; it encapsulates the instructions for performing calculations, the calculation's required and optional input parameters, and the calculation's outputs.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

`RepositoryPath` — Path for `RepositoryPath`

character vector

`RepositoryPath` defines the path location for the repository folder in a file system, specified as a character vector.

Attributes:

SetAccess	public
GetAccess	public

`Repository` — Define repositories
vector

Repositories are collections of templates and are defined as a folder in a file system, specified as a vector.

Attributes:

SetAccess	public
GetAccess	public

`Context` — CAIL application context in which to perform the calculations
vector

The calculation context manages all the CAIL information, specified as a vector. `Context` contains the location of the template repository and is responsible for creating a CAIL application context in which to perform the calculations.

Attributes:

SetAccess	public
GetAccess	public

`LookupsPath` — Path for the numeric instruments data types
character vector

Defines the path for the numeric instruments data types, specified as a character vector.

Attributes:

SetAccess	public
GetAccess	public

`MarketsPath` — Path for logical schema for naming market data
character vector

Defines a path for the logical schema for naming all the market data, specified as a character vector. `MarketsPath` enables you to provide a data dictionary to map business market data to CAIL to reduce the task of inputting market data into CAIL objects directly.

Attributes:

SetAccess	public
GetAccess	public

`FixingsPath` — Path for schema for naming historical fixing data for rates and prices
character vector

Defines the path for the schema for naming historical fixing data for rates and prices, specified as a character vector.

Attributes:

SetAccess	public
GetAccess	public

`TradesPath` — Path to trade instrument definitions

character vector

Defines the path to the trade instrument definitions, specified as a character vector.

Attributes:

SetAccess	public
GetAccess	public

`Parameters` — Calculation parameters and market data

vector

Defines the calculation parameters and market data, if available, specified as a vector.

Attributes:

SetAccess	public
GetAccess	public

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Construct a Numerix Object

Initialize Numerix CAIL environment.

```
import com.numerix.integration.*;
import com.numerix.integration.implementation.*;
```

Construct a numerix object.

```
n = numerix('i:\NumeriX_java_10_3_0\data')
```

```
n =

             Path: 'i:\NumeriX_java_10_3_0\data'
   RepositoryPath: 'i:\NumeriX_java_10_3_0\data\Repository'
       Repository: [1x1 com.numerix.integration.implementation.FileSystemRepository]
          Context: [1x1 com.numerix.integration.implementation.LocalCalculationContext]
      LookupsPath: 'i:\NumeriX_java_10_3_0\data\Data\LookupRules'
      MarketsPath: 'i:\NumeriX_java_10_3_0\data\Data\Markets'
      FixingsPath: 'i:\NumeriX_java_10_3_0\data\Data\Fixings'
```

```
    TradesPath: 'i:\NumeriX_java_10_3_0\data\Data\Trades'
    Parameters: [1x1 com.numerix.integration.implementation.CalculationParameters]
```

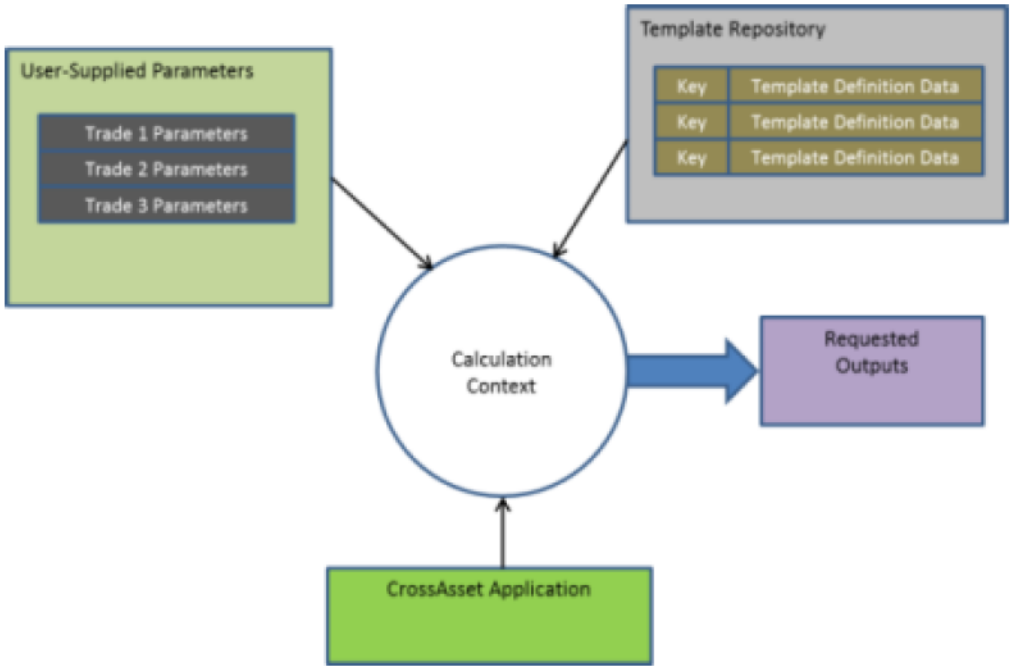
More About

CrossAsset Integration Layer (CAIL)

The CrossAsset Integration Layer (CAIL) is an application programming interface (API), which extends the data-driven approach of Numerix.

The calculation workflow of CAIL is:

1. Select a trade template for a specified deal or trade from the repository. The trade template specifies a set of inputs, a set of outputs, and the dependencies on other information (model, market data, calendar, and so on).
2. Provide the input parameters to the trade template and call the calculation context. The calculation context follows the dependency path to collect the needed information and produces the output specified by the template.



Version History

Introduced in R2013b

See Also

`parseResults` | `numerixCrossAsset`

Topics

“Working with Simple Numerix Trades” on page 10-2
“Working with Advanced Numerix Trades” on page 10-4

“Use Numerix to Price Cash Deposits” on page 10-8
“Use Numerix for Interest-Rate Risk Assessment” on page 10-10
Class Attributes
Property Attributes

External Websites
`https://www.numerix.com/cail`

numerixCrossAsset class

Create `numerixCrossAsset` object to set up Numerix CROSSASSET environment for Java or C++

Description

Creating a `numerixCrossAsset` object initializes a Numerix CrossAsset object based on the Numerix data-driven CROSSASSET API. To use the Numerix engine directly from MATLAB, you must have the CROSSASSET client installed on your local desktop.

You must add the Numerix library file to MATLAB path to use the documentation examples:

- Add <*Numerix software package installation root*>`/lib` to <*matlabroot*>`/toolbox/local/librarypath.txt`

  or
- Place <*Numerix software package installation root*>`/lib/NxProjava.dll` in the folder <*matlabroot*>`/bin/win64`

In addition, when using the Java SDK API, you must add `\NumeriX_Java_12_3_0_2\lib\NxProJava.jar` to the MATLAB file `classpath.txt`.

Construction

`c = numerixCrossAsset` constructs the `numerixCrossAsset` object and sets up the Numerix CROSSASSET environment using the Java SDK API.

`c = numerixCrossAsset(true)` constructs the `numerixCrossAsset` object and sets up the Numerix CROSSASSET environment using the C++ SDK API on Windows.

Properties

`Application` — `Application` object
object

`Application` object, created when `numerixCrossAsset` object is initialized.

Example: `app = Application;`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

`ApplicationWarning` — `ApplicationWarning` object
object

`ApplicationWarning` object, created when `numerixCrossAsset` object is initialized.

Note The `ApplicationWarning` object is `[]` when using the C++ interface.

Example: `appWarnings = ApplicationWarning;`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Methods

applicationCall	Create and register Numerix CROSSASSET Call object
applicationData	Create and register data with Numerix CROSSASSET Application Data object
applicationMatrix	Create and register Numerix CROSSASSET Application Matrix object
close	Close numerixCrossAsset object
getdata	Convert Numerix CROSSASSET Application object to MATLAB structure

Examples

Construct a `numerixCrossAsset` Object for Java

Construct a `numerixCrossAsset` object for the Java SDK API.

```
c = numerixCrossAsset
```

```
 c =

numerixCrossAsset with properties:

Application: [1x1 com.numerix.pro.Application]
ApplicationWarning: [1x1 com.numerix.pro.ApplicationWarning]
```

Construct a `numerixCrossAsset` Object for C++

Construct a `numerixCrossAsset` object for the C++ SDK API on Windows.

```
c = numerixCrossAsset(true)
```

```
 c =

numerixCrossAsset with properties:

Application: [1×1 fininst.internal.NumerixCAIL]
ApplicationWarning: []
```

More About

CROSSASSET

The CROSSASSET API is an application programming interface (API) for Java or C++, which extends the data-driven approach of Numerix.

Version History

Introduced in R2016b

Support for C++ interface

Use `c = numerixCrossAsset(true)` to construct the `numerixCrossAsset` object for a C++ SDK API on Windows.

See Also

`applicationCall` | `applicationData` | `applicationMatrix` | `getdata` | `close`

Topics

“Numerix CROSSASSET Interface Workflow Example Using Matrix, Data, and Call Objects” on page 10-12
Class Attributes
Property Attributes

External Websites

https://www.numerix.com/product/crossasset

applicationCall

Class: `numerixCrossAsset`

Create and register Numerix CROSSASSET Call object

Syntax

```
applicationCall(C,Headers,Name,Value)
```

Description

`applicationCall(C,Headers,Name,Value)` creates and registers the Numerix CROSSASSET Call object with additional options specified by one or more `Name,Value` pair arguments. The name-value parameters conform to the Numerix Cross Asset Integration Layer interface and are defined by `N1`, `N2`, ..., `N`*N* to the values given in `V1`, `V2`, ..., `V`*N*.

Creating and registering the Call object calculates values in the Numerix Cross Asset Integration Layer and returns the data in MATLAB.

Input Arguments

`C` — Connection object to Numerix CROSSASSET

object

Connection object to Numerix CROSSASSET, specified using the `numerixCrossAsset` constructor.

`Headers` — Output names of the returned values from `numerixCrossAsset` connection object

cell array of character vectors

Output names of the returned values from `numerixCrossAsset` connection object, specified as a cell array of character vectors.

Data Types: `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`N1` — Numerix name-value parameter

name-value parameter defined by Numerix

Numerix parameters, specified as a `Name,Value` argument pair.

Example: `applicationCall(c,Headers,'ID','RATESPEC','OBJECT','MARKET DATA','TYPE','YIELD','COMMENT','Comments`

```
here','SKIP',false,'INTERPMETHOD','LogLinear','INTERPVARIABLE','DF','CURRENCY
','USD','DATA','USD_3MLIBOR_CURVE','BASIS','ACT/360');
```

Data Types: `char` | `double` | `logical`

N2 — Numerix name-value parameters
name-value parameter defined by Numerix

Numerix parameter, specified as a `Name,Value` argument pair.

Example: `applicationCall(c,Headers,'ID','RATESPEC','OBJECT','MARKET DATA','TYPE','YIELD','COMMENT','Comments here','SKIP',false,'INTERPMETHOD','LogLinear','INTERPVARIABLE','DF','CURRENCY','USD','DATA','USD_3MLIBOR_CURVE','BASIS','ACT/360');`

Data Types: `char` | `double` | `logical`

Attributes

`Access`	`public`
`Static`	`false`
`Hidden`	`false`

To learn about attributes of methods, see Method Attributes.

Examples

Create and Register a Numerix CROSSASSET Call Object

Create a `datetime` object.

```
dates = datetime({'18-Feb-2014';'20-May-2014';'18-Jun-2014';'16-Jul-2014';
                  '20-Aug-2014';'17-Sep-2014';'15-Oct-2014';'19-Nov-2014';
                 '17-Dec-2014';'18-Mar-2015';'17-Jun-2015';'16-Sep-2015';
                 '16-Dec-2015';'16-Mar-2016';'15-Jun-2016';'21-Sep-2016';
                 '21-Dec-2016';'15-Mar-2017';'20-Feb-2018';'20-Feb-2019';
                 '20-Feb-2020';'22-Feb-2021';'22-Feb-2022';'21-Feb-2023';
                 '20-Feb-2024';'20-Feb-2025';'20-Feb-2026';'20-Feb-2029';
                 '21-Feb-2034';'22-Feb-2039';'22-Feb-2044';'20-Feb-2054';
                 '20-Feb-2064'});
```

Create the corresponding vector of discount factors for a 3-month LIBOR curve.

```
 discountFactors = [1;0.99942;0.999231;0.999037;0.998797;0.998616;0.998385;...
                     0.998122;0.997941;0.997159;0.996157;0.994825;0.993065;...
                     0.99078;0.987889;0.984092;0.979913;0.975459;0.952707;...
                     0.922223;0.888128;0.852291;0.816462;0.781228;0.746677;...
                     0.712892;0.680462;0.592285;0.474003;0.383493;0.312617;...
                     0.213809;0.152345];
```

Create a Numerix CROSSASSET object.

```
c = numerixCrossAsset;
```

Add data to the Numerix Cross Asset Application Data object.

```
applicationData(c,'USD_3MLIBOR_CURVE','DATE',dates,'DISCOUNTFACTOR',discountFactors)
```

Define the `Headers` input and add the `RateSpec` Call object to the Numerix CROSSASSET Application object using name-value pairs, where `USD_3MLIBOR_CURVE` denotes the yield curve data object created previously.

```
headers = {'ID','LOCAL ID','TIMER','TIMER CPU','UPDATED'};
applicationCall(c,headers,'ID','RATESPEC','OBJECT','MARKET DATA','TYPE','YIELD','COMMENT','Comments here',...
                'SKIP',false,'INTERPMETHOD','LogLinear','INTERPVARIABLE','DF',...
                'CURRENCY','USD','DATA','USD_3MLIBOR_CURVE','BASIS','ACT/360');
```

Version History

Introduced in R2016b

See Also

`numerixCrossAsset` | `applicationData` | `applicationMatrix` | `getdata` | `close`

Topics

“Numerix CROSSASSET Interface Workflow Example Using Matrix, Data, and Call Objects” on page 10-12

External Websites

https://www.numerix.com/product/crossasset

applicationData

Class: `numerixCrossAsset`

Create and register data with Numerix CROSSASSET Application Data object

Syntax

```
applicationData(C,Desc,Name,Value)
applicationData(C,Desc,T)
applicationData(C,Desc,S)
```

Description

`applicationData(C,Desc,Name,Value)` `applicationData` creates and registers the data for Numerix CROSSASSET Application Data object with additional options specified by one or more `Name,Value` pair arguments. The name-value parameters conform to the Numerix Cross Asset Integration Layer interface and are defined by `N1`, `N2`, ..., `N`*N* to the values given in `V1`, `V2`, ..., `V`*N*.

`applicationData(C,Desc,T)` creates and registers the data for Numerix CROSSASSET Application Data object in table `T`.

`applicationData(C,Desc,S)` creates and registers the data in the structure, `S`. The structure `fieldnames` represents the property names for the values in each field.

Input Arguments

`C` — Connection object to Numerix CROSSASSET

object

Connection object to Numerix CROSSASSET, specified using the `numerixCrossAsset` constructor.

`Desc` — Description of data

character vector or cell array of character vectors

Description of data, specified as a character vector or cell array of character vectors.

Data Types: `char` | `cell`

`T` — Table input

table `VariableNames` represents the property names for values in the corresponding column

Table input for data to register for the Numerix CROSSASSET Application Data object, specified using `VariableNames`.

Data Types: `table`

`S` — Structure input

structure `fieldnames` represents the property names for the values in each field

Structure input for data to register for the Numerix CROSSASSET Application Data object, specified using the `fieldnames`.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`N1` — Numerix name-value parameter

name-value parameter defined by Numerix

Numerix parameter, specified as `Name,Value` argument pair.

Example:
`applicationData(c,'USD_3MLIBOR_CURVE','DATE',dates,'DISCOUNTFACTOR',discountFactors)`

Data Types: `char` | `double` | `logical`

`N2` — Numerix name-value parameters

name-value parameter defined by Numerix

Numerix parameter, specified as `Name,Value` argument pair.

Example:
`applicationData(c,'USD_3MLIBOR_CURVE','DATE',dates,'DISCOUNTFACTOR',discountFactors)`

Data Types: `char` | `double` | `logical`

Attributes

`Access`	`public`
`Static`	`false`
`Hidden`	`false`

To learn about attributes of methods, see Method Attributes.

Examples

Create a Numerix CROSSASSET Application Data Object

Create a `datetime` object.

```
dates = datetime({'18-Feb-2014';'20-May-2014';'18-Jun-2014';'16-Jul-2014';
                  '20-Aug-2014';'17-Sep-2014';'15-Oct-2014';'19-Nov-2014';
                 '17-Dec-2014';'18-Mar-2015';'17-Jun-2015';'16-Sep-2015';
                 '16-Dec-2015';'16-Mar-2016';'15-Jun-2016';'21-Sep-2016';
                 '21-Dec-2016';'15-Mar-2017';'20-Feb-2018';'20-Feb-2019';
                 '20-Feb-2020';'22-Feb-2021';'22-Feb-2022';'21-Feb-2023';
                 '20-Feb-2024';'20-Feb-2025';'20-Feb-2026';'20-Feb-2029';
                 '21-Feb-2034';'22-Feb-2039';'22-Feb-2044';'20-Feb-2054';
                 '20-Feb-2064'});
```

Create the corresponding vector of discount factors for a 3-month LIBOR curve.

```
discountFactors = [1;0.99942;0.999231;0.999037;0.998797;0.998616;0.998385;...
                       0.998122;0.997941;0.997159;0.996157;0.994825;0.993065;...
                       0.99078;0.987889;0.984092;0.979913;0.975459;0.952707;...
                       0.922223;0.888128;0.852291;0.816462;0.781228;0.746677;...
                       0.712892;0.680462;0.592285;0.474003;0.383493;0.312617;...
                       0.213809;0.152345];
```

Create a Numerix CROSSASSET object.

```
c = numerixCrossAsset;
```

Create and register the data with the Numerix CROSSASSET Application Data object.

```
applicationData(c,'USD_3MLIBOR_CURVE','DATE',dates,'DISCOUNTFACTOR',discountFactors)
```

Version History

Introduced in R2016b

See Also

`numerixCrossAsset` | `applicationCall` | `applicationMatrix` | `getdata` | `close`

Topics

“Numerix CROSSASSET Interface Workflow Example Using Matrix, Data, and Call Objects” on page 10-12

External Websites

https://www.numerix.com/product/crossasset

applicationMatrix

Class: `numerixCrossAsset`

Create and register Numerix CROSSASSET Application Matrix object

Syntax

```
applicationMatrix(C,Desc,Rdata,Cdata,Mdata)
```

Description

`applicationMatrix(C,Desc,Rdata,Cdata,Mdata)` creates the CROSSASSET Application Matrix object from the row information (`Rdata`), column information (`Cdata`), and matrix (`Mdata`).

Input Arguments

`C` — Connection object to Numerix CROSSASSET

object

Connection object to Numerix CROSSASSET, specified using the `numerixCrossAsset` constructor.

`Desc` — Description of data

character vector or cell array of character vectors

Description of data, specified as a character vector or cell array of character vectors.

Data Types: `char` | `cell`

`Rdata` — Row information for Application Matrix object

numeric values

Row information for Application Matrix object, specified using numeric values.

Example: `Rdata = [41992,42020,42449,42905,43115];`

Data Types: `double`

`Cdata` — Column information for Application Matrix object

numeric values

Column information for Application Matrix object, specified as numeric values.

Example: `Cdata = [390,395,400,405];`

Data Types: `double`

`Mdata` — Volatility information for Application Matrix object

numeric values

Volatility information for Application Matrix object, specified as numeric values.

Example:
```
Mdata = [0.35778, 0.35132, 0.34394, 0.33582;...
0.33405, 0.32819, 0.32669, 0.31904;...
```

```
0.31576, 0.31235, 0.30371, 0.30261;...
0.29391, 0.29366, 0.28962, 0.28932;...
0.28787, NaN, 0.28347, NaN ];
```

Data Types: `double`

Attributes

`Access`	`public`
`Static`	`false`
`Hidden`	`false`

To learn about attributes of methods, see Method Attributes.

Examples

Create a Numerix CROSSASSET Application Matrix Object

Create a volatility matrix with dates describing the rows and strike prices describing the columns with the description `BYSTRIKEVOLDATA`. Missing values in the matrix input are denoted as `NaN`.

Create a Numerix CROSSASSET object.

```
c = numerixCrossAsset;
```

Define the matrix data.

```
rowData = [41992, 42020, 42449, 42905, 43115];
colData = [390, 395, 400, 405];
volData = [0.35778, 0.35132, 0.34394, 0.33582;...
           0.33405, 0.32819, 0.32669, 0.31904;...
           0.31576, 0.31235, 0.30371, 0.30261;...
           0.29391, 0.29366, 0.28962, 0.28932;...
           0.28787, NaN,     0.28347, NaN    ];
```

Create and register a Numerix CROSSASSET Application Matrix object.

```
applicationMatrix(c,'BYSTRIKEVOLDATA',rowData,colData,volData);
```

Version History

Introduced in R2016b

See Also

`numerixCrossAsset` | `applicationCall` | `applicationData` | `getdata` | `close`

Topics

“Numerix CROSSASSET Interface Workflow Example Using Matrix, Data, and Call Objects” on page 10-12

External Websites
https://www.numerix.com/product/crossasset

close

Class: `numerixCrossAsset`

Close numerixCrossAsset object

Syntax

```
AppData = close(C)
```

Description

`AppData = close(C)` closes the `numerixCrossAsset` object (`C`).

Input Arguments

`C` — Connection object to Numerix CROSSASSET

object

Connection object to Numerix CROSSASSET, specified using the `numerixCrossAsset` constructor.

Attributes

`Access`	`public`
`Static`	`false`
`Hidden`	`false`

To learn about attributes of methods, see Method Attributes.

Examples

Close the Numerix CROSSASSET Object

Construct a `numerixCrossAsset` object.

```
c = numerixCrossAsset
```

```
 c =

numerixCrossAsset with properties:

Application: [1x1 com.numerix.pro.Application]
ApplicationWarning: [1x1 com.numerix.pro.ApplicationWarning]
```

Close the `numerixCrossAsset` object.

```
close(c)
```

Version History

Introduced in R2016b

See Also

`numerixCrossAsset` | `applicationCall` | `applicationData` | `applicationMatrix` | `getdata`

Topics

“Numerix CROSSASSET Interface Workflow Example Using Matrix, Data, and Call Objects” on page 10-12

External Websites

https://www.numerix.com/product/crossasset

getdata

Class: `numerixCrossAsset`

Convert Numerix CROSSASSET Application object to MATLAB structure

Syntax

```
AppData = getdata(C)
```

Description

`AppData = getdata(C)` converts a Numerix CROSSASSET Application object to a MATLAB structure.

Input Arguments

`C` — Connection object to Numerix CROSSASSET

object

Connection object to Numerix CROSSASSET, specified using the `numerixCrossAsset` constructor.

Output Arguments

`AppData` — Converted Numerix CROSSASSET Application object

structure

Converted Numerix CROSSASSET Application object, returned as a MATLAB structure

Attributes

`Access`	`public`
`Static`	`false`
`Hidden`	`false`

To learn about attributes of methods, see Method Attributes.

Examples

Convert Numerix CROSSASSET Application Object to a MATLAB Structure

Create a `datetime` object.

```
dates = datetime({'18-Feb-2014';'20-May-2014';'18-Jun-2014';'16-Jul-2014';
                  '20-Aug-2014';'17-Sep-2014';'15-Oct-2014';'19-Nov-2014';
                 '17-Dec-2014';'18-Mar-2015';'17-Jun-2015';'16-Sep-2015';
                 '16-Dec-2015';'16-Mar-2016';'15-Jun-2016';'21-Sep-2016';
                 '21-Dec-2016';'15-Mar-2017';'20-Feb-2018';'20-Feb-2019';
                 '20-Feb-2020';'22-Feb-2021';'22-Feb-2022';'21-Feb-2023';
                 '20-Feb-2024';'20-Feb-2025';'20-Feb-2026';'20-Feb-2029';
```

```
                '21-Feb-2034';'22-Feb-2039';'22-Feb-2044';'20-Feb-2054';
                '20-Feb-2064'});
```

Create the corresponding vector of discount factors for a 3-month LIBOR curve.

```
discountFactors = [1;0.99942;0.999231;0.999037;0.998797;0.998616;0.998385;...
                    0.998122;0.997941;0.997159;0.996157;0.994825;0.993065;...
                    0.99078;0.987889;0.984092;0.979913;0.975459;0.952707;...
                    0.922223;0.888128;0.852291;0.816462;0.781228;0.746677;...
                    0.712892;0.680462;0.592285;0.474003;0.383493;0.312617;...
                    0.213809;0.152345];
```

Create a Numerix CROSSASSET object.

```
c = numerixCrossAsset;
```

Add data to the Numerix CROSSASSET Application Data object.

```
applicationData(c,'USD_3MLIBOR_CURVE','DATE',dates,'DISCOUNTFACTOR',discountFactors)
```

Add the `RATESPEC` Call object to the Numerix CROSSASSET Application object using name-value pairs, where `USD_3MLIBOR_CURVE` denotes yield curve data object created previously.

```
headers = {'ID','LOCAL ID','TIMER','TIMER CPU','UPDATED'};
 applicationCall(c,headers,'ID','RATESPEC','OBJECT','MARKET DATA','TYPE','YIELD','COMMENT','Comments here',...
                'SKIP',false,'INTERPMETHOD','LogLinear','INTERPVARIABLE','DF',...
                'CURRENCY','USD','DATA','USD_3MLIBOR_CURVE','BASIS','ACT/360');
```

Use `getdata` to convert the Numerix CROSSASSET Application object to a MATLAB structure.

```
APPDATA = getdata(C)
```

Version History

Introduced in R2016b

See Also

`numerixCrossAsset` | `applicationCall` | `applicationData` | `applicationMatrix` | `close`

Topics

“Numerix CROSSASSET Interface Workflow Example Using Matrix, Data, and Call Objects” on page 10-12

External Websites

https://www.numerix.com/product/crossasset

parseResults

Class: `numerix`

Converts Numerix CAIL data to MATLAB data types

Syntax

```
R = parseResults(N,Results)
```

Description

`R = parseResults(N,Results)` returns Numerix CAIL data in native MATLAB data types.

Input Arguments

`N` — Numerix object

object

Numerix object constructed using `numerix`.

`Results` — Result instances for each trade instance

Numerix CAIL trade

Result instances for each trade instance.

Output Arguments

`R` — Row information for Numerix CAIL data

character vector | numeric

Results from Numerix table output represented as MATLAB data types.

Attributes

`Access`	`public`
`Static`	`false`
`Hidden`	`false`

To learn about attributes of methods, see Method Attributes.

Examples

Return Results for Numerix CAIL API to Price a Callable Reverse Floater

Initialize Numerix environment.

```
import com.numerix.integration.*;
import com.numerix.integration.implementation.*;

n = numerix('i:\NumeriX_java_10_3_0\data')
```

```
n =

             Path: 'i:\NumeriX_java_10_3_0\data'
    RepositoryPath: 'i:\NumeriX_java_10_3_0\data\Repository'
        Repository: [1x1 com.numerix.integration.implementation.FileSystemRepository]
           Context: [1x1 com.numerix.integration.implementation.LocalCalculationContext]
       LookupsPath: 'i:\NumeriX_java_10_3_0\data\Data\LookupRules'
       MarketsPath: 'i:\NumeriX_java_10_3_0\data\Data\Markets'
       FixingsPath: 'i:\NumeriX_java_10_3_0\data\Data\Fixings'
        TradesPath: 'i:\NumeriX_java_10_3_0\data\Data\Trades'
        Parameters: [1x1 com.numerix.integration.implementation.CalculationParameters]
```

Create a market.

```
quotes = java.util.HashMap;
quotes.put('IR.USD-LIBOR-3M.SWAP-1Y.MID', 0.0066056);
quotes.put('IR.USD-LIBOR-3M.SWAP-10Y.MID', 0.022465005);
quotes.put('IR.USD-LIBOR-3M.SWAP-20Y.MID', 0.027544995);
market = Market('EOD_14-NOV-2011', DateExtensions.date('14-Nov-2011'), quotes.entrySet);
```

Define a trade instance based on instrument template found in the `Repository`.

```
tradeDescriptor = 'TRADE.IR.CALLABLEREVERSEFLOATER';
tradeParameters = java.util.HashMap;
tradeParameters.put('Trade ID','1001');
tradeParameters.put('Quote Type', 'MID');
tradeParameters.put('Currency', 'USD');
tradeParameters.put('Notional', 1000000.0);
tradeParameters.put('Effective Date', DateExtensions.date('1-Dec-2011'));
tradeParameters.put('Termination Date', DateExtensions.date('1-Dec-2021'));
tradeParameters.put('IR Index', 'LIBOR');
tradeParameters.put('IR Index Tenor', '3M');
tradeParameters.put('Structured Freq', '3M');
tradeParameters.put('Structured Side', 'Receive');
tradeParameters.put('Structured Coupon Floor', 0.0);
tradeParameters.put('Structured Coupon UpBd', 0.08);
tradeParameters.put('StructuredCoupon Multiplier', 1.4);
tradeParameters.put('Structured Coupon Cap', 0.05);
tradeParameters.put('Structured Basis', 'ACT/360');
tradeParameters.put('Funding Freq', '3M');
tradeParameters.put('Funding Side', 'Pay');
tradeParameters.put('Funding Spread', 0.003);
tradeParameters.put('Funding Basis', 'ACT/360');
tradeParameters.put('Call Start Date', DateExtensions.date('1-Dec-2013'));
tradeParameters.put('Call End Date', DateExtensions.date('1-Dec-2020'));
tradeParameters.put('Option Side', 'Short');
tradeParameters.put('Option Type', 'Right to Terminate');
tradeParameters.put('Call Frequency', '3M');
tradeParameters.put('Model', 'IR.USD-LIBOR-3M.MID.DET');
tradeParameters.put('Method', 'BackwardAnalytic');
```

Create a trade instance.

```
trade = RepositoryExtensions.createTradeInstance(n.Repository, tradeDescriptor, tradeParameters);
```

Price the trades.

```
results = CalculationContextExtensions.calculate(n.Context, trade, market, Request.getAll);
```

Parse the results for MATLAB and display.

```
r = n.parseResults(results)
disp([r.Name r.Category r.Currency r.Data])
```

```
r =
```

```
Category: {13x1 cell}
Currency: {13x1 cell}
    Name: {13x1 cell}
    Data: {13x1 cell}

'Reporting Currency'           'Price'       ''       'USD'
'Structured Cashflow Log'      'Cashflow'    ''        {41x20 cell}
'Structured Leg PV Accrued'    'Price'       'USD'    [          0]
'PV'                           'Price'       'USD'    [ 6.4133e+04]
'Structured Leg PV Clean'      'Price'       'USD'    [ 4.2637e+05]
'Option PV'                    'Price'       'USD'    [-1.3220e+05]
'Funding Cashflow Log'         'Cashflow'    ''        {41x20 cell}
'Structured Leg PV'            'Price'       'USD'    [ 4.2637e+05]
'Funding Leg PV'               'Price'       'USD'    [-2.3004e+05]
'Funding Leg PV Accrued'       'Price'       'USD'    [          0]
'Funding Leg PV Clean'         'Price'       'USD'    [-2.3004e+05]
'Yield Risk Report'            ''            ''        { 4x30 cell}
'Messages'                     ''            ''        { 4x1  cell}
```

Version History

Introduced in R2013b

See Also

numerix

Topics

“Working with Simple Numerix Trades” on page 10-2
“Working with Advanced Numerix Trades” on page 10-4
“Use Numerix to Price Cash Deposits” on page 10-8
“Use Numerix for Interest-Rate Risk Assessment” on page 10-10

External Websites

https://www.numerix.com/cail

oasbybdt

Determine option adjusted spread using Black-Derman-Toy model

Syntax

```
[OAS,OAD,OAC] = oasbybdt(BDTTree,Price,CouponRate,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[OAS,OAD,OAC] = oasbybdt( ___ ,Name,Value)
```

Description

`[OAS,OAD,OAC] = oasbybdt(BDTTree,Price,CouponRate,Settle,Maturity,OptSpec, Strike,ExerciseDates)` calculates option adjusted spread using a Black-Derman-Toy model.

`oasbybdt` computes prices of vanilla bonds with embedded options, stepped coupon bonds with embedded options, amortizing bonds with embedded options, and sinking fund bonds with call embedded option. For more information, see “More About” on page 11-1450.

`[OAS,OAD,OAC] = oasbybdt( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute OAS Using the Black-Derman-Toy (BDT) Model

This example shows how to compute OAS using the Black-Derman-Toy (BDT) model with the following data.

```
ValuationDate = datetime(2010,10,1);
Rates = [0.035; 0.042; 0.047; 0.052];
StartDates = ValuationDate;
EndDates = datemnth(ValuationDate, 12:12:48)';
Compounding = 1;
% define RateSpec
RateSpec = intenvset('ValuationDate', ValuationDate,...
'StartDates', StartDates, 'EndDates', EndDates, ...
'Rates', Rates, 'Compounding', Compounding);

% specify VolSpec and TimeSpec
Sigma = 0.20;
VS = bdtvolspec(ValuationDate, EndDates, Sigma*ones(size(EndDates)));
TS = bdttimespec(ValuationDate, EndDates, Compounding);

% build the BDT tree
BDTTree = bdttree(VS, RateSpec, TS);
BDTTreenew = cvtree(BDTTree);

% instrument information
CouponRate = 0.065;
Settle = ValuationDate;
Maturity = datetime(2014,10,1);
```

```
OptSpec = 'call';
Strike = 100;
ExerciseDates = datetime(2011,10,1);
Period = 1;
Price = 101.58;

% compute the OAS
OAS = oasbybdt(BDTTree, Price, CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates, 'Period', Period)

OAS = 32.7688
```

Compute OAS for an Amortizing Callable Bond Using a BDT Interest-Rate Tree Model

This example shows how to compte the OAS for an amortizing callable bond using a BDT lattice model.

Create a `RateSpec`.

```
Rates = [0.025;0.028;0.030;0.031];
ValuationDate = 'Jan-1-2018';
StartDates = ValuationDate;
EndDates = {'Jan-1-2019'; 'Jan-1-2020'; 'Jan-1-2021'; 'Jan-1-2022'};
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates',...
StartDates, 'EndDates', EndDates, 'Rates', Rates, 'Compounding', Compounding);
```

Build a BDT tree and assume a volatility of 5%.

```
Sigma = 0.05;
BDTTimeSpec = bdttimespec(ValuationDate, EndDates);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Define the callable bond.

```
CouponRate = 0.05;
Settle = 'Jan-1-2018';
Maturity =  'Jan-1-2021';
Period = 1;

    Face = {
                {'Jan-1-2019'  100;
                 'Jan-1-2020'   70; ...
                  'Jan-1-2021'   50};
                };

OptSpec = 'call';
Strike = [97 95 93];
ExerciseDates ={'Jan-1-2019' 'Jan-1-2020' 'Jan-1-2021'};
```

Compute the OAS for a callable amortizing bond using the BDT tree.

```
Price = 99;
BondType = 'amortizing';
```

```
OAS = oasbybdt(BDTT, Price, CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates, 'Period', Period, 'Face', Face,'BondType', BondType)
```

```
OAS = 53.0303
```

Input Arguments

`BDTTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bdttree`.

Data Types: `struct`

`Price` — Market prices of bonds with embedded options

numeric

Market prices of bonds with embedded options, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every bond with an embedded option is set to the `ValuationDate` of the BDT tree. The bond argument `Settle` is ignored.

To support existing code, `oasbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: char | cell

Strike — Option strike price values
nonnegative integer

Option strike price value, specified as a NINST-by-1 or NINST-by-NSTRIKES depending on the type of option:

- European option — NINST-by-1 vector of strike price values.
- Bermuda option — NINST by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- American option — NINST-by-1 vector of strike price values for each option.

Data Types: double

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a NINST-by-1, NINST-by-2, or NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates. Each row is the schedule for one option.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a NINST-by-1 vector, the option is exercised between the underlying bond Settle date and the single listed exercise date.

To support existing code, oasbybdt also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: OAS = oasbybdt(BDTTree,Price,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates,'Period',4)

AmericanOpt — Option type
0 European/Bermuda (default) | integer with values 0 or 1

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and NINST-by-1 positive integer flags with values:

- 0 — European/Bermuda

- `1` — American

Data Types: `double`

`Period` — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

IssueDate — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of 'IssueDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, oasbybdt also accepts serial date numbers as inputs, but they are not recommended.

FirstCouponDate — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, oasbybdt also accepts serial date numbers as inputs, but they are not recommended.

When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

LastCouponDate — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, oasbybdt also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

StartDate — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of 'StartDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, oasbybdt also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify StartDate, the effective start date is the Settle date.

Face — Face value
100 (default) | NINST-by-1 vector | NINST-by-1 cell array

Face or par value, specified as the comma-separated pair consisting of 'Face' and a NINST-by-1 vector or a NINST-by-1 cell array where each element is a NumDates-by-2 cell array where the first

column is dates using a datetime, string, or date character vector, and the second column is associated face value. The date indicates the last day that the face value is valid.

Data Types: `double` | `string` | `char` | `datetime`

`BondType` — Type of underlying bond
`'vanilla'` for scalar `Face` values, `'callablesinking'` for scheduled `Face` values (default) | cell array of character vectors with values `'vanilla'`,`'amortizing'`, or `'callablesinking'` | string array with values `"vanilla"`, `"amortizing"`, or `"callablesinking"`

Type of underlying bond, specified as the comma-separated pair consisting of `'BondType'` and a `NINST`-by-`1` cell array of character vectors or string array specifying if the underlying is a vanilla bond, an amortizing bond, or a callable sinking fund bond. The supported types are:

- `'vanilla'` is a standard callable or puttable bond with a scalar `Face` value and a single coupon or stepped coupons.
- `'callablesinking'` is a bond with a schedule of `Face` values and a sinking fund call provision with a single or stepped coupons.
- `'amortizing'` is an amortizing callable or puttable bond with a schedule of `Face` values with single or stepped coupons.

Data Types: `char` | `string`

`Options` — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`OAS` — Option adjusted spread
vector

Option adjusted spread, returned as a `NINST`-by-`1` vector.

`OAD` — Option adjusted duration
vector

Option adjusted duration, returned as a `NINST`-by-`1` vector.

`OAC` — Option adjusted convexity
vector

Option adjusted convexity, returned as a `NINST`-by-`1` vector.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2011a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `oasbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Fabozzi, F. *Handbook of Fixed Income Securities*. 7th Edition. McGraw-Hill, 2005.

[2] Windas, T. *Introduction to Option-Adjusted Spread Analysis*. 3rd Edition. Bloomberg Press, 2007.

See Also

`bdttree` | `bdtprice` | `instoptembnd` | `oasbybk` | `oasbyhjm` | `oasbyhw` | `optembndbybdt`

Topics

“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Bond with Embedded Options” on page 2-7
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

oasbybk

Determine option adjusted spread using Black-Karasinski model

Syntax

```
[OAS,OAD,OAC] = oasbybk(BKTree,Price,CouponRate,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[OAS,OAD,OAC] = oasbybk( ___ ,Name,Value)
```

Description

`[OAS,OAD,OAC] = oasbybk(BKTree,Price,CouponRate,Settle,Maturity,OptSpec, Strike,ExerciseDates)` calculates option adjusted spread using a Black-Karasinski model.

`oasbybk` computes prices of vanilla bonds with embedded options, stepped coupon bonds with embedded options, amortizing bonds with embedded options, and sinking fund bonds with call embedded option. For more information, see “More About” on page 11-1450.

`[OAS,OAD,OAC] = oasbybk( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute OAS and OAD Using the Black-Karasinski (BK) Model

This example shows how to compute OAS and OAD using the Black-Karasinski (BK) model using the following data.

```
ValuationDate = datetime(2010,8,2);
Rates = [0.0355; 0.0382; 0.0427; 0.0489];
StartDates = ValuationDate;
EndDates = datemnth(ValuationDate, 12:12:48)';
Compounding = 1;

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

% specify VolSpec and TimeSpec
Sigma = 0.10;
Alpha = 0.01;
VS = bkvolspec(ValuationDate, EndDates, Sigma*ones(size(EndDates)),...
EndDates, Alpha*ones(size(EndDates)));
TS = bktimespec(ValuationDate, EndDates, Compounding);

% build the BK tree
BKTree = bktree(VS, RateSpec, TS);

% instrument information
CouponRate = 0.045;
```

```
Settle = ValuationDate;
Maturity = datetime(2014,8,2);
OptSpec = 'put';
Strike = 100;
ExerciseDates = datetime(2013,8,2);
Period = 1;
AmericanOpt = 1;
Price = 101;

% compute OAS and OAD
[OAS, OAD] = oasbybk(BKTree, Price, CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates, 'Period', Period, 'AmericanOpt', AmericanOpt)

OAS = 9.9999e+04

OAD = 0
```

Compute OAS for an Amortizing Callable Bond Using a BK Interest-Rate Tree Model

This example shows how to compute the OAS for an amortizing callable bond using a BK lattice model.

Create a RateSpec.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);
```

Build a BK tree.

```
VolDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2016,1,1);
AlphaCurve = 0.1;

BKVolSpec = bkvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RS.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RS, BKTimeSpec);
```

Define the callable bond.

```
CouponRate = 0.05;
Settle = datetime(2012,1,1);
Maturity = datetime(2016,1,1);
Period = 1;

    Face = {
                {datetime(2014,1,1)  100;
                 datetime(2015,1,1)  70;
                 datetime(2016,1,1)  50};
                };
```

```
OptSpec = 'call';
Strike = [97 95 93];
ExerciseDates =[datetime(2014,1,1) datetime(2015,1,1)  datetime(2016,1,1)];
```

Compute OAS for a callable amortizing bond using the BK tree.

```
Price = 99;
BondType = 'amortizing';
OAS = oasbybk(BKT, Price, CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates, 'Period', Period, 'Face', Face,'BondType', BondType)
```

```
OAS = -13.7366
```

Input Arguments

`BKTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bktree`.

Data Types: `struct`

`Price` — Market prices of bonds with embedded options

numeric

Market prices of bonds with embedded options, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every bond with an embedded option is set to the `ValuationDate` of the BK tree. The bond argument `Settle` is ignored.

To support existing code, `oasbybk` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbybk` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-1 cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
nonnegative integer

Option strike price value, specified as a `NINST`-by-1 or `NINST`-by-`NSTRIKES` depending on the type of option:

- 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.

Data Types: `double`

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-1, `NINST`-by-2, or `NINST`-by-`NSTRIKES` vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a `NINST`-by-1 vector of dates. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a `NINST`-by-`NSTRIKES` vector of dates. Each row is the schedule for one option.
- For an American option, use a `NINST`-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-1 vector, the option is exercised between the underlying bond `Settle` date and the single listed exercise date.

To support existing code, `oasbybk` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
```
OAS =
oasbybk(BDTTree,Price,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates
,'Period',4)
```

`AmericanOpt` — Option type
`0` European/Bermuda (default) | integer with values `0` or `1`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`Period` — Coupons per year
`2` per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbybk` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbybk` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbybk` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbybk` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

Face — Face value
100 (default) | NINST-by-1 vector | NINST-by-1 cell array

Face or par value, specified as the comma-separated pair consisting of 'Face' and a NINST-by-1 vector or a NINST-by-1 cell array where each element is a NumDates-by-2 cell array where the first column is dates using a datetime, string, or date character vector, and the second column is associated face value. The date indicates the last day that the face value is valid.

Data Types: double | char | string | datetime

BondType — Type of underlying bond
'vanilla' for scalar Face values, 'callablesinking' for scheduled Face values (default) | cell array of character vectors with values 'vanilla','amortizing', or 'callablesinking' | string array with values "vanilla", "amortizing", or "callablesinking"

Type of underlying bond, specified as the comma-separated pair consisting of 'BondType' and a NINST-by-1 cell array of character vectors or string array specifying if the underlying is a vanilla bond, an amortizing bond, or a callable sinking fund bond. The supported types are:

- 'vanilla' is a standard callable or puttable bond with a scalar Face value and a single coupon or stepped coupons.
- 'callablesinking' is a bond with a schedule of Face values and a sinking fund call provision with a single or stepped coupons.
- 'amortizing' is an amortizing callable or puttable bond with a schedule of Face values with single or stepped coupons.

Data Types: char | string

Options — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of 'Options' and a structure that is created with derivset.

Data Types: struct

Output Arguments

OAS — Option adjusted spread
vector

Option adjusted spread, returned as a NINST-by-1 vector.

OAD — Option adjusted duration
vector

Option adjusted duration, returned as a NINST-by-1 vector.

OAC — Option adjusted convexity
vector

Option adjusted convexity, returned as a NINST-by-1 vector.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2011a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `oasbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Fabozzi, F. *Handbook of Fixed Income Securities.* 7th Edition. McGraw-Hill, 2005.

[2] Windas, T. *Introduction to Option-Adjusted Spread Analysis.* 3rd Edition. Bloomberg Press, 2007.

See Also

`bktree` | `bkprice` | `instoptembnd` | `optembndbybk` | `oasbyhjm` | `oasbyhw` | `oasbybdt`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Bond with Embedded Options” on page 2-7
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

oasbycir

Determine option adjusted spread using Cox-Ingersoll-Ross model

Syntax

```
[OAS,OAD,OAC] = oasbycir(CIRTree,Price,CouponRate,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[OAS,OAD,OAC] = oasbycir( ___ ,Name,Value)
```

Description

`[OAS,OAD,OAC] = oasbycir(CIRTree,Price,CouponRate,Settle,Maturity,OptSpec, Strike,ExerciseDates)` calculates the option adjusted spread from a Cox-Ingersoll-Ross (CIR) interest-rate tree using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`oasbycir` computes prices of vanilla bonds with embedded options, stepped coupon bonds with embedded options, amortizing bonds with embedded options, and sinking fund bonds with embedded option. For more information, see “More About” on page 11-1450.

`[OAS,OAD,OAC] = oasbycir( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute OAS Using a CIR Interest-Rate Tree

Create a `RateSpec` using the `intenvset` function.

```
ValuationDate = datetime(2018,10,25);
Rates = [0.0355; 0.0382; 0.0427; 0.0489];
StartDates = ValuationDate;
EndDates = datemnth(ValuationDate, 12:12:48)';
Compounding = 1;

RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',EndD
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Maturity = datetime(2023,10,25);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
```

```
    RateSpec: [1x1 struct]
        tObs: [0 1.2500 2.5000 3.7500]
        dObs: [737358 737814 738271 738727]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Define the OAS instrument.

```
CouponRate = 0.045;
Settle = ValuationDate;
Maturity = '25-October-2019';
OptSpec = 'call';
Strike = 100;
ExerciseDates = {'25-October-2018','25-October-2019'};
Period = 1;
AmericanOpt = 0;
Price = 97;
```

Compute the OAS.

```
[OAS,OAD] = oasbycir(CIRT,Price,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates,'Period'
```

```
OAS = 411.4425
```

```
OAD = 0.9282
```

Compute OAS for an Amortizing Callable Bond Using a CIR Interest-Rate Tree

his example shows how to compute the OAS for an amortizing callable bond using a CIR lattice model.

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.025; 0.032; 0.037; 0.042];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = datetime(2016,1,1);
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',EndD
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Maturity = datetime(2019,1,1);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);
CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
```

```
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 0.7500 1.5000 2.2500]
        dObs: [736330 736604 736878 737152]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Define the callable bond.

```
BondSettlement = datetime(2016,1,1);
BondMaturity   = datetime(2020,1,1);
CouponRate = 0.035;
Period = 1;
OptSpec = 'call';
Strike = 100;

 Face = {
                 {datetime(2018,1,1) 100;
                  datetime(2019,1,1) 70;
                  datetime(2020,1,1) 50};
                 };

ExerciseDates = [datetime(2018,1,1) ; datetime(2019,1,1)];
```

Compute OAS for a callable amortizing bond using the CIR tree.

```
Price = 99;
BondType = 'amortizing';
OAS = oasbycir(CIRT, Price, CouponRate, BondSettlement, Maturity,...
OptSpec, Strike, ExerciseDates, 'Period', Period, 'Face', Face,'BondType', BondType)
```

```
OAS = 2×1

   80.4801
   84.3684
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `cirtree`.

Data Types: `struct`

`Price` — Market prices of bonds with embedded options

numeric

Market prices of bonds with embedded options, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an NINST-by-1 decimal annual rate.

Data Types: double

Settle — Settlement date
datetime array | string array | date character vector

Settlement date for the bond option, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

Note The Settle date for every bond with an embedded option is set to the ValuationDate of the CIR tree. The bond argument Settle is ignored.

To support existing code, oasbycir also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, oasbycir also accepts serial date numbers as inputs, but they are not recommended.

OptSpec — Definition of option
character vector with value 'call' or 'put' | cell array of character vectors with values 'call' or 'put' | string array with values "call" or "put"

Definition of option, specified as a NINST-by-1 cell array of character vectors or string arrays with values 'call' or 'put'.

Data Types: char | cell | string

Strike — Option strike price values
nonnegative integer

Option strike price value, specified as a NINST-by-1 or NINST-by-NSTRIKES depending on the type of option:

- European option — NINST-by-1 vector of strike price values.
- Bermuda option — NINST by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- American option — NINST-by-1 vector of strike price values for each option.

Data Types: double

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a NINST-by-1, NINST-by-2, or NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a NINST-by-1 vector, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

.

To support existing code, oasbycir also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: OAS = oasbycir(CIRTree,Price,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates,'Period',4)

AmericanOpt — Option type

0 European/Bermuda (default) | integer with values 0 or 1

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and a NINST-by-1 positive integer flags with values:

- 0 — European/Bermuda
- 1 — American

Data Types: double

Period — Coupons per year

2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of 'Period' and a NINST-by-1 vector.

Data Types: double

Basis — Day-count basis

0 (actual/actual) (default) | integer from 0 to 13

Day-count basis, specified as the comma-separated pair consisting of 'Basis' and a NINST-by-1 vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360

- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbycir` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbycir` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbycir` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbycir` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | `NINST`-by-`1` vector | `NINST`-by-`1` cell array

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector or a `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array where the first column is dates using a datetime, string, or date character vector, and the second column is associated face value. The date indicates the last day that the face value is valid.

Data Types: `double` | `char` | `string` | `datetime`

`BondType` — Type of underlying bond
`'vanilla'` for scalar `Face` values, `'callablesinking'` for scheduled `Face` values (default) | cell array of character vectors with values `'vanilla'`,`'amortizing'`, or `'callablesinking'` | string array with values `"vanilla"`, `"amortizing"`, or `"callablesinking"`

Type of underlying bond, specified as the comma-separated pair consisting of `'BondType'` and a `NINST`-by-`1` cell array of character vectors or string array specifying if the underlying is a vanilla bond, an amortizing bond, or a callable sinking fund bond. The supported types are:

- `'vanilla'` is a standard callable or puttable bond with a scalar `Face` value and a single coupon or stepped coupons.
- `'callablesinking'` is a bond with a schedule of `Face` values and a sinking fund call provision with a single or stepped coupons.
- `'amortizing'` is an amortizing callable or puttable bond with a schedule of `Face` values with single or stepped coupons.

Data Types: `char` | `string`

Output Arguments

`OAS` — Option adjusted spread

vector

Option adjusted spread, returned as a `NINST`-by-1 vector.

`OAD` — Option adjusted duration

vector

Option adjusted duration, returned as a `NINST`-by-1 vector.

`OAC` — Option adjusted convexity

vector

Option adjusted convexity, returned as a `NINST`-by-1 vector.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `oasbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J., and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Bond with Embedded Options” on page 2-7
“Understanding Interest-Rate Tree Models” on page 2-66

“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

oasbyhjm

Determine option adjusted spread using Heath-Jarrow-Morton model

Syntax

```
[OAS,OAD,OAC] = oasbyhjm(HJMTree,Price,CouponRate,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[OAS,OAD,OAC] = oasbyhjm( ___ ,Name,Value)
```

Description

`[OAS,OAD,OAC] = oasbyhjm(HJMTree,Price,CouponRate,Settle,Maturity,OptSpec, Strike,ExerciseDates)` calculates option adjusted spread using a Heath-Jarrow-Morton model.

`oasbyhjm` computes prices of vanilla bonds with embedded options, stepped coupon bonds with embedded options, amortizing bonds with embedded options, and sinking fund bonds with call embedded option. For more information, see “More About” on page 11-1450.

`[OAS,OAD,OAC] = oasbyhjm( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute OAS Using the Heath-Jarrow-Morton (HJM) Model

This example shows how to compute OAS using the Heath-Jarrow-Morton (HJM) model using the following data.

```
ValuationDate = datetime(2010,11,1);
Rates = [0.0356; 0.0427; 0.0478; 0.0529];
StartDates = ValuationDate;
EndDates = datemnth(ValuationDate, 12:12:48)';
Compounding = 1;

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

% specify VolSpec and TimeSpec
Sigma = 0.02;
VS = hjmvolspec('Constant', Sigma);
TS = hjmtimespec(ValuationDate, EndDates, Compounding);

% build the HJM tree
HJMTree = hjmtree(VS, RateSpec, TS);
HJMTreenew = cvtree(HJMTree);

% instrument information
CouponRate = 0.05;
Settle = ValuationDate;
```

```
Maturity = datetime(2014,11,1);
OptSpec = 'call';
Strike = 100;
ExerciseDates = datetime(2010,11,1);
Period = 1;
Price = 97.5;

% compute the OAS
OAS = oasbyhjm(HJMTree, Price, CouponRate, Settle, Maturity, OptSpec, Strike,...
ExerciseDates, 'Period', Period)

OAS = 49.3289
```

Compute OAS for an Amortizing Callable Bond Using an HJM Interest-Rate Tree Model

This example shows how to compute the OAS for amortizing callable bond using an HJM lattice model.

Create a RateSpec.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);
```

Build a HJM tree.

```
VolSpec = hjmvolspec('Constant', 0.01)

VolSpec = struct with fields:
          FinObj: 'HJMVolSpec'
    FactorModels: {'Constant'}
      FactorArgs: {{1x1 cell}}
      SigmaShift: 0
      NumFactors: 1
       NumBranch: 2
         PBranch: [0.5000 0.5000]
     Fact2Branch: [-1 1]


TimeSpec = hjmtimespec(ValuationDate, EndDates, Compounding)

TimeSpec = struct with fields:
           FinObj: 'HJMTimeSpec'
    ValuationDate: 734869
         Maturity: [4x1 double]
      Compounding: 1
            Basis: 0
     EndMonthRule: 1


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

```
HJMTree = struct with fields:
      FinObj: 'HJMFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734869 735235 735600 735965]
        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]}
```

Define the callable bond.

```
CouponRate = 0.05;
Settle = datetime(2012,1,1);
Maturity =  datetime(2016,1,1);
Period = 1;

    Face = {
                 {datetime(2014,1,1) 100;
                  datetime(2015,1,1) 70;
                  datetime(2016,1,1) 50};
                 };

OptSpec = 'call';
Strike = [97 95 93];
ExerciseDates = [datetime(2014,1,1)  datetime(2015,1,1)  datetime(2016,1,1)];
```

Compute the OAS for a callable amortizing bond using the HJM tree.

```
Price = 99;
BondType = 'amortizing';
OAS = oasbyhjm(HJMTree, Price, CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates, 'Period', Period, 'Face', Face,'BondType', BondType)

OAS = -19.1325
```

Input Arguments

`HJMTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hjmtree`.

Data Types: `struct`

`Price` — Market prices of bonds with embedded options

numeric

Market prices of bonds with embedded options, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an NINST-by-1 decimal annual rate.

Data Types: double

Settle — Settlement date
datetime array | string array | date character vector

Settlement date for the bond option, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

Note The Settle date for every bond with an embedded option is set to the ValuationDate of the HJM tree. The bond argument Settle is ignored.

To support existing code, oasbyhjm also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, oasbyhjm also accepts serial date numbers as inputs, but they are not recommended.

OptSpec — Definition of option
character vector with value 'call' or 'put' | cell array of character vectors with values 'call' or 'put'

Definition of option, specified as a NINST-by-1 cell array of character vectors.

Data Types: char | cell

Strike — Option strike price values
nonnegative integer

Option strike price value, specified as a NINST-by-1 or NINST-by-NSTRIKES depending on the type of option:

- European option — NINST-by-1 vector of strike price values.
- Bermuda option — NINST by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- American option — NINST-by-1 vector of strike price values for each option.

Data Types: double

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a NINST-by-1, NINST-by-2, or NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates. Each row is the schedule for one option.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a NINST-by-1 vector, the option is exercised between the underlying bond Settle date and the single listed exercise date.

To support existing code, oasbyhjm also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: OAS = oasbybk(BDTTree,Price,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates,'Period',4)

AmericanOpt — Option type

0 European/Bermuda (default) | integer with values 0 or 1

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and NINST-by-1 positive integer flags with values:

- 0 — European/Bermuda
- 1 — American

Data Types: double

Period — Coupons per year

2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of 'Period' and a NINST-by-1 vector.

Data Types: double

Basis — Day-count basis

0 (actual/actual) (default) | integer from 0 to 13

Day-count basis, specified as the comma-separated pair consisting of 'Basis' and a NINST-by-1 vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | `NINST`-by-`1` vector | `NINST`-by-`1` cell array

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector or a `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array where the first column is dates using a datetime, string, or date character vector, and the second column is associated face value. The date indicates the last day that the face value is valid.

Data Types: `double` | `char` | `string` | `datetime`

`BondType` — Type of underlying bond
`'vanilla'` for scalar `Face` values, `'callablesinking'` for scheduled `Face` values (default) | cell array of character vectors with values `'vanilla'`,`'amortizing'`, or `'callablesinking'` | string array with values `"vanilla"`, `"amortizing"`, or `"callablesinking"`

Type of underlying bond, specified as the comma-separated pair consisting of `'BondType'` and a `NINST`-by-`1` cell array of character vectors or string array specifying if the underlying is a vanilla bond, an amortizing bond, or a callable sinking fund bond. The supported types are:

- `'vanilla'` is a standard callable or puttable bond with a scalar `Face` value and a single coupon or stepped coupons.
- `'callablesinking'` is a bond with a schedule of `Face` values and a sinking fund call provision with a single or stepped coupons.
- `'amortizing'` is an amortizing callable or puttable bond with a schedule of `Face` values with single or stepped coupons.

Data Types: `char` | `string`

`Options` — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`OAS` — Option adjusted spread
vector

Option adjusted spread, returned as a `NINST`-by-`1` vector.

`OAD` — Option adjusted duration
vector

Option adjusted duration, returned as a `NINST`-by-`1` vector.

`OAC` — Option adjusted convexity
vector

Option adjusted convexity, returned as a `NINST`-by-`1` vector.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via

a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2011a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `oasbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Fabozzi, F. *Handbook of Fixed Income Securities.* 7th Edition. McGraw-Hill, 2005.

[2] Windas, T. *Introduction to Option-Adjusted Spread Analysis.* 3rd Edition. Bloomberg Press, 2007.

See Also

`hjmtree` | `hjmprice` | `instoptembnd` | `optembndbyhjm` | `oasbybdt` | `oasbyhw` | `oasbybk`

Topics

“Computing Instrument Prices” on page 2-81
“Bond with Embedded Options” on page 2-7
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

oasbyhw

Determine option adjusted spread using Hull-White model

Syntax

```
[OAS,OAD,OAC] = oasbyhw(HWTree,Price,CouponRate,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[OAS,OAD,OAC] = oasbyhw( ___ ,Name,Value)
```

Description

`[OAS,OAD,OAC] = oasbyhw(HWTree,Price,CouponRate,Settle,Maturity,OptSpec, Strike,ExerciseDates)` calculates option adjusted spread using a Hull-White model.

`oasbyhw` computes prices of vanilla bonds with embedded options, stepped coupon bonds with embedded options, amortizing bonds with embedded options, and sinking fund bonds with call embedded option. For more information, see “More About” on page 11-1450.

`[OAS,OAD,OAC] = oasbyhw( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute OAS and OAD Using the Hull-White (HW) Model

This example shows how to compute OAS and OAD using the Hull-White (HW) model using the following data.

```
ValuationDate = datetime(2010,10,25);
Rates = [0.0355; 0.0382; 0.0427; 0.0489];
StartDates = ValuationDate;
EndDates = datemnth(ValuationDate, 12:12:48)';
Compounding = 1;

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

% specify VolsSpec and TimeSpec
Sigma = 0.05;
Alpha = 0.01;
VS = hwvolspec(ValuationDate, EndDates, Sigma*ones(size(EndDates)),...
EndDates, Alpha*ones(size(EndDates)));
TS = hwtimespec(ValuationDate, EndDates, Compounding);

% build the HW tree
HWTree = hwtree(VS, RateSpec, TS);

% instrument information
CouponRate = 0.045;
```

```
Settle = ValuationDate;
Maturity = datetime(2014,10,25);
OptSpec = 'call';
Strike = 100;
ExerciseDates = [datetime(2010,10,25) ; datetime(2013,10,25)];
Period = 1;
AmericanOpt = 0;
Price = 97;

% compute the OAS
[OAS, OAD] = oasbyhw(HWTree, Price, CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates, 'Period', Period, 'AmericanOpt', AmericanOpt)

OAS = 2×1

   52.3118
  -12.8538


OAD = 2×1

    3.5507
    3.2910
```

Compute the OAS to Measure Cost of an Embedded Option Relative to a Risk-Free Curve

This example shows how to compute the price of a callable bond using a Hull-White tree.

Use the following bond data:

```
Settle = datetime(2014,8,20);

% Bond Properties
Maturity = datetime(2034,4,1);
CouponRate = .0625;
CallDates = datemnth('01-Oct-2014',6*(0:19));
CallStrikes = [102.85 102.7 102.55 102.4 102.25 102.1 101.95 101.8 ...
    101.65 101.5 101.35 101.2 101.05 100.9 100.75 100.6 100.45 100.3 ...
    100.15 100];
```

Use the following zero-curve data:

```
CurveDates = datemnth(Settle,12*[1 2 3 5 7 10 20 30]');
ZeroRates = [.11 0.30 0.64 1.44 2.07 2.61 3.29 3.55]'/100;
```

Define the `RateSpec` and build the HW tree.

```
RateSpec = intenvset('StartDate',Settle,'EndDates',CurveDates,'Rates',ZeroRates);

% HW Model Properties
alpha = .1;
sigma = .01;

TimeSpec = hwtimespec(Settle,cfdates(Settle,Maturity,12),2);
VolSpec = hwvolspec(Settle,Maturity,sigma,Maturity,alpha);
```

```
HWTree = hwtree(VolSpec,RateSpec,TimeSpec,'method','HW2000')
```

```
HWTree = struct with fields:
      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 0.0652 0.2295 0.3967 0.5628 0.7283 0.8967 1.0652 1.2295 ... ]
        dObs: [735831 735843 735873 735904 735934 735965 735996 736024 ... ]
      CFlowT: {1x236 cell}
       Probs: {1x235 cell}
     Connect: {1x235 cell}
     FwdTree: {1x236 cell}
```

Compute the OAS for the bond.

```
Price = 103.25;
OAS = oasbyhw(HWTree, Price, CouponRate, Settle, Maturity, 'call', CallStrikes, CallDates)
```

```
OAS = 234.8209
```

If you want to compute an OAS that only measures the option cost, you can pass in an issuer-specific curve instead of a risk-free curve (this would be done in the `RateSpec` argument).

Compute OAS for an Amortizing Callable Bond Using an HW Interest-Rate Tree Model

This example shows how to compute the OAS for an amortizing callable bond using an HW lattice model.

Create a `RateSpec`.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = 'Jan-1-2012';
StartDates = ValuationDate;
EndDates = {'Jan-1-2013'; 'Jan-1-2014'; 'Jan-1-2015'; 'Jan-1-2016'};
Compounding = 1;
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);
```

Build a HW tree.

```
VolDates = ['1-Jan-2013'; '1-Jan-2014'; '1-Jan-2015'; '1-Jan-2016'];
VolCurve = 0.01;
AlphaDates = '01-01-2016';
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RS.ValuationDate, EndDates, Compounding);
HWT = hwtree(HWVolSpec, RS, HWTimeSpec);
```

Define the callable bond.

```
CouponRate = 0.05;
Settle = 'Jan-1-2012';
```

```
Maturity =  'Jan-1-2016';
Period = 1;

    Face = {
                  {'Jan-1-2014'  100;
                   'Jan-1-2015'   70;
                   'Jan-1-2016'   50};
                  };

OptSpec = 'call';
Strike = [97 95 93];
ExerciseDates ={'Jan-1-2014' 'Jan-1-2015' 'Jan-1-2016'};
```

Compute the OAS for a callable amortizing bond using the HW tree.

```
Price = 55;
BondType = 'amortizing';
OAS = oasbyhw(HWT,Price, CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates, 'Period', Period, 'Face', Face,'BondType', BondType)
```

```
OAS = 2.4023e+03
```

Input Arguments

`HWTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hwtree`.

Data Types: `struct`

`Price` — Market prices of bonds with embedded options

numeric

Market prices of bonds with embedded options, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every bond with an embedded option is set to the `ValuationDate` of the HW tree. The bond argument `Settle` is ignored.

To support existing code, `oasbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
nonnegative integer

Option strike price value, specified as a `NINST`-by-`1` or `NINST`-by-`NSTRIKES` depending on the type of option:

- 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.

Data Types: `double`

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1`, `NINST`-by-`2`, or `NINST`-by-`NSTRIKES` vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a `NINST`-by-`1` vector of dates. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a `NINST`-by-`NSTRIKES` vector of dates. Each row is the schedule for one option.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-`1` vector, the option is exercised between the underlying bond `Settle` date and the single listed exercise date.

To support existing code, `oasbyhw` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `OAS = oasbybk(BDTTree,Price,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates,'Period',4)`

`AmericanOpt` — Option type

`0` European/Bermuda (default) | integer with values `0` or `1`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`Period` — Coupons per year

2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis

`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbyhw` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbyhw` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `oasbyhw` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | `NINST`-by-`1` vector | `NINST`-by-`1` cell array

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector or a `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array where the first column is dates using a datetime, string, or date character vector, and the second column is associated face value. The date indicates the last day that the face value is valid.

Data Types: `double` | `char` | `string` | `datetime`

`BondType` — Type of underlying bond
`'vanilla'` for scalar `Face` values, `'callablesinking'` for scheduled `Face` values (default) | cell array of character vectors with values `'vanilla'`,`'amortizing'`, or `'callablesinking'` | string array with values `"vanilla"`, `"amortizing"`, or `"callablesinking"`

Type of underlying bond, specified as the comma-separated pair consisting of `'BondType'` and a `NINST`-by-`1` cell array of character vectors or string array specifying if the underlying is a vanilla bond, an amortizing bond, or a callable sinking fund bond. The supported types are:

- `'vanilla'` is a standard callable or puttable bond with a scalar `Face` value and a single coupon or stepped coupons.
- `'callablesinking'` is a bond with a schedule of `Face` values and a sinking fund call provision with a single or stepped coupons.
- `'amortizing'` is an amortizing callable or puttable bond with a schedule of `Face` values with single or stepped coupons.

Data Types: `char` | `string`

`Options` — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`OAS` — Option adjusted spread
vector

Option adjusted spread, returned as a `NINST`-by-`1` vector.

`OAD` — Option adjusted duration
vector

Option adjusted duration, returned as a `NINST`-by-`1` vector.

`OAC` — Option adjusted convexity
vector

Option adjusted convexity, returned as a `NINST`-by-`1` vector.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2011a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `oasbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Fabozzi, F. *Handbook of Fixed Income Securities.* 7th Edition. McGraw-Hill, 2005.

[2] Windas, T. *Introduction to Option-Adjusted Spread Analysis.* 3rd Edition. Bloomberg Press, 2007.

See Also

`hwtree` | `hwprice` | `instoptembnd` | `optembndbyhw` | `oasbybdt` | `oasbyhjm` | `oasbybk`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Bond with Embedded Options” on page 2-7
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

optbndbybdt

Price bond option from Black-Derman-Toy interest-rate tree

Syntax

```
[Price,PriceTree] = optbndbybdt(BDTTree,OptSpec,Strike,ExerciseDates,
AmericanOpt,CouponRate,Settle,Maturity)
[Price,PriceTree] = optbndbybdt( ___ ,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate,StartDate,Face,Options)
```

Description

`[Price,PriceTree] = optbndbybdt(BDTTree,OptSpec,Strike,ExerciseDates, AmericanOpt,CouponRate,Settle,Maturity)` calculates the price for a bond option from a Black-Derman-Toy interest-rate tree.

Note Alternatively, you can use the `FixedBondOption` object to price fixed-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optbndbybdt( ___ ,Period,Basis,EndMonthRule,IssueDate, FirstCouponDate,LastCouponDate,StartDate,Face,Options)` adds optional arguments.

Examples

Price a European Call and Put Option on a Bond

Using the BDT interest-rate tree in the `deriv.mat` file, price a European call and put 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 `'Call'` option.

```
[Price,PriceTree] = optbndbybdt(BDTTree,'Call',95,'01-Jan-2002',...
0,0.10,datetime(2000,1,1),datetime(2003,1,1),1)
```

```
Price = 1.7657
```

```
PriceTree = struct with fields:
    FinObj: 'BDTPriceTree'
      tObs: [0 1 2 3 4]
     PTree: {[1.7657]  [3.1458 0.7387]  [5.2187 1.6890 0]  [0 0 0 0]  [0 0 0 0]}
    ExTree: {[0]  [0 0]  [1 1 0]  [0 0 0 0]  [0 0 0 0]}
```

Now use `optbndbybdt` to compute the price of a `'Put'` option on the same bond.

```
[Price,PriceTree] = optbndbybdt(BDTTree,'Put',95,'01-Jan-2002',...
0,0.10,datetime(2000,1,1),datetime(2003,1,1),1)
```

```
Price = 0.5740
```

```
PriceTree = struct with fields:
    FinObj: 'BDTPriceTree'
      tObs: [0 1 2 3 4]
     PTree: {[0.5740]  [0 1.2628]  [0 0 2.8871]  [0 0 0 0]  [0 0 0 0]}
    ExTree: {[0]  [0 0]  [0 0 1]  [0 0 0 0]  [0 0 0 0]}
```

The `PriceTree.ExTree` output for the `'Call'` and `'Put'` option contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it is not.

Input Arguments

`BDTTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bdttree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char`

`Strike` — Option strike price values

nonnegative integer

Option strike price value, specified as a `NINST`-by-`1` or `NINST`-by-`NSTRIKES` depending on the type of option:

- 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.

Data Types: `double`

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1`, `NINST`-by-`2`, or `NINST`-by-`NSTRIKES` vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a NINST-by-1 vector, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

To support existing code, optbndbybdt also accepts serial date numbers as inputs, but they are not recommended.

AmericanOpt — Option type
0 European/Bermuda (default) | integer with values 0 or 1

(Optional) Option type, specified as NINST-by-1 positive integer flags with values:

- 0 — European/Bermuda
- 1 — American

Data Types: double

CouponRate — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an NINST-by-1 decimal annual rate or NINST-by-1 cell array, where each element is a NumDates-by-2 cell array. The first column of the NumDates-by-2 cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: double | cell

Settle — Settlement date
datetime array | string array | date character vector

Settlement date for the bond option, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

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

To support existing code, optbndbybdt also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optbndbybdt also accepts serial date numbers as inputs, but they are not recommended.

`Period` — Coupons per year
2 per year (default) | vector

(Optional) Coupons per year, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis, specified as a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

(Optional) End-of-month rule flag is specified as a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

(Optional) Bond issue date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

(Optional) Irregular first coupon date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

(Optional) Irregular last coupon date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

(Optional) Forward starting date of payments (the date from which a bond cash flow is considered), specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

(Optional) Face or par value, specified as an`NINST`-by-`1` vector.

Data Types: `double`

`Options` — Derivatives pricing options
structure

(Optional) Derivatives pricing options, specified as structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of bond option at time `0`

matrix

Expected price of the bond option at time `0`, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure containing trees of vectors of instrument prices and accrued interest for each node

structure

Structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it isn't.

More About

Bond Option

A bond option gives the holder 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.

Financial Instruments Toolbox 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.

For more information, see “Bond Options” on page 2-6.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optbndbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdtprice` | `bdttree` | `instoptbnd` | `FixedBondOption`

Topics

“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond Options” on page 2-6
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optbndbybk

Price bond option from Black-Karasinski interest-rate tree

Syntax

```
[Price,PriceTree] = optbndbybk(BKTree,OptSpec,Strike,ExerciseDates,
AmericanOpt,CouponRate,Settle,Maturity)
[Price,PriceTree] = optbndbybk( ___ ,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate,StartDate,Face,Options)
```

Description

`[Price,PriceTree] = optbndbybk(BKTree,OptSpec,Strike,ExerciseDates, AmericanOpt,CouponRate,Settle,Maturity)` calculates the price for a bond option from a Black-Karasinski interest-rate tree.

Note Alternatively, you can use the `FixedBondOption` object to price fixed-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optbndbybk( ___ ,Period,Basis,EndMonthRule,IssueDate, FirstCouponDate,LastCouponDate,StartDate,Face,Options)` adds optional arguments.

Examples

Price a European Call and Put Option on a Bond

Using the BK interest rate tree in the `deriv.mat` file, price a European call and put 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 `'Call'` option.

```
[Price,PriceTree] = optbndbybk(BKTree,'Call',96,'01-Jan-2006',...
0,0.04,datetime(2005,1,1),datetime(2009,1,1))
```

```
Warning: OptBonds are valued at Tree ValuationDate rather than Settle.
> In optbndbytrintree (line 40)
  In optbndbybk (line 92)
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In optbndbytrintree (line 151)
  In optbndbybk (line 92)
```

```
Price =

    0.1512


PriceTree =

  struct with fields:

     FinObj: 'BKPriceTree'
      PTree: {[0.1512]  [0.0281 0.1481 0.3119]  [0 0 0.1329 0.3886 0.3086]  [0 0 0 0 0]  [0 0 0 0
       tObs: [0 1 2 3 4]
    Connect: {[2]  [2 3 4]  [2 2 3 4 4]}
      Probs: {[3×1 double]  [3×3 double]  [3×5 double]}
     ExTree: {[0]  [0 0 0]  [0 0 1 1 1]  [0 0 0 0 0]  [0 0 0 0 0]}
```

Now use `optbndbybdt` to compute the price of a `'Put'` option on the same bond.

```
[Price,PriceTree] = optbndbybk(BKTree,'Put',96,'01-Jan-2006',...
0,0.04,datetime(2005,1,1),datetime(2009,1,1))

Warning: OptBonds are valued at Tree ValuationDate rather than Settle.
> In optbndbytrintree (line 40)
  In optbndbybk (line 92)
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In optbndbytrintree (line 151)
  In optbndbybk (line 92)

Price =

    0.0272


PriceTree =

  struct with fields:

     FinObj: 'BKPriceTree'
      PTree: {[0.0272]  [0.0860 0.0204 0]  [0.0474 0.1266 0 0 0]  [0 0 0 0 0]  [0 0 0 0 0]}
       tObs: [0 1 2 3 4]
    Connect: {[2]  [2 3 4]  [2 2 3 4 4]}
      Probs: {[3×1 double]  [3×3 double]  [3×5 double]}
     ExTree: {[0]  [0 0 0]  [1 1 0 0 0]  [0 0 0 0 0]  [0 0 0 0 0]}
```

The `PriceTree.ExTree` output for the `'Call'` and `'Put'` option contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it is not.

Input Arguments

BKTree — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bktree`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char`

`Strike` — Option strike price values
nonnegative integer

Option strike price value, specified as a `NINST`-by-`1` or `NINST`-by-`NSTRIKES` depending on the type of option:

- 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.

Data Types: `double`

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1`, `NINST`-by-`2`, or `NINST`-by-`NSTRIKES` vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a `NINST`-by-`1` vector of dates. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a `NINST`-by-`NSTRIKES` vector of dates.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-`1` vector, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `optbndbybk` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
`0` European/Bermuda (default) | integer with values `0` or `1`

(Optional) Option type, specified as `NINST`-by-`1` positive integer flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`CouponRate` — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates

and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optbndbybk` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbybk` also accepts serial date numbers as inputs, but they are not recommended.

`Period` — Coupons per year
`2` per year (default) | vector

(Optional) Coupons per year, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis, specified as a `NINST`-by-`1` vector of integers.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

(Optional) End-of-month rule flag is specified as a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

(Optional) Bond issue date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbybk` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

(Optional) Irregular first coupon date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbybk` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

(Optional) Irregular last coupon date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbybk` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

(Optional) Forward starting date of payments (the date from which a bond cash flow is considered), specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbybk` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

(Optional) Face or par value, specified as an`NINST`-by-`1` vector.

Data Types: `double`

`Options` — Derivatives pricing options
structure

(Optional) Derivatives pricing options, specified as structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of bond option at time `0`
matrix

Expected price of the bond option at time `0`, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure containing trees of vectors of instrument prices and accrued interest for each node
structure

Structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.
- `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it isn't.

More About

Bond Option

A bond option gives the holder 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.

Financial Instruments Toolbox 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.

For more information, see “Bond Options” on page 2-6.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optbndbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bkprice` | `bktree` | `instoptbnd` | `FixedBondOption`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond Options” on page 2-6
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optbndbycir

Price bond option from Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = optbndbycir(CIRTree,OptSpec,Strike,ExerciseDates,
AmericanOpt,CouponRate,Settle,Maturity)
[Price,PriceTree] = optbndbycir( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optbndbycir(CIRTree,OptSpec,Strike,ExerciseDates, AmericanOpt,CouponRate,Settle,Maturity)` calculates the price for a bond option from a Cox-Ingersoll-Ross (CIR) interest-rate tree using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = optbndbycir( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a European Call and Put Option on a Bond Using a CIR Interest-Rate Tree

Compute the price for a European call option on a 4% bond with a strike of 96. The exercise date for the option is Jan. 01, 2018. The settle date for the bond is Jan. 01, 2017, and the maturity date is Jan. 01, 2020.

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = 'Jan-1-2017';
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',EndD
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = datetime(2017,1,1);
Maturity = datetime(2019,1,1);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
```

```
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 0.5000 1 1.5000]
        dObs: [736696 736878 737061 737243]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Price the `'Call'` option.

```
[Price,PriceTree] = optbndbycir(CIRT,'Call',96,datetime(2018,1,1),...
0,0.04,datetime(2017,1,1),datetime(2020,1,1))
```

```
Price = 2.6827

PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
       tObs: [0 0.5000 1 1.5000 2]
      PTree: {1x5 cell}
    Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
     ExTree: {[0]  [0 0 0]  [0 0 1 1 1]  [0 0 0 0 0 0 0]  [0 0 0 0 0 0 0]}
```

Price the `'Put'` option.

```
[Price,PriceTree] = optbndbycir(CIRT,'Put',96,datetime(2018,1,1),...
0,0.04,datetime(2017,1,1),datetime(2020,1,1))
```

```
Price = 0.6835

PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
       tObs: [0 0.5000 1 1.5000 2]
      PTree: {1x5 cell}
    Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
     ExTree: {[0]  [0 0 0]  [1 1 0 0 0]  [0 0 0 0 0 0 0]  [0 0 0 0 0 0 0]}
```

The `PriceTree.ExTree` output for the `'Call'` and `'Put'` option contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it is not.

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `cirtree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'` | string arrays with values `"call"` or `"put"`

Definition of option, specified as a NINST-by-1 cell array of character vectors or string arrays.

Data Types: char | cell | string

Strike — Option strike price values
nonnegative integer

Option strike price value, specified as a NINST-by-1 or NINST-by-NSTRIKES depending on the type of option:

- European option — NINST-by-1 vector of strike price values.
- Bermuda option — NINST by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- American option — NINST-by-1 vector of strike price values for each option.

Data Types: double

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a NINST-by-1, NINST-by-2, or NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a NINST-by-1 vector, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

To support existing code, optbndbycir also accepts serial date numbers as inputs, but they are not recommended.

AmericanOpt — Option type
0 European/Bermuda (default) | integer with values 0 or 1

Option type, specified as NINST-by-1 positive integer flags with values:

- 0 — European/Bermuda
- 1 — American

Data Types: double

CouponRate — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an NINST-by-1 decimal annual rate or NINST-by-1 cell array, where each element is a NumDates-by-2 cell array. The first column of the NumDates-by-2 cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: double | cell

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optbndbycir` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbycir` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = optbndbycir(CIRTree,OptSpec, Strike,ExerciseDates,AmericanOpt,CouponRate,Settle,Maturity,'Period'6,'Basis' ,7,'Face',1000)`

`Period` — Coupons per year
2 per year (default) | possible values include: `0`, `1`, `2`, `3`, `4`, `6`, `12`.

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)

- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
date character vector | string array | datetime

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbycir` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbycir` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbycir` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbycir` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected prices of bond option at time `0`
matrix

Expected price of the bond option at time `0`, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure containing trees of vectors of instrument prices and accrued interest for each node
structure

Structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

- `PriceTree.tObs` contains the observation times.
- `PriceTree.PTree` contains the clean prices.
- `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it isn't.

More About

Bond Option

A bond option gives the holder 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.

Financial Instruments Toolbox 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.

For more information, see “Bond Options” on page 2-6.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optbndbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J., and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica.* Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice.* Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance.* CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling.* Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies.* Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir` | `instoptbnd`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Bond Options” on page 2-6
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

optbndbyhjm

Price bond option from Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree] = optbndbyhjm(HJMTree,OptSpec,Strike,ExerciseDates,
AmericanOpt,CouponRate,Settle,Maturity)
[Price,PriceTree] = optbndbyhjm( ___ ,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate,StartDate,Face,Options)
```

Description

`[Price,PriceTree] = optbndbyhjm(HJMTree,OptSpec,Strike,ExerciseDates, AmericanOpt,CouponRate,Settle,Maturity)` calculates the price for a bond option from a Black-Karasinski interest-rate tree.

`[Price,PriceTree] = optbndbyhjm( ___ ,Period,Basis,EndMonthRule,IssueDate, FirstCouponDate,LastCouponDate,StartDate,Face,Options)` adds optional arguments.

Examples

Price a European Call and Put Option on a Bond

Using the HJM forward-rate tree in the `deriv.mat` file, price a European call and put option on a 4% bond with a strike of 96. The exercise date for the option is Jan. 01, 2003. The settle date for the bond is Jan. 01, 2000, and the maturity date is Jan. 01, 2004.

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

```
load deriv.mat;
```

Use `optbndbyhjm` to compute the price of the `'Call'` option.

```
[Price,PriceTree] = optbndbyhjm(HJMTree,'Call',96,datetime(2003,1,1),...
0,0.04,datetime(2000,1,1),datetime(2004,1,1))
```

```
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In optbndbyhjm (line 223)

Price =

    2.2410


PriceTree =

  struct with fields:

    FinObj: 'HJMPriceTree'
```

```
      tObs: [0 1 2 3 4]
     PBush: {[2.2410]  [1×1×2 double]  [1×2×2 double]  [1×4×2 double]  [0 0 0 0 0 0 0 0]}
    ExBush: {[0]  [1×1×2 double]  [1×2×2 double]  [1×4×2 double]  [0 0 0 0 0 0 0 0]}
```

Now use `optbndbyhjm` to compute the price of a `'Put'` option on the same bond.

```
[Price,PriceTree] = optbndbyhjm(HJMTree,'Put',96,datetime(2003,1,1),...
0,0.04,datetime(2000,1,1),datetime(2004,1,1))
```

```
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In optbndbyhjm (line 223)

Price =

    0.0446


PriceTree =

  struct with fields:

    FinObj: 'HJMPriceTree'
      tObs: [0 1 2 3 4]
     PBush: {[0.0446]  [1×1×2 double]  [1×2×2 double]  [1×4×2 double]  [0 0 0 0 0 0 0 0]}
    ExBush: {[0]  [1×1×2 double]  [1×2×2 double]  [1×4×2 double]  [0 0 0 0 0 0 0 0]}
```

The `PriceTree.ExBush` output for the `'Call'` and `'Put'` option contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it is not.

Input Arguments

`HJMTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hjmtree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char`

`Strike` — Option strike price values

nonnegative integer

Option strike price value, specified as a `NINST`-by-`1` or `NINST`-by-`NSTRIKES` depending on the type of option:

- 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.

Data Types: double

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a NINST-by-1, NINST-by-2, or NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a NINST-by-1 vector, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

To support existing code, optbndbyhjm also accepts serial date numbers as inputs, but they are not recommended.

AmericanOpt — Option type
0 European/Bermuda (default) | integer with values 0 or 1

(Optional) Option type, specified as NINST-by-1 positive integer flags with values:

- 0 — European/Bermuda
- 1 — American

Data Types: double

CouponRate — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an NINST-by-1 decimal annual rate or NINST-by-1 cell array, where each element is a NumDates-by-2 cell array. The first column of the NumDates-by-2 cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: double | cell

Settle — Settlement date
datetime array | string array | date character vector

Settlement date for the bond option, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

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

To support existing code, optbndbyhjm also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`Period` — Coupons per year
2 per year (default) | vector

(Optional) Coupons per year, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis, specified as a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

(Optional) End-of-month rule flag is specified as a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: double

IssueDate — Bond issue date
datetime array | string array | date character vector

(Optional) Bond issue date, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optbndbyhjm also accepts serial date numbers as inputs, but they are not recommended.

FirstCouponDate — Irregular first coupon date
datetime array | string array | date character vector

(Optional) Irregular first coupon date, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optbndbyhjm also accepts serial date numbers as inputs, but they are not recommended.

When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

LastCouponDate — Irregular last coupon date
datetime array | string array | date character vector

(Optional) Irregular last coupon date, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optbndbyhjm also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

StartDate — Forward starting date of payments
datetime array | string array | date character vector

(Optional) Forward starting date of payments (the date from which a bond cash flow is considered), specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optbndbyhjm also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify StartDate, the effective start date is the Settle date.

Face — Face value
100 (default) | nonnegative value | cell array of nonnegative values

(Optional) Face or par value, specified as anNINST-by-1 vector.

Data Types: double

`Options` — Derivatives pricing options
structure

(Optional) Derivatives pricing options, specified as structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of bond option at time `0`
matrix

Expected price of the bond option at time `0`, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure containing trees of vectors of instrument prices and accrued interest for each node
structure

Structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

- `PriceTree.PBush` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.ExBush` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it isn't.

More About

Bond Option

A bond option gives the holder 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.

Financial Instruments Toolbox 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.

For more information, see “Bond Options” on page 2-6.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `optbndbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmprice` | `hjmtree` | `instoptbnd`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Bond Options” on page 2-6
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

optbndbyhw

Price bond option from Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = optbndbyhw(HWTree,OptSpec,Strike,ExerciseDates,
AmericanOpt,CouponRate,Settle,Maturity)
[Price,PriceTree] = optbndbyhw( ___ ,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate,StartDate,Face,Options)
```

Description

`[Price,PriceTree] = optbndbyhw(HWTree,OptSpec,Strike,ExerciseDates, AmericanOpt,CouponRate,Settle,Maturity)` calculates the price for a bond option from a Hull-White interest-rate tree.

Note Alternatively, you can use the `FixedBondOption` object to price fixed-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optbndbyhw( ___ ,Period,Basis,EndMonthRule,IssueDate, FirstCouponDate,LastCouponDate,StartDate,Face,Options)` adds optional arguments.

Examples

Price a European Call and Put Option on a Bond

Using the HW interest rate tree in the `deriv.mat` file, price a European call option on a 4% bond with a strike of 96. The exercise date for the option is Jan. 01, 2006. The settle date for the bond is Jan. 01, 2005, and the maturity date is Jan. 01, 2009.

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

```
load deriv.mat;
```

Use `optbndbyhw` to compute the price of the `'Call'` option.

```
[Price,PriceTree] = optbndbyhw(HWTree,'Call',96,datetime(2006,1,1),...
0,0.04,datetime(2005,1,1),datetime(2009,1,1))
```

```
Warning: OptBonds are valued at Tree ValuationDate rather than Settle.
> In optbndbytrintree (line 40)
  In optbndbyhw (line 92)
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In optbndbytrintree (line 151)
  In optbndbyhw (line 92)
```

```
Price =

    1.1556


PriceTree =

  struct with fields:

     FinObj: 'HWPriceTree'
      PTree: {[1.1556]  [0.0150 0.8509 3.7085]  [0 0 0.0722 4.9980 3.8429]  [0 0 0 0 0]  [0 0 0 0
       tObs: [0 1 2 3 4]
    Connect: {[2]  [2 3 4]  [2 2 3 4 4]}
      Probs: {[3×1 double]  [3×3 double]  [3×5 double]}
     ExTree: {[0]  [0 0 0]  [0 0 1 1 1]  [0 0 0 0 0]  [0 0 0 0 0]}
```

Now use `optbndbyhw` to compute the price of a `'Put'` option on the same bond.

```
[Price,PriceTree] = optbndbyhw(HWTree,'Put',96,datetime(2006,1,1),...
0,0.04,datetime(2005,1,1),datetime(2009,1,1))
```

```
Warning: OptBonds are valued at Tree ValuationDate rather than Settle.
> In optbndbytrintree (line 40)
  In optbndbyhw (line 92)
Warning: Not all cash flows are aligned with the tree. Result will be approximated.
> In optbndbytrintree (line 151)
  In optbndbyhw (line 92)

Price =

    1.0150


PriceTree =

  struct with fields:

     FinObj: 'HWPriceTree'
      PTree: {[1.0150]  [3.2945 0.7413 0]  [3.5551 4.6060 0 0 0]  [0 0 0 0 0]  [0 0 0 0 0]}
       tObs: [0 1 2 3 4]
    Connect: {[2]  [2 3 4]  [2 2 3 4 4]}
      Probs: {[3×1 double]  [3×3 double]  [3×5 double]}
     ExTree: {[0]  [0 0 0]  [1 1 0 0 0]  [0 0 0 0 0]  [0 0 0 0 0]}
```

The `PriceTree.ExTree` output for the `'Call'` and `'Put'` option contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it is not.

Input Arguments

`HWTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hwtree`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char`

`Strike` — Option strike price values
nonnegative integer

Option strike price value, specified as a `NINST`-by-`1` or `NINST`-by-`NSTRIKES` depending on the type of option:

- 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.

Data Types: `double`

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1`, `NINST`-by-`2`, or `NINST`-by-`NSTRIKES` vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a `NINST`-by-`1` vector of dates. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a `NINST`-by-`NSTRIKES` vector of dates.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-`1` vector, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `optbndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
`0` European/Bermuda (default) | integer with values `0` or `1`

Option type, specified as `NINST`-by-`1` positive integer flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`CouponRate` — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates

and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optbndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Period` — Coupons per year
2 per year (default) | vector

(Optional) Coupons per year, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis, specified as a `NINST`-by-`1` vector of integers.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

(Optional) End-of-month rule flag is specified as a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

(Optional) Bond issue date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

(Optional) Irregular first coupon date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

(Optional) Irregular last coupon date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

(Optional) Forward starting date of payments (the date from which a bond cash flow is considered), specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optbndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | nonnegative value | cell array of nonnegative values

(Optional) Face or par value, specified as an`NINST`-by-`1` vector.

Data Types: `double`

`Options` — Derivatives pricing options
structure

(Optional) Derivatives pricing options, specified as structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of bond option at time `0`
matrix

Expected price of the bond option at time `0`, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure containing trees of vectors of instrument prices and accrued interest for each node
structure

Structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.
- `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it isn't.

More About

Bond Option

A bond option gives the holder 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.

Financial Instruments Toolbox 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.

For more information, see “Bond Options” on page 2-6.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optbndbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hwprice` | `hwtree` | `instoptbnd` | `FixedBondOption`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond Options” on page 2-6
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optByBatesFD

Option price by Bates model using finite differences

Syntax

```
[Price,PriceGrid,AssetPrices,Variances,Times] = optByBatesFD(Rate,AssetPrice,
Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,
JumpVol,JumpFreq)
[Price,PriceGrid,AssetPrices,Variances,Times] = optByBatesFD( ___ ,Name,Value)
```

Description

`[Price,PriceGrid,AssetPrices,Variances,Times] = optByBatesFD(Rate,AssetPrice, Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ, JumpVol,JumpFreq)` computes a vanilla European or American option price by the Bates model, using the alternating direction implicit (ADI) method.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceGrid,AssetPrices,Variances,Times] = optByBatesFD( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Price an American Option Using the Bates Model

Define the option variables and Bates model parameters.

```
AssetPrice = 90;
Strike = 100;
Rate = 0.03;
Settle = datetime(2018,1,10);
ExerciseDates = datetime(2018,7,2);

V0 = 0.04;
ThetaV = 0.04;
Kappa = 2;
SigmaV = 0.25;
RhoSV = -0.5;
JumpVol = 0.4;
MeanJ = exp(-0.5+JumpVol.^2/2)-1;
JumpFreq = 0.2;
```

Compute the American put option price using the finite differences method.

```
OptSpec = 'Put';
Price = optByBatesFD(Rate, AssetPrice, Settle, ExerciseDates, OptSpec, Strike, ...
V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, 'AmericanOpt', 1)
```

```
Price = 11.4925
```

Input Arguments

`Rate` — Continuously compounded risk-free interest rate

scalar decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal.

Data Types: `double`

`AssetPrice` — Current underlying asset price

scalar numeric

Current underlying asset price, specified as a scalar numeric.

Data Types: `double`

`Settle` — Option settlement date

datetime scalar | string scalar | date character vector

Option settlement date, specified as a scalar datetime, string, or data character vector.

To support existing code, `optByBatesFD` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a scalar datetime, string, or data character vector. For a European option, `ExerciseDates` contains only one value: the option expiry date.
- For an American option, use a 1-by-2 vector of dates to specify the exercise date boundaries. An American option can be exercised on any date between or including the pair of dates. If only one non-`NaN` date is listed, then the option can be exercised between `Settle` date and the single listed value in `ExerciseDates`.

To support existing code, `optByBatesFD` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option

character vector with value of `'call'` or `'put'` | string with value of `"call"` or `"put"`

Definition of the option, specified as a scalar using a character vector or string with a value of `'call'` or `'put'`.

Data Types: `cell` | `string`

`Strike` — Option strike price value

scalar numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

`V0` — Initial variance of underlying asset
scalar numeric

Initial variance of the underling asset, specified as a scalar numeric.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
scalar numeric

Long-term variance of the underling asset, specified as a scalar numeric.

Data Types: `double`

`Kappa` — Mean revision speed for the variance of underlying asset
scalar numeric

Mean revision speed for the underlying asset, specified as a scalar numeric.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
scalar numeric

Volatility of the variance of the underling asset, specified as a scalar numeric.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
scalar numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric.

Data Types: `double`

`MeanJ` — Mean of random percentage jump size
scalar decimal

Mean of the random percentage jump size (J), specified as a scalar decimal where `log`(1+J) is normally distributed with the mean (`log`(1+`MeanJ`)-0.5*`JumpVol^`2) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log`(1+J)
scalar decimal

Standard deviation of `log`(1+J) where `J` is the random percentage jump size, specified as a scalar decimal.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
scalar numeric

Annual frequency of the Poisson jump process, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceGrid] = optByBatesFD(Rate,AssetPrice,Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq,'Basis',7)`

`Basis` — Day-count basis of instrument

`0` (default) | numeric values: `0`,1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield

`0` (default) | scalar numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric.

Note If you enter a value for `DividendYield`, then set `DividendAmounts` and `ExDividendDates` = `[ ]` or do not enter them. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield` = `0`.

Data Types: double

DividendAmounts — Cash dividend amounts
[ ] (default) | vector

Cash dividend amounts, specified as the comma-separated pair consisting of 'DividendAmounts' and an NDIV-by-1 vector.

Note Each dividend amount must have a corresponding ex-dividend date. If you enter values for DividendAmounts and ExDividendDates, then set DividendYield = 0.

Data Types: double

ExDividendDates — Ex-dividend dates
[ ] (default) | datetime array | string array | date character vector

Ex-dividend dates, specified as the comma-separated pair consisting of 'ExDividendDates' and an NDIV-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optByBatesFD also accepts serial date numbers as inputs, but they are not recommended.

AssetPriceMax — Maximum price for price grid boundary
if unspecified, value is calculated based on asset price distribution at maturity (default) | positive scalar numeric

Maximum price for the price grid boundary, specified as the comma-separated pair consisting of 'AssetPriceMax' and a positive scalar numeric.

Data Types: double

VarianceMax — Maximum variance for variance grid boundary
1.0 (default) | scalar numeric

Maximum variance for the variance grid boundary, specified as the comma-separated pair consisting of 'VarianceMax' as a scalar numeric.

Data Types: double

AssetGridSize — Size of asset grid for finite difference grid
400 (default) | scalar numeric

Size of the asset grid for finite difference grid, specified as the comma-separated pair consisting of 'AssetGridSize' and a scalar numeric.

Data Types: double

VarianceGridSize — Number of nodes of variance grid for finite difference grid
200 (default) | scalar numeric

Number of nodes of the variance grid for the finite difference grid, specified as the comma-separated pair consisting of 'VarianceGridSize' and a scalar numeric.

Data Types: double

`TimeGridSize` — Number of nodes of time grid for finite difference grid
`100` (default) | positive numeric scalar

Number of nodes of the time grid for the finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive numeric scalar.

Data Types: `double`

`AmericanOpt` — Option type
`0` (European) (default) | scalar with value of `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar flag with one of these values:

- `0` — European
- `1` — American

Data Types: `double`

Output Arguments

`Price` — Option price
scalar numeric

Option price, returned as a scalar numeric.

`PriceGrid` — Grid containing prices calculated by the finite difference method
grid

Grid containing prices calculated by the finite difference method, returned as a two-dimensional grid with size `AssetGridSize` × `TimeGridSize`. The number of columns is not necessarily equal to the `TimeGridSize` because exercise the function adds exercise and ex-dividend dates to the time grid. `PriceGrid(:, :, end)` contains the price for $t = 0$.

`AssetPrices` — Prices of the asset
vector

Prices of the asset corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Variances` — Variances
vector

Variances corresponding to the second dimension of `PriceGrid`, returned as a vector.

`Times` — Times
vector

Times corresponding to the third dimension of `PriceGrid`, returned as a vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Bates Stochastic Volatility Jump Diffusion Model

The Bates model [1] extends the Heston model by including stochastic volatility and (similar to Merton) jump diffusion parameters in the modeling of sudden asset price movements.

The stochastic differential equation is:

$$dS_t = (r - q - \lambda_p \mu_J)S_t dt + \sqrt{v_t} S_t dW_t + JS_t dP_t$$
$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t$$
$$\mathrm{E}\left[dW_t dW_t^v\right] = pdt$$
$$\mathrm{prob}(dP_t = 1) = \lambda_p dt$$

where:

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1 + J)\delta\sqrt{2\pi}} \exp\left\{ \frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2} \right\}$$

where:

v_0 is the initial variance of the asset price at $t = 0$ ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for ($\kappa > 0$).

σ_v is the volatility of variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and W_t^v for ($-1 \le p \le 1$).

μ_J is the mean of J for ($\mu_J > -1$).

δ is the standard deviation of $\ln(1+J)$ for ($\delta \ge 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \ge 0$).

Version History

Introduced in R2019a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optByBatesFD` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bates, D. S. "Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options." *The Review of Financial Studies.* Vol. 9, Number 1, 1996.

See Also

`optByLocalVolFD` | `optSensByLocalVolFD` | `optByHestonFD` | `optSensByHestonFD` | `optSensByBatesFD` | `optByMertonFD` | `optSensByMertonFD` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optSensByBatesFD

Option price and sensitivities by Bates model using finite differences

Syntax

```
[PriceSens,PriceGrid,AssetPrices,Variances,Times] = optSensByBatesFD(Rate,
AssetPrice,Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,
MeanJ,JumpVol,JumpFreq)
[PriceSens,PriceGrid,AssetPrices,Variances,Times] = optSensByBatesFD( ___ ,
Name,Value)
```

Description

`[PriceSens,PriceGrid,AssetPrices,Variances,Times] = optSensByBatesFD(Rate, AssetPrice,Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV, MeanJ,JumpVol,JumpFreq)` computes a vanilla European or American option price and sensitivities by the Bates model, using the alternating direction implicit (ADI) method.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,PriceGrid,AssetPrices,Variances,Times] = optSensByBatesFD( ___ , Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute Price and Sensitivities for an American Option Using the Bates Model

Define the option variables and Bates model parameters.

```
AssetPrice = 90;
Strike = 100;
Rate = 0.03;
Settle = datetime(2018,1,1);
ExerciseDates = datetime(2018,6,2);

V0 = 0.04;
ThetaV = 0.04;
Kappa = 2;
SigmaV = 0.25;
RhoSV = -0.5;
JumpVol = 0.4;
MeanJ = exp(-0.5+JumpVol.^2/2)-1;
JumpFreq = 0.2;
```

Compute the American put option price and sensitivities using the finite differences method.

```
OptSpec = 'Put';
[Price, Delta, Gamma, Rho, Theta, Vega, VegaLT] = optSensByBatesFD(Rate, AssetPrice, Settle, Exe
OptSpec, Strike, V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, 'AmericanOpt', 1,..
'OutSpec', ["Price" "Delta" "Gamma" "Rho" "Theta" "Vega" "VegaLT"])

Price = 11.2164

Delta = -0.6988

Gamma = 0.0391

Rho = -17.1376

Theta = -4.7656

Vega = 13.3283

VegaLT = 6.1456
```

Input Arguments

`Rate` — Continuously compounded risk-free interest rate

scalar decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal.

Data Types: `double`

`AssetPrice` — Current underlying asset price

scalar numeric

Current underlying asset price, specified as a scalar numeric.

Data Types: `double`

`Settle` — Option settlement date

datetime scalar | string scalar | date character vector

Option settlement date, specified as a scalar datetime, string, or data character vector.

To support existing code, `optSensByBatesFD` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a scalar date. For a European option, `ExerciseDates` contains only one value: the option expiry date.
- For an American option, use a 1-by-2 vector of dates to specify the exercise date boundaries. An American option can be exercised on any date between or including the pair of dates. If only one non-`NaN` date is listed, then the option can be exercised between `Settle` date and the single listed value in `ExerciseDates`.

To support existing code, `optSensByBatesFD` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with value of `'call'` or `'put'` | string with value of `"call"` or `"put"`

Definition of the option, specified as a scalar using a character vector or string with a value of `'call'` or `'put'`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

`V0` — Initial variance of underlying asset
scalar numeric

Initial variance of the underling asset, specified as a scalar numeric.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
scalar numeric

Long-term variance of the underling asset, specified as a scalar numeric.

Data Types: `double`

`Kappa` — Mean revision speed for the variance of underlying asset
scalar numeric

Mean revision speed for the underlying asset, specified as a scalar numeric.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
scalar numeric

Volatility of the variance of the underling asset, specified as a scalar numeric.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
scalar numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric.

Data Types: `double`

`MeanJ` — Mean of random percentage jump size
scalar decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal where `log`(1+*J*) is normally distributed with the mean (`log`(1+`MeanJ`)-0.5*`JumpVol^`2) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log(1+J)`
scalar decimal

Standard deviation of `log(1+J)` where `J` is the random percentage jump size, specified as a scalar decimal.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
scalar numeric

Annual frequency of the Poisson jump process, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PriceSens,PriceGrid] = optSensByBatesFD(Rate,AssetPrice,Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq,'Basis',7,'OutSpec','delta')`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

DividendYield — Continuously compounded underlying asset yield
0 (default) | scalar numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of 'DividendYield' and a scalar numeric.

Note If you enter a value for DividendYield, then set DividendAmounts and ExDividendDates = [ ] or do not enter them. If you enter values for DividendAmounts and ExDividendDates, then set DividendYield = 0.

Data Types: double

DividendAmounts — Cash dividend amounts
[ ] (default) | vector

Cash dividend amounts, specified as the comma-separated pair consisting of 'DividendAmounts' and an NDIV-by-1 vector.

Note Each dividend amount must have a corresponding ex-dividend date. If you enter values for DividendAmounts and ExDividendDates, then set DividendYield = 0.

Data Types: double

ExDividendDates — Ex-dividend dates
[ ] (default) | datetime array | string array | date character vector

Ex-dividend dates, specified as the comma-separated pair consisting of 'ExDividendDates' and an NDIV-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optSensByBatesFD also accepts serial date numbers as inputs, but they are not recommended.

AssetPriceMax — Maximum price for price grid boundary
if unspecified, value is calculated based on asset price distribution at maturity (default) | positive scalar numeric

Maximum price for the price grid boundary, specified as the comma-separated pair consisting of 'AssetPriceMax' and a positive scalar numeric.

Data Types: double

VarianceMax — Maximum variance for variance grid boundary
1.0 (default) | scalar numeric

Maximum variance for the variance grid boundary, specified as the comma-separated pair consisting of 'VarianceMax' as a scalar numeric.

Data Types: double

AssetGridSize — Size of asset grid for finite difference grid
400 (default) | scalar numeric

Size of the asset grid for finite difference grid, specified as the comma-separated pair consisting of `'AssetGridSize'` and a scalar numeric.

Data Types: `double`

`VarianceGridSize` — Number of nodes of variance grid for finite difference grid
`200` (default) | scalar numeric

Number of nodes of the variance grid for the finite difference grid, specified as the comma-separated pair consisting of `'VarianceGridSize'` and a scalar numeric.

Data Types: `double`

`TimeGridSize` — Number of nodes of time grid for finite difference grid
`100` (default) | positive numeric scalar

Number of nodes of the time grid for the finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive numeric scalar.

Data Types: `double`

`AmericanOpt` — Option type
`0` (European) (default) | scalar with value of `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar flag with one of these values:

- `0` — European
- `1` — American

Data Types: `double`

`OutSpec` — Define outputs
`['price']` (default) | cell array of character vectors with values `'price'`, `'delta'`, `'gamma'`, `'vega'`, `'rho'`, `'theta'`, and `'vegalt'` | string array with values `"price"`, `"delta"`, `"gamma"`, `"vega"`, `"rho"`, `"theta"`, and `"vegalt"`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` string array or cell array of character vectors with supported values.

Note `'vega'` is the sensitivity with respect to the initial volatility sqrt(`V0`). In contrast, `'vegalt'` is the sensitivity with respect to the long-term volatility sqrt(`ThetaV`).

Example: `OutSpec = ['price','delta','gamma','vega','rho','theta','vegalt']`

Data Types: `string` | `cell`

Output Arguments

`PriceSens` — Option price or sensitivities
numeric

Option price or sensitivities, returned as a numeric. The name-value pair argument `OutSpec` determines the types and order of the outputs.

`PriceGrid` — Grid containing prices calculated by the finite difference method
grid

Grid containing prices calculated by the finite difference method, returned as a two-dimensional grid with size `AssetGridSize` × `TimeGridSize`. The number of columns is not necessarily equal to the `TimeGridSize` because the function adds exercise and ex-dividend dates to the time grid. `PriceGrid(:, :, end)` contains the price for $t = 0$.

`AssetPrices` — Prices of the asset
vector

Prices of the asset corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Variances` — Variances
vector

Variances corresponding to the second dimension of `PriceGrid`, returned as a vector.

`Times` — Times
vector

Times corresponding to the third dimension of `PriceGrid`, returned as a vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Bates Stochastic Volatility Jump Diffusion Model

The Bates model [1] extends the Heston model by including stochastic volatility and (similar to Merton) jump diffusion parameters in the modeling of sudden asset price movements.

The stochastic differential equation is

$$dS_t = (r - q - \lambda_p \mu_J) S_t dt + \sqrt{v_t} S_t dW_t + J S_t dP_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$

$$\mathrm{prob}(dP_t = 1) = \lambda_p dt$$

where:

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

J is the random percentage jump size conditional on the jump occurring, where `ln`(1+J) is normally distributed with mean $\ln(1+\mu_J)-\frac{\delta^2}{2}$ and the standard deviation δ, and (1+J) has a lognormal distribution:

$$\frac{1}{(1+J)\delta\sqrt{2\pi}}\exp\left\{\frac{-\left[\ln(1+J)-\left(\ln(1+\mu_J)-\frac{\delta^2}{2}\right]\right]^2}{2\delta^2}\right\}$$

where:

v_0 is the initial variance of the asset price at $t = 0$ ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for ($\kappa > 0$).

σ_v is the volatility of variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and W_t^v for ($-1 \le p \le 1$).

μ_J is the mean of J for ($\mu_J > -1$).

δ is the standard deviation of `ln`(1+J) for ($\delta \ge 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \ge 0$).

Version History

Introduced in R2019a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optSensByBatesFD` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bates, D. S. "Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options." *The Review of Financial Studies.* Vol. 9, Number 1, 1996.

See Also

`optByLocalVolFD` | `optSensByLocalVolFD` | `optByHestonFD` | `optSensByHestonFD` | `optByBatesFD` | `optByMertonFD` | `optSensByMertonFD` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optByBatesFFT

Option price by Bates model using FFT and FRFT

Syntax

```
[Price,StrikeOut] = optByBatesFFT(Rate,AssetPrice,Settle,Maturity,OptSpec,
Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq)
[Price,StrikeOut] = optByBatesFFT( ___ ,Name,Value)
```

Description

`[Price,StrikeOut] = optByBatesFFT(Rate,AssetPrice,Settle,Maturity,OptSpec, Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq)` computes vanilla European option price by Bates model, using Carr-Madan FFT and Chourdakis FRFT methods.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,StrikeOut] = optByBatesFFT( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Price Surface Using the Bates Model

Use `optByBatesFFT` to calibrate the FFT strike grid, compute option prices, and plot an option price surface.

Define Option Variables and Bates Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

V0 = 0.04;
ThetaV = 0.05;
Kappa = 1.0;
SigmaV = 0.2;
RhoSV = -0.7;
MeanJ = 0.02;
JumpVol = 0.08;
JumpFreq = 2;
```

Compute Option Prices for the Entire FFT (or FRFT) Strike Grid, Without Specifying `Strike`

Compute option prices and also output the corresponding strikes. If the `Strike` input is empty (`[]`), option prices will be computed on the entire FFT (or FRFT) strike grid. The FFT (or FRFT) strike grid

is determined as `exp(log-strike grid)`, where each column of the log-strike grid has `NumFFT` points with `LogStrikeStep` spacing that are roughly centered around each element of `log(AssetPrice)`. The default value for `NumFFT` is `2^12`. In addition to the prices in the first output, the optional last output contains the corresponding strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = []; % Strike is not specified (will use the entire FFT strike grid)

% Compute option prices for the entire FFT strike grid
[Call, Kout] = optByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield);

% Show the lowest and highest strike values on the FFT strike grid
format
MinStrike = Kout(1) % Lowest possible strike in the current FFT strike grid

MinStrike = 2.9205e-135

MaxStrike = Kout(end) % Highest possible strike in the current FFT strike grid

MaxStrike = 1.8798e+138


% Show a subset of the strikes and corresponding option prices
Range = (2046:2052);
[Kout(Range) Call(Range)]

ans = 7×2

   50.4929   29.4990
   58.8640   21.4545
   68.6231   12.8544
   80.0000    5.3484
   93.2631    1.2404
  108.7251    0.1648
  126.7505    0.0152
```

Change the Number of FFT (or FRFT) Points and Compare with optByBatesNI

Try a different number of FFT (or FRFT) points, and compare the results with direct numerical integration. Unlike `optByBatesFFT`, which uses FFT (or FRFT) techniques for fast computation across the whole range of strikes, the `optByBatesNI` function uses direct numerical integration and it is typically slower, especially for multiple strikes. However, the values computed by `optByBatesNI` can serve as a benchmark for adjusting the settings for `optByBatesFFT`.

```
% Try a smaller number of FFT (or FRFT) points
% (e.g. for faster performance or smaller memory footprint)
NumFFT = 2^10; % Smaller than the default value of 2^12
Strike = []; % Strike is not specified (will use the entire FFT strike grid)
[Call, Kout] = optByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'NumFFT', NumFFT);

% Compare with numerical integration method
Range = (510:516);
```

```
Strike = Kout(Range);
CallFFT = Call(Range);
CallNI = optByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield);
Error = abs(CallFFT-CallNI);
table(Strike, CallFFT, CallNI, Error)
```

```
ans=7×4 table
    Strike     CallFFT       CallNI         Error
    ______    _________    ___________    _________

    12.696       66.237         66.696      0.45912
    23.449        55.86         56.103      0.24239
    43.312       36.418         36.541      0.12246
        80       5.4029         5.3484     0.054469
    147.76     0.044921      0.0010864     0.043835
    272.93    0.0094655    -7.8249e-08    0.0094656
    504.11    0.0024986    -3.3873e-07    0.0024989
```

Make Further Adjustments to FFT (or FRFT)

If the values in the output `CallFFT` are significantly different from those in `CallNI`, try making adjustments to `optByBatesFFT` settings, such as `CharacteristicFcnStep`, `LogStrikeStep`, `NumFFT`, `DampingFactor`, and so on. Note that if (`LogStrikeStep` * `CharacteristicFcnStep`) is 2*pi / `NumFFT`, FFT is used. Otherwise, FRFT is used.

```
Strike = []; % Strike is not specified (will use the entire FFT or FRFT strike grid)
[Call, Kout] = optByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001);

% Compare with numerical integration method
Strike = Kout(Range);
CallFFT = Call(Range);
CallNI = optByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield);
Error = abs(CallFFT-CallNI);
table(Strike, CallFFT, CallNI, Error)
```

```
ans=7×4 table
    Strike    CallFFT    CallNI      Error
    ______    _______    ______    __________

    79.76     5.4682     5.4682    1.5355e-08
    79.84     5.4281     5.4281    1.4833e-08
    79.92     5.3882     5.3882    1.4244e-08
       80     5.3484     5.3484     1.359e-08
    80.08     5.3088     5.3088    1.2875e-08
    80.16     5.2693     5.2693    1.2101e-08
    80.24       5.23       5.23    1.1272e-08
```

```
% Save the final FFT (or FRFT) strike grid for future reference. For
% example, it provides information about the range of |Strike| inputs for
% which the FFT (or FRFT) operation is valid.
```

```
FFTStrikeGrid = Kout;
MinStrike = FFTStrikeGrid(1) % Strike cannot be less than MinStrike
```

```
MinStrike = 47.9437
```

```
MaxStrike = FFTStrikeGrid(end) % Strike cannot be greater than MaxStrike
```

```
MaxStrike = 133.3566
```

Compute the Option Price for a Single Strike

Once the desired FFT (or FRFT) settings are determined, use the `Strike` input to specify the strikes (rather than providing an empty array). If the specified strikes do not match a value on the FFT (or FRFT) strike grid, the outputs are interpolated on the specified strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;

Call = optByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV,  MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001)
```

```
Call = 5.3484
```

Compute the Option Prices for a Vector of Strikes

Use the `Strike` input to specify the strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Call = optByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV,  MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001)
```

```
Call = 5×1

    7.5765
    6.4020
    5.3484
    4.4173
    3.6073
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes
```

```
Call = optByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV,  MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001) % Five values in vector output
```

```
Call = 5×1

    9.7516
   10.3931
   10.8865
   11.2990
   11.6491
```

Expand the Outputs for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the outputs into `NStrikes`-by-`NMaturities` matrices. In this case, they are square matrices.

```
[Call, Kout] = optByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV,  MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001, 'ExpandOutput', true) % (5 x 5) matrix output
```

```
Call = 5×5

    9.7516   11.4387   12.8395   14.0588   15.1361
    8.6554   10.3931   11.8344   13.0890   14.1980
    7.6432    9.4149   10.8865   12.1693   13.3046
    6.7153    8.5035    9.9952   11.2990   12.4553
    5.8705    7.6581    9.1594   10.4771   11.6491


Kout = 5×5

    76    76    76    76    76
    78    78    78    78    78
    80    80    80    80    80
    82    82    82    82    82
    84    84    84    84    84
```

Compute Option Prices for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities` (that is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular).

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Call = optByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV,  MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001, 'ExpandOutput', true) % (5 x 6) matrix output
```

```
Call = 5×6

    7.5765    9.7516   11.4387   12.8395   14.0588   15.1361
```

```
    6.4020    8.6554   10.3931   11.8344   13.0890   14.1980
    5.3484    7.6432    9.4149   10.8865   12.1693   13.3046
    4.4173    6.7153    8.5035    9.9952   11.2990   12.4553
    3.6073    5.8705    7.6581    9.1594   10.4771   11.6491
```

Compute the Option Prices for a Vector of Strikes and a Vector of Asset Prices

When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

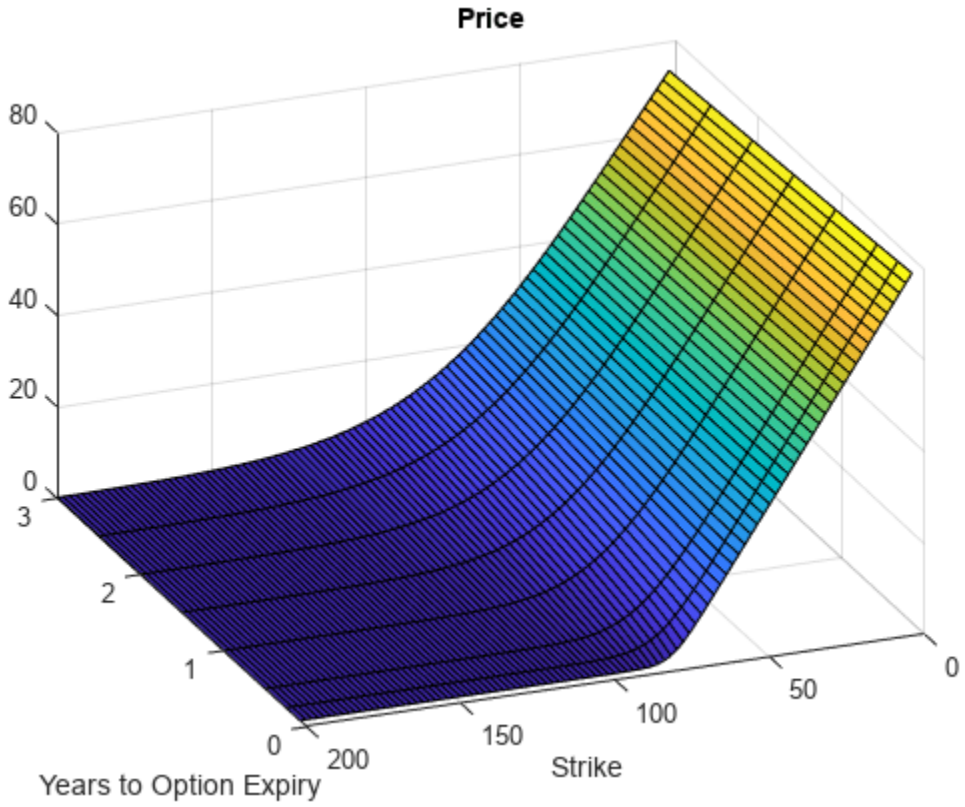
```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Call = optByBatesFFT(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV,  MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001, 'ExpandOutput', true) % (5 x 4) matrix output
```

```
Call = 5×4

    4.2033    6.6918    9.7516   13.2808
    3.5558    5.8112    8.6554   11.9993
    2.9906    5.0181    7.6432   10.7934
    2.5018    4.3096    6.7153    9.6652
    2.0825    3.6818    5.8705    8.6158
```

Plot an Option Price Surface

Use the `Strike` input to specify the strikes. Increase the value for `NumFFT` to support a wider range of strikes. Also, the `Maturity` input can be a vector. Set `ExpandOutput` to `"true"` to output the surface as a `NStrikes`-by-`NMaturities` matrix.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

% Increase |NumFFT| to support a wider range of strikes
NumFFT = 2^13;

Call = optByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV,  MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001, 'ExpandOutput', true);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Call);
title('Price');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
zlim([0 80]);
```



Input Arguments

`Rate` — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

numeric

Current underlying asset price, specified as numeric value using a scalar or a `NINST`-by-`1` or `NColumns`-by-`1` vector.

For more information on the proper dimensions for `AssetPrice`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Settle` — Option settlement date

datetime array | string array | date character vector

Option settlement date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `optByBatesFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Settle`, see the name-value pair argument `ExpandOutput`.

`Maturity` — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optByBatesFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Maturity`, see the name-value pair argument `ExpandOutput`.

`OptSpec` — Definition of option
cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

For more information on the proper dimensions for `OptSpec`, see the name-value pair argument `ExpandOutput`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a `NINST`-by-`1`, `NRows`-by-`1`, `NRows`-by-`NColumns` vector of strike prices.

If this input is an empty array (`[]`), option prices are computed on the entire FFT (or FRFT) strike grid, which is determined as `exp(log-strike grid)`. Each column of the log-strike grid has `'NumFFT'` points with `'LogStrikeStep'` spacing that are roughly centered around each element of `log(AssetPrice)`.

For more information on the proper dimensions for `Strike`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`V0` — Initial variance of underlying asset
numeric

Initial variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
numeric

Long-term variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

Kappa — Mean revision speed for the variance of underlying asset
numeric

Mean revision speed for the underling asset, specified as a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
numeric

Volatility of the variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric value.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal value where `log`(1+*J*) is normally distributed with mean (`log`(1+`MeanJ`)-0.5*`JumpVol^`2) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log`(1+*J*)
decimal

Standard deviation of `log`(1+*J*) where `J` is the random percentage jump size, specified as a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
numeric

Annual frequency of Poisson jump process, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price, StrikeOut] = optByBatesFFT(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,V0,ThetaV,Kappa, SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq,'Basis',7)`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric value.

Data Types: `double`

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, specified as the comma-separated pair consisting of `'VolRiskPremium'` and a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating Little Heston Trap formulation
`true` (default) | logical with values `true` or `false`

Flag indicating Little Heston Trap formulation by Albrecher *et al*, specified as the comma-separated pair consisting of `'LittleTrap'` and a logical:

- `true` — Use the Albrecher *et al* formulation.
- `false` — Use the original Heston formation.

Data Types: `logical`

`NumFFT` — Number of grid points in the characteristic function variable
`4096` (default) | numeric

Number of grid points in the characteristic function variable and in each column of the log-strike grid, specified as the comma-separated pair consisting of `'NumFFT'` and a scalar numeric value.

Data Types: `double`

`CharacteristicFcnStep` — Characteristic function variable grid spacing
`0.01` (default) | numeric

Characteristic function variable grid spacing, specified as the comma-separated pair consisting of `'CharacteristicFcnStep'` and a scalar numeric value.

Data Types: `double`

`LogStrikeStep` — Log-strike grid spacing
`2*pi/NumFFT/CharacteristicFcnStep` (default) | numeric

Log-strike grid spacing, specified as the comma-separated pair consisting of `'LogStrikeStep'` and a scalar numeric value.

Note If (`LogStrikeStep*CharacteristicFcnStep`) is `2*pi/NumFFT`, FFT is used. Otherwise, FRFT is used.

Data Types: `double`

`DampingFactor` — Damping factor for Carr-Madan formulation
`1.5` (default) | numeric

Damping factor for Carr-Madan formulation, specified as the comma-separated pair consisting of `'DampingFactor'` and a scalar numeric value.

Data Types: `double`

`Quadrature` — Type of quadrature
`"simpson"` (default) | character vector with values: `'simpson'` or `'trapezoidal'` | string array with values: `"simpson"` or `"trapezoidal"`

Type of quadrature, specified as the comma-separated pair consisting of `'Quadrature'` and a single character vector or string array with a value of `'simpson'` or `'trapezoidal'`.

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-`1` vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-`1` vector, or a `NRows`-by-`NColumns` matrix. If `Strike` is empty, `NRows` is equal to `NumFFT`. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-`1` vectors.

- `false` — If `false`, the outputs are `NINST`-by-`1` vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-`1` vectors.

Data Types: `logical`

Output Arguments

`Price` — Option prices

numeric

Option prices, returned as a `NINST`-by-`1`, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

`StrikeOut` — Strikes corresponding to `Price`

numeric

Strikes corresponding to `Price`, returned as a `NINST`-by-`1`, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Bates Stochastic Volatility Jump Diffusion Model

The Bates model (Bates (1996)) is an extension of the Heston model, where, in addition to stochastic volatility, the jump diffusion parameters similar to Merton (1976) were also added to model sudden asset price movements.

The stochastic differential equation is:

$$dS_t = (r - q - \lambda_p \mu_J) S_t dt + \sqrt{v_t} S_t dW_t + J S_t dP_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$

$$\mathrm{prob}(dP_t = 1) = \lambda_p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

J is the random percentage jump size conditional on the jump occurring, where `ln`(1+J) is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and (1+J) has a lognormal distribution:

$$\frac{1}{(1 + J)\delta\sqrt{2\pi}}\exp\left\{\frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2}\right\}$$

v_0 is the initial variance of the asset price at $t = 0$ ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for ($\kappa > 0$).

σ_v is the volatility of variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and W_t^v for ($-1 \le p \le 1$).

μ_J is the mean of J for ($\mu_J > -1$).

δ is the standard deviation of `ln`(1+J) for ($\delta \ge 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \ge 0$).

The characteristic function $f_{Bates_j(\phi)}$ for $j = 1$ (asset price mean measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Bates(\phi)} = \exp(C_j + D_j v_0 + i\phi \ln S_t)\exp\left(\lambda_p \tau(1+\mu_J)^{m_j + \frac{1}{2}}\left[(1+\mu_j)^{i\phi} e^{\delta^2 (m_j i\phi + \frac{(i\phi)^2}{2})} - 1\right] - \lambda_p \tau \mu_j i\phi\right)$$

$$m_j = \begin{Bmatrix} m_1 = \frac{1}{2} \\ m_2 = -\frac{1}{2} \end{Bmatrix}$$

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1 - g_j e^{d_j \tau}}{1 - g_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1 - e^{d_j \tau}}{1 - g_j e^{d_j \tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

where

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity for ($\tau = T - t$).

i is the unit imaginary number for ($i^2 = -1$).

The definitions for C_j and D_j under "The Little Heston Trap" by Albrecher et al. (2007) are:

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1 - \varepsilon_j e^{-d_j \tau}}{1 - \varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1 - e^{-d_j \tau}}{1 - \varepsilon_j e^{-d_j \tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Carr-Madan Formulation

The Carr and Madan (1999) formulation is a popular modified implementation of Heston (1993) framework.

Rather than computing the probabilities P_1 and P_2 as intermediate steps, Carr and Madan developed an alternative expression so that taking its inverse Fourier transform gives the option price itself directly.

$$Call(k) = \frac{e^{-\alpha k}}{\pi}\int_0^{\infty} \mathrm{Re}\left[e^{-iuk}\psi(u)\right]du$$

$$\psi(u) = \frac{e^{-r\tau}f_2(\phi = (u-(\alpha+1)i))}{\alpha^2+\alpha-u^2+iu(2\alpha+1)}$$

$$Put(K) = Call(K) + Ke^{-r\tau} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K

i is a unit imaginary number (i^2= -1)

ϕ is the characteristic function variable.

α is the damping factor.

u is the characteristic function variable for integration, where ϕ = (u - (α+1)i).

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

To apply FFT or FRFT to this formulation, the characteristic function variable for integration, u, is discretized into `NumFFT`(N) points with the step size `CharacteristicFcnStep` (Δu), and the log-strike k is discretized into N points with the step size `LogStrikeStep`(Δk).

The discretized characteristic function variable for integration, u_j(for j = 1,2,3,...,N), has a minimum value of 0 and a maximum value of (N-1) (Δu), and it approximates the continuous integration range from 0 to infinity.

The discretized log-strike grid, k_n(for n = 1, 2, 3, N) is approximately centered around `ln`(S_t), with a minimum value of

$$\ln(S_t) - \frac{N}{2}\Delta k$$

and a maximum value of

$$\ln(S_t) + \left(\frac{N}{2} - 1\right)\Delta k$$

Where the minimum allowable strike is

$$S_t \exp\left(-\frac{N}{2}\Delta k\right)$$

and the maximum allowable strike is

$$S_t \exp\left[\left(\frac{N}{2} - 1\right)\Delta k\right]$$

As a result of the discretization, the expression for the call option becomes

$$Call(k_n) = \Delta u \frac{e^{-\alpha k_n}}{\pi} \sum_{j=1}^{N} \mathrm{Re}\left[e^{-i\Delta k \Delta u (j-1)(n-1)} e^{iuj\left[\frac{N\Delta k}{2} - \ln(S_t)\right]} \psi(u_j)\right] w_j$$

where

Δu is the step size of discretized characteristic function variable for integration.

Δk is the step size of discretized log-strike.

N is the number of FFT/FRFT points

w_j is the weights for quadrature used for approximating the integral.

FFT is used to evaluate the above expression if Δk and Δu are subject to the following constraint:

$$\Delta k \Delta u = \left(\frac{2\pi}{N}\right)$$

otherwise, the functions use the FRFT method described in Chourdakis (2005).

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optByBatesFFT` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Albrecher, H., Mayer, P., Schoutens, W., and Tistaert, J. "The Little Heston Trap." Working Paper, Linz and Graz University of Technology, K.U. Leuven, ING Financial Markets, 2006.

[2] Bates, D. S. “Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options.” *The Review of Financial Studies.* Vol 9. No. 1. 1996.

[3] Carr, P. and D.B. Madan. “Option Valuation Using the Fast Fourier Transform.” *Journal of Computational Finance.* Vol 2. No. 4. 1999.

[4] Chourdakis, K. “Option Pricing Using Fractional FFT.” *Journal of Computational Finance.* 2005.

[5] Heston, S. L. “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.” *The Review of Financial Studies.* Vol 6. No. 2. 1993.

See Also

`optByHestonFFT` | `optSensByHestonFFT` | `optByHestonNI` | `optSensByHestonNI` | `optSensByBatesFFT` | `optByBatesNI` | `optSensByBatesNI` | `optByMertonFFT` | `optSensByMertonFFT` | `optByMertonNI` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optSensByBatesFFT

Option price and sensitivities by Bates model using FFT and FRFT

Syntax

```
[PriceSens,StrikeOut] = optSensByBatesFFT(Rate,AssetPrice,Settle,Maturity,
OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq)
[PriceSens,StrikeOut] = optSensByBatesFFT( ___ ,Name,Value)
```

Description

`[PriceSens,StrikeOut] = optSensByBatesFFT(Rate,AssetPrice,Settle,Maturity, OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq)` computes vanilla European option price and sensitivities by Bates model, using Carr-Madan FFT and Chourdakis FRFT methods.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,StrikeOut] = optSensByBatesFFT( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Sensitivity Surface Using the Bates Model

Use `optSensByBatesFFT` to calibrate the FFT strike grid for sensitivities, compute option sensitivities, and plot option sensitivity surfaces.

Define Option Variables and Bates Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

V0 = 0.04;
ThetaV = 0.05;
Kappa = 1.0;
SigmaV = 0.2;
RhoSV = -0.7;
MeanJ = 0.02;
JumpVol = 0.08;
JumpFreq = 2;
```

Compute the Option Sensitivities for the Entire FFT (or FRFT) Strike Grid, Without Specifying "Strike"

Compute option sensitivities and also output the corresponding strikes. If the `Strike` input is empty (`[]`), option sensitivities will be computed on the entire FFT (or FRFT) strike grid. The FFT (or FRFT) strike grid is determined as `exp(log-strike grid)`, where each column of the log-strike grid has `NumFFT` points with `LogStrikeStep` spacing that are roughly centered around each element of `log(AssetPrice)`. The default value for `NumFFT` is `2^12`. In addition to the sensitivities in the first output, the optional last output contains the corresponding strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = []; % Strike is not specified (will use the entire FFT strike grid)

% Compute option sensitivities for the entire FFT strike grid
[Delta, Kout] = optSensByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta");

% Show the lowest and highest strike values on the FFT strike grid
format
MinStrike = Kout(1) % Lowest possible strike in the current FFT strike grid

MinStrike = 2.9205e-135

MaxStrike = Kout(end) % Highest possible strike in the current FFT strike grid

MaxStrike = 1.8798e+138


% Show a subset of the strikes and corresponding option sensitivities
Range = (2046:2052);
[Kout(Range) Delta(Range)]

ans = 7×2

   50.4929    0.9846
   58.8640    0.9585
   68.6231    0.8498
   80.0000    0.5630
   93.2631    0.1955
  108.7251    0.0319
  126.7505    0.0033
```

Change the Number of FFT (or FRFT) Points and Compare with optSensByBatesNI

Try a different number of FFT (or FRFT) points, and compare the results with numerical integration. Unlike `optSensByBatesFFT`, which uses FFT (or FRFT) techniques for fast computation across the whole range of strikes, the `optSensByBatesNI` function uses direct numerical integration and it is typically slower, especially for multiple strikes. However, the values computed by `optSensByBatesNI` can serve as a benchmark for adjusting the settings for `optSensByBatesFFT`.

```
% Try a smaller number of FFT points
% (e.g. for faster performance or smaller memory footprint)
NumFFT = 2^10; % Smaller than the default value of 2^12
Strike = []; % Strike is not specified (will use the entire FFT strike grid)
[Delta, Kout] = optSensByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
```

```
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta", 'NumFFT', NumFFT);

% Compare with numerical integration method
Range = (510:516);
Strike = Kout(Range);
DeltaFFT = Delta(Range);
DeltaNI = optSensByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ...
    ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta");
Error = abs(DeltaFFT-DeltaNI);
table(Strike, DeltaFFT, DeltaNI, Error)

ans=7×4 table
    Strike     DeltaFFT      DeltaNI        Error
    ______    __________    __________    __________

    12.696       0.9265       0.99002      0.063524
    23.449      0.95153       0.99002      0.038497
    43.312      0.95928       0.98928      0.029994
        80       0.5355       0.56303      0.027531
    147.76    0.0016267    0.00025691     0.0013698
    272.93    0.00058267    1.8942e-09    0.00058267
    504.11    0.00017752    8.7099e-10    0.00017752
```

Make Further Adjustments to FFT (or FRFT)

If the values in the output `DeltaFFT` are significantly different from those in `DeltaNI`, try making adjustments to `optSensByBatesFFT` settings, such as `CharacteristicFcnStep`, `LogStrikeStep`, `NumFFT`, `DampingFactor`, and so on. Note that if (`LogStrikeStep` * `CharacteristicFcnStep`) is 2*pi/ `NumFFT`, FFT is used. Otherwise, FRFT is used.

```
Strike = []; % Strike is not specified (will use the entire FFT or FRFT strike grid)
[Delta, Kout] = optSensByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta", 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001);

% Compare with numerical integration method
Strike = Kout(Range);
DeltaFFT = Delta(Range);
DeltaNI = optSensByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ...
    ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta");
Error = abs(DeltaFFT-DeltaNI);
table(Strike, DeltaFFT, DeltaNI, Error)

ans=7×4 table
    Strike    DeltaFFT    DeltaNI      Error
    ______    ________    _______    __________

    79.76     0.57037     0.57037    6.3042e-09
    79.84     0.56793     0.56793     7.156e-09
    79.92     0.56548     0.56548     7.975e-09
       80     0.56303     0.56303    8.7573e-09
    80.08     0.56057     0.56057    9.4992e-09
    80.16     0.55811     0.55811    1.0197e-08
```

```
    80.24     0.55564     0.55564    1.0847e-08
```

```
% Save the final FFT (or FRFT) strike grid for future reference. For
% example, it provides information about the range of Strike inputs for
% which the FFT (or FRFT) operation is valid.
FFTStrikeGrid = Kout;
MinStrike = FFTStrikeGrid(1) % Strike cannot be less than MinStrike
```

```
MinStrike = 47.9437
```

```
MaxStrike = FFTStrikeGrid(end) % Strike cannot be greater than MaxStrike
```

```
MaxStrike = 133.3566
```

Compute the Option Sensitivity for a Single Strike

Once the desired FFT (or FRFT) settings are determined, use the `Strike` input to specify the strikes rather than providing an empty array. If the specified strikes do not match a value on the FFT (or FRFT) strike grid, the outputs are interpolated on the specified strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;

Delta = optSensByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta", 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001)
```

```
Delta = 0.5630
```

Compute the Option Sensitivities for a Vector of Strikes

Use the `Strike` input to specify the strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Delta = optSensByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta", 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001)
```

```
Delta = 5×1

    0.6807
    0.6234
    0.5630
    0.5011
    0.4392
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Delta = optSensByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta", 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001) % Five values in vector output
```

```
Delta = 5×1

    0.6625
    0.6232
    0.5958
    0.5748
    0.5577
```

Expand the Outputs for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the outputs into `NStrikes`-by-`NMaturities` matrices. In this case, they are square matrices.

```
[Delta, Kout] = optSensByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta", 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'ExpandOutput', true) % (5 x 5) matrix output
```

```
Delta = 5×5

    0.6625    0.6556    0.6515    0.6483    0.6455
    0.6222    0.6232    0.6239    0.6241    0.6238
    0.5805    0.5900    0.5958    0.5996    0.6019
    0.5381    0.5564    0.5674    0.5748    0.5798
    0.4954    0.5225    0.5389    0.5499    0.5577


Kout = 5×5

    76    76    76    76    76
    78    78    78    78    78
    80    80    80    80    80
    82    82    82    82    82
    84    84    84    84    84
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities`. That is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Delta = optSensByBatesFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta", 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'ExpandOutput', true) % (5 x 6) matrix output
```

```
Delta = 5×6

    0.6807    0.6625    0.6556    0.6515    0.6483    0.6455
    0.6234    0.6222    0.6232    0.6239    0.6241    0.6238
    0.5630    0.5805    0.5900    0.5958    0.5996    0.6019
    0.5011    0.5381    0.5564    0.5674    0.5748    0.5798
    0.4392    0.4954    0.5225    0.5389    0.5499    0.5577
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Asset Prices

When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Delta = optSensByBatesFFT(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, ...
    Strike, V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta", 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'ExpandOutput', true) % (5 x 4) matrix output
```

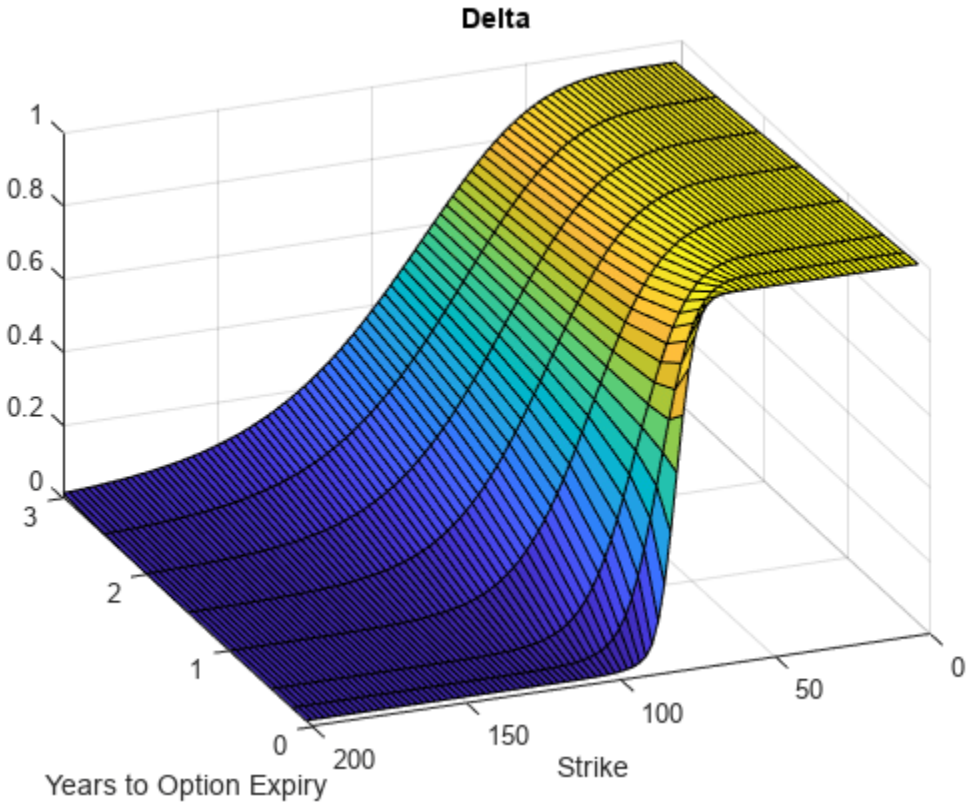
```
Delta = 5×4

    0.4350    0.5579    0.6625    0.7457
    0.3881    0.5124    0.6222    0.7120
    0.3432    0.4670    0.5805    0.6763
    0.3010    0.4223    0.5381    0.6390
    0.2619    0.3789    0.4954    0.6002
```

Plot Option Sensitivity Surfaces

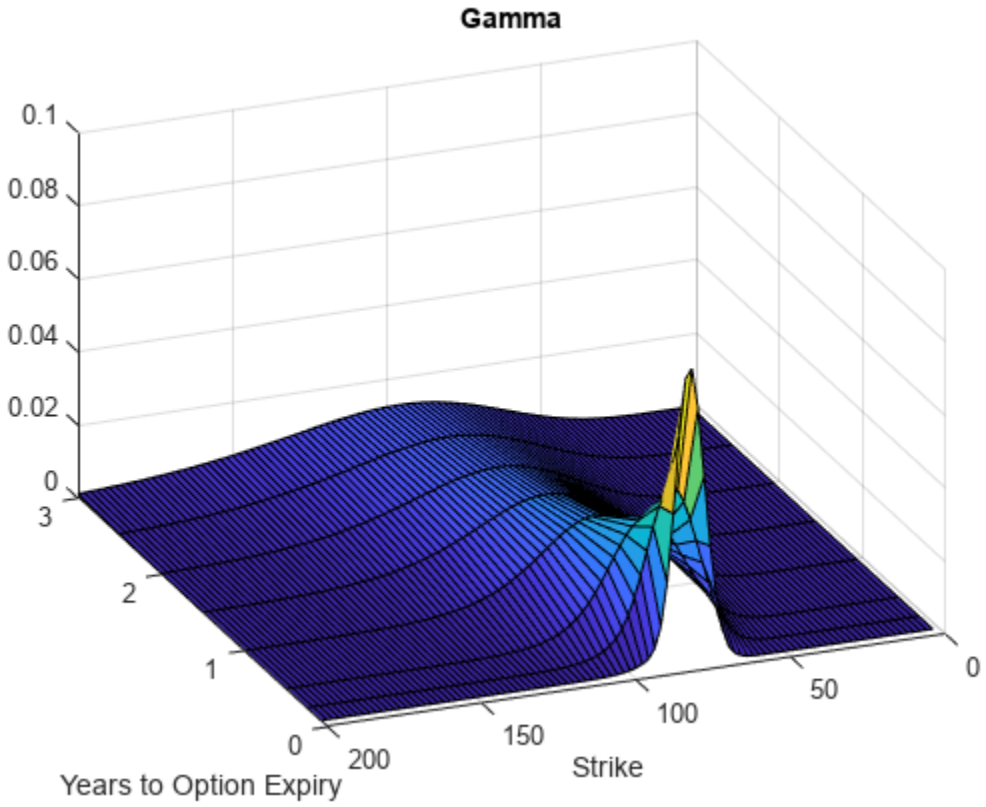
Use the `Strike` input to specify the strikes. Increase the value for `NumFFT` to support a wider range of strikes. Also, the `Maturity` input can be a vector. Set `ExpandOutput` to `"true"` to output the surfaces as `NStrikes`-by-`NMaturities` matrices.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

% Increase 'NumFFT' to support a wider range of strikes
NumFFT = 2^13;
```

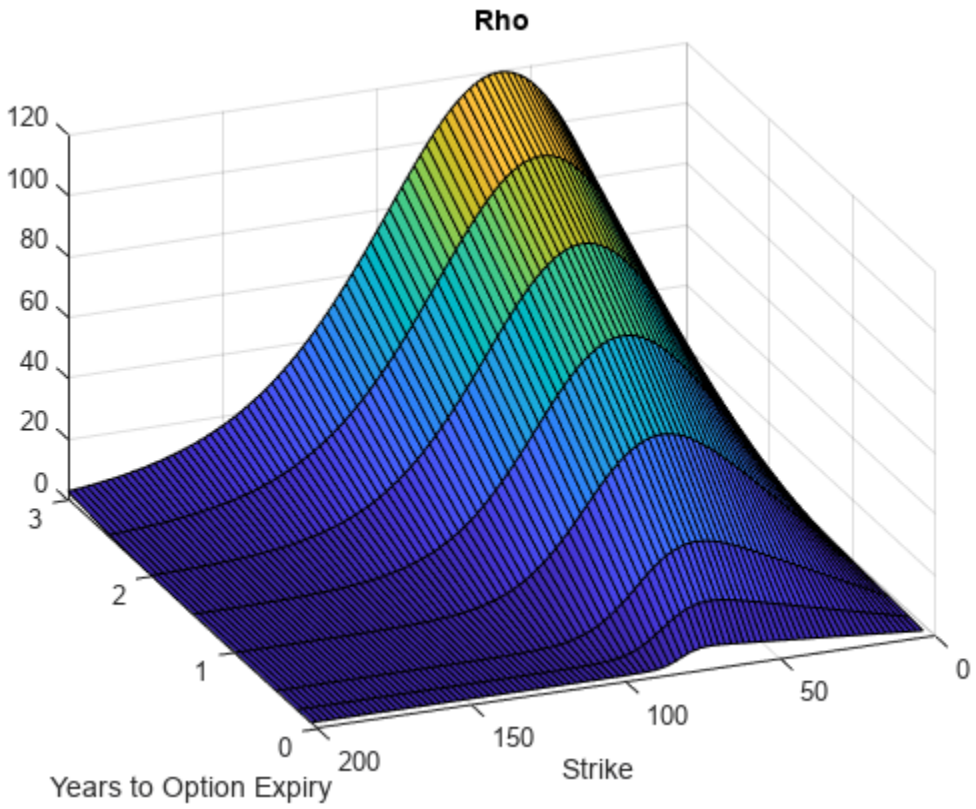
```
[Delta, Gamma, Rho, Theta, Vega, VegaLT] = optSensByBatesFFT(...
    Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'OutSpec', ["delta", "gamma", "rho", "theta", "vega", "vegalt"], ...
    'ExpandOutput', true);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Delta);
title('Delta');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
```



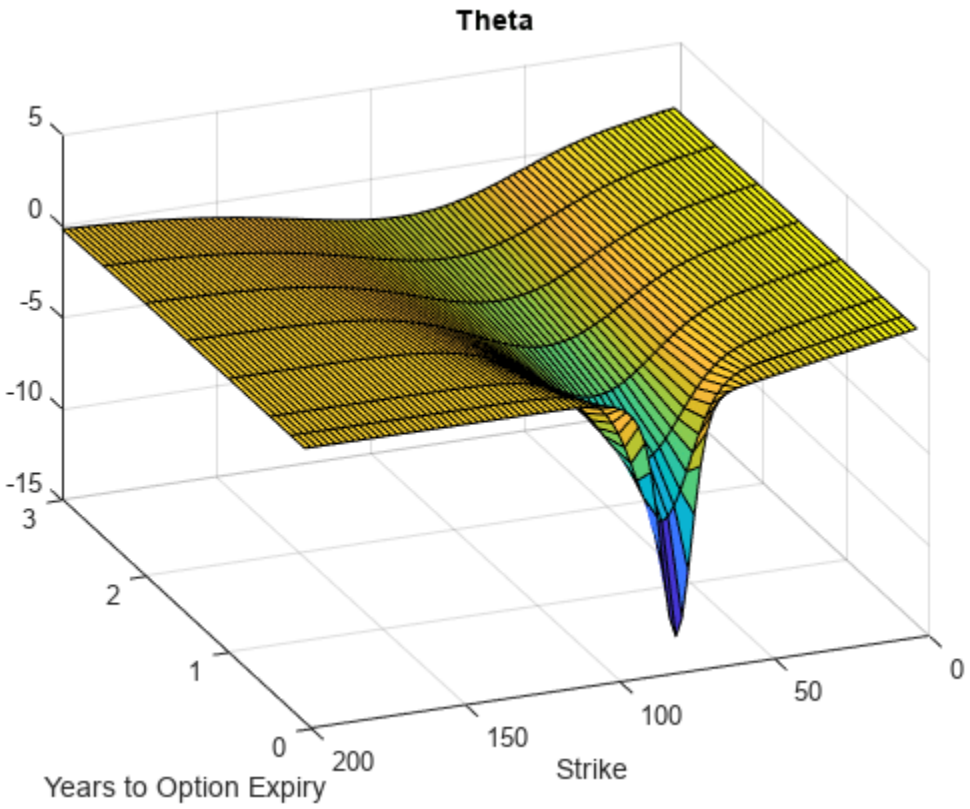

```
figure;
surf(X,Y,Gamma)
title('Gamma')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



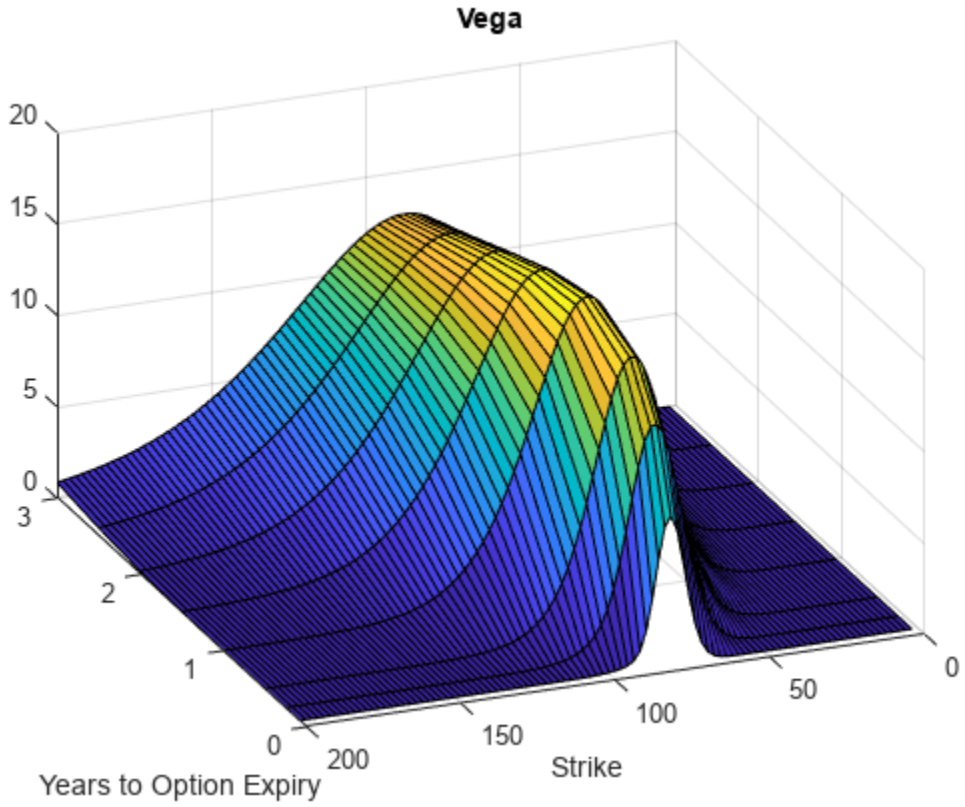

```
figure;
surf(X,Y,Rho)
title('Rho')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



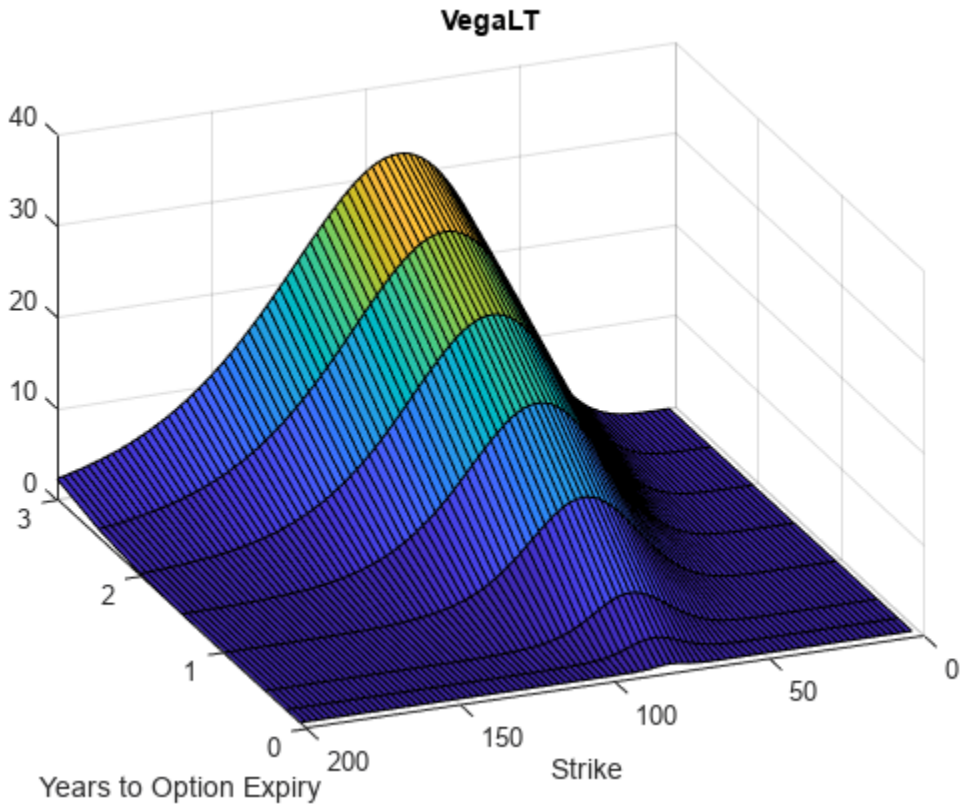

```
figure;
surf(X,Y,Theta)
title('Theta')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,Vega)
title('Vega')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,VegaLT)
title('VegaLT')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



Input Arguments

`Rate` — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

numeric

Current underlying asset price, specified as numeric value using a scalar or a `NINST`-by-`1` or `NColumns`-by-`1` vector.

For more information on the proper dimensions for `AssetPrice`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Settle` — Option settlement date

datetime array | string array | date character vector

Option settlement date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `optSensByBatesFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Settle`, see the name-value pair argument `ExpandOutput`.

`Maturity` — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optSensByBatesFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Maturity`, see the name-value pair argument `ExpandOutput`.

`OptSpec` — Definition of option
cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

For more information on the proper dimensions for `OptSpec`, see the name-value pair argument `ExpandOutput`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a `NINST`-by-`1`, `NRows`-by-`1`, `NRows`-by-`NColumns` vector of strike prices.

If this input is an empty array (`[]`), option prices are computed on the entire FFT (or FRFT) strike grid, which is determined as `exp(log-strike grid)`. Each column of the log-strike grid has `'NumFFT'` points with `'LogStrikeStep'` spacing that are roughly centered around each element of `log(AssetPrice)`.

For more information on the proper dimensions for `Strike`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`V0` — Initial variance of underlying asset
numeric

Initial variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
numeric

Long-term variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

Kappa — Mean revision speed for the variance of underlying asset
numeric

Mean revision speed for the underling asset, specified as a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
numeric

Volatility of the variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric value.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal value where `log`(1+*J*) is normally distributed with mean (`log`(1+`MeanJ`)-0.5*`JumpVol^`2) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log`(1+*J*)
decimal

Standard deviation of `log`(1+*J*) where `J` is the random percentage jump size, specified as a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
numeric

Annual frequency of Poisson jump process, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PriceSens,StrikeOut] = optSensByBatesFFT(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq,'Basis',7)`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric value.

Data Types: `double`

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, specified as the comma-separated pair consisting of `'VolRiskPremium'` and a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating Little Heston Trap formulation
`true` (default) | logical with values `true` or `false`

Flag indicating Little Heston Trap formulation by Albrecher *et al*, specified as the comma-separated pair consisting of `'LittleTrap'` and a logical:

- `true` — Use the Albrecher *et al* formulation.
- `false` — Use the original Heston formation.

Data Types: `logical`

`OutSpec` — Define outputs
`["price"]` (default) | string array with values `"price"`, `"delta"`, `"gamma"`, `"vega"`, `"rho"`, `"theta"`, and `"vegalt"` | cell array of character vectors with values `'price'`, `'delta'`, `'gamma'`, `'vega'`, `'rho'`, `'theta'`, and `'vegalt'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` string array or cell array of character vectors with supported values.

Note `"vega"` is the sensitivity with respect the initial volatility sqrt(`V0`). In contrast, `"vegalt"` is the sensitivity with respect to the long-term volatility sqrt(`ThetaV`).

Example: `OutSpec = ["price","delta","gamma","vega","rho","theta","vegalt"]`

Data Types: `string` | `cell`

`NumFFT` — Number of grid points in the characteristic function variable
`4096` (default) | numeric

Number of grid points in the characteristic function variable and in each column of the log-strike grid, specified as the comma-separated pair consisting of `'NumFFT'` and a scalar numeric value.

Data Types: `double`

`CharacteristicFcnStep` — Characteristic function variable grid spacing
`0.01` (default) | numeric

Characteristic function variable grid spacing, specified as the comma-separated pair consisting of `'CharacteristicFcnStep'` and a scalar numeric value.

Data Types: `double`

`LogStrikeStep` — Log-strike grid spacing
`2*pi/NumFFT/CharacteristicFcnStep` (default) | numeric

Log-strike grid spacing, specified as the comma-separated pair consisting of `'LogStrikeStep'` and a scalar numeric value.

Note If (`LogStrikeStep*CharacteristicFcnStep`) is `2*pi/NumFFT`, FFT is used. Otherwise, FRFT is used.

Data Types: `double`

`DampingFactor` — Damping factor for Carr-Madan formulation
`1.5` (default) | numeric

Damping factor for Carr-Madan formulation, specified as the comma-separated pair consisting of `'DampingFactor'` and a scalar numeric value.

Data Types: `double`

`Quadrature` — Type of quadrature
`"simpson"` (default) | character vector with values:`'simpson'` or `'trapezoidal'` | string array with values: `"simpson"` or `"trapezoidal"`

Type of quadrature, specified as the comma-separated pair consisting of `'Quadrature'` and a single character vector or string array with a value of `'simpson'` or `'trapezoidal'`.

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-1 vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-1 vector, or a `NRows`-by-`NColumns` matrix. If `Strike` is empty, `NRows` is equal to `NumFFT`. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-1 vectors.
- `false` — If `false`, the outputs are `NINST`-by-1 vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-1 vectors.

Data Types: `logical`

Output Arguments

`PriceSens` — Option prices or sensitivities
numeric

Option prices or sensitivities, returned as a `NINST`-by-1, or `NRows`-by-`NColumns`, depending on `ExpandOutput`. The name-value pair argument `OutSpec` determines the types and order of the outputs.

`StrikeOut` — Strikes corresponding to `Price`
numeric

Strikes corresponding to `Price`, returned as a `NINST`-by-1, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Bates Stochastic Volatility Jump Diffusion Model

The Bates model (Bates (1996)) is an extension of the Heston model, where, in addition to stochastic volatility, the jump diffusion parameters similar to Merton (1976) were also added to model sudden asset price movements.

The stochastic differential equation is:

$$dS_t = (r - q - \lambda_p \mu_J)S_t dt + \sqrt{v_t} S_t dW_t + JS_t dP_t$$
$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t$$
$$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$
$$\mathrm{prob}(dP_t = 1) = \lambda_p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

J is the random percentage jump size conditional on the jump occurring, where `ln`$(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1 + J)\delta\sqrt{2\pi}} \exp\left\{ \frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2} \right\}$$

v_0 is the initial variance of the asset price at $t = 0$ ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for ($\kappa > 0$).

σ_v is the volatility of variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and W_t^v for ($-1 \le p \le 1$).

μ_J is the mean of J for ($\mu_J > -1$).

δ is the standard deviation of `ln`$(1+J)$ for ($\delta \ge 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \ge 0$).

The characteristic function $f_{Bates_j(\phi)}$ for $j = 1$ (asset price mean measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Bates(\phi)} = \exp(C_j + D_j v_0 + i\phi \ln S_t)\exp\left(\lambda_p \tau(1+\mu_J)^{m_j} + \frac{1}{2}\left[(1+\mu_j)^{i\phi} e^{\delta^2\left(m_j i\phi + \frac{(i\phi)^2}{2}\right)} - 1\right] - \lambda_p \tau \mu_J i\phi\right)$$

$$m_j = \begin{Bmatrix} m_1 = \frac{1}{2} \\ m_2 = -\frac{1}{2} \end{Bmatrix}$$

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1 - g_j e^{d_j \tau}}{1 - g_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1 - e^{d_j\tau}}{1 - g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

where

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity for ($\tau = T - t$).

i is the unit imaginary number for ($i^2 = -1$).

The definitions for C_j and D_j under "The Little Heston Trap" by Albrecher et al. (2007) are:

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1 - \varepsilon_j e^{-d_j \tau}}{1 - \varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1 - e^{-d_j\tau}}{1 - \varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Carr-Madan Formulation

The Carr and Madan (1999) formulation is a popular modified implementation of Heston (1993) framework.

Rather than computing the probabilities P_1 and P_2 as intermediate steps, Carr and Madan developed an alternative expression so that taking its inverse Fourier transform gives the option price itself directly.

$$Call(k) = \frac{e^{-\alpha k}}{\pi}\int_0^\infty \mathrm{Re}\left[e^{-iuk}\psi(u)\right]du$$

$$\psi(u) = \frac{e^{-r\tau}f_2(\phi = (u-(\alpha+1)i))}{\alpha^2+\alpha-u^2+iu(2\alpha+1)}$$

$$Put(K) = Call(K) + Ke^{-r\tau} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number (i^2= -1).

ϕ is the characteristic function variable.

α is the damping factor.

u is the characteristic function variable for integration, where ϕ = (u - (α+1)i).

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

To apply FFT or FRFT to this formulation, the characteristic function variable for integration, u, is discretized into `NumFFT`(N) points with the step size `CharacteristicFcnStep` (Δu), and the log-strike k is discretized into N points with the step size `LogStrikeStep`(Δk).

The discretized characteristic function variable for integration, u_j(for j = 1,2,3,...,N), has a minimum value of 0 and a maximum value of (N-1) (Δu), and it approximates the continuous integration range from 0 to infinity.

The discretized log-strike grid, k_n(for n = 1, 2, 3, N) is approximately centered around `ln`(S_t), with a minimum value of

$$\ln(S_t) - \frac{N}{2}\Delta k$$

and a maximum value of

$$\ln(S_t) + \left(\frac{N}{2} - 1\right)\Delta k$$

Where the minimum allowable strike is

$$S_t \exp\left(-\frac{N}{2}\Delta k\right)$$

and the maximum allowable strike is

$$S_t \exp\left[\left(\frac{N}{2}-1\right)\Delta k\right]$$

As a result of the discretization, the expression for the call option becomes

$$Call(k_n) = \Delta u \frac{e^{-\alpha k_n}}{\pi} \sum_{j=1}^{N} \mathrm{Re}\left[e^{-i\Delta k \Delta u (j-1)(n-1)} e^{iuj\left[\frac{N\Delta k}{2} - \ln(S_t)\right]} \psi(u_j)\right] w_j$$

where

Δu is the step size of discretized characteristic function variable for integration.

Δk is the step size of discretized log-strike.

N is the number of FFT/FRFT points

w_j is the weights for quadrature used for approximating the integral.

FFT is used to evaluate the above expression if Δk and Δu are subject to the following constraint:

$$\Delta k \Delta u = \left(\frac{2\pi}{N}\right)$$

otherwise, the functions use the FRFT method described in Chourdakis (2005).

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optSensByBatesFFT` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Albrecher, H., Mayer, P., Schoutens, W., and Tistaert, J. "The Little Heston Trap." Working Paper, Linz and Graz University of Technology, K.U. Leuven, ING Financial Markets, 2006.

[2] Bates, D. S. “Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options.” *The Review of Financial Studies.* Vol 9. No. 1. 1996.

[3] Carr, P. and D.B. Madan. “Option Valuation Using the Fast Fourier Transform.” *Journal of Computational Finance.* Vol 2. No. 4. 1999.

[4] Chourdakis, K. “Option Pricing Using Fractional FFT.” *Journal of Computational Finance.* 2005.

[5] Heston, S. L. “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.” *The Review of Financial Studies.* Vol 6. No. 2. 1993.

See Also

`optByHestonFFT` | `optSensByHestonFFT` | `optByHestonNI` | `optSensByHestonNI` | `optByBatesFFT` | `optByBatesNI` | `optSensByBatesNI` | `optByMertonFFT` | `optSensByMertonFFT` | `optByMertonNI` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optByBatesNI

Option price by Bates model using numerical integration

Syntax

```
Price = optByBatesNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,V0,
ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq)
Price = optByBatesNI( ___ ,Name,Value)
```

Description

`Price = optByBatesNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,V0, ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq)` computes vanilla European option price by Bates model, using numerical integration methods.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = optByBatesNI( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Price Surface Using the Bates Model

`optByBatesNI` uses numerical integration to compute option prices and then plot an option price surface.

Define Option Variables and Bates Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

V0 = 0.04;
ThetaV = 0.05;
Kappa = 1.0;
SigmaV = 0.2;
RhoSV = -0.7;
MeanJ = 0.02;
JumpVol = 0.08;
JumpFreq = 2;
```

Compute the Option Price for a Single Strike

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;
```

```
Call = optByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ...
    ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield)

Call = 5.3484
```

Compute the Option Prices for a Vector of Strikes

The `Strike` input can be a vector.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Call = optByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ...
    ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield)

Call = 5×1

    7.5765
    6.4020
    5.3484
    4.4173
    3.6073
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Call = optByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ...
    ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield) % Five values in vector output

Call = 5×1

    9.7516
   10.3931
   10.8865
   11.2990
   11.6491
```

Expand the Output for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the output into a `NStrikes`-by-`NMaturities` matrix. In this case, it is a square matrix.

```
Call = optByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ...
    ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'ExpandOutput', true) % (5 x 5) matrix output

Call = 5×5

    9.7516   11.4387   12.8395   14.0588   15.1361
    8.6554   10.3931   11.8344   13.0890   14.1980
    7.6432    9.4149   10.8865   12.1693   13.3046
    6.7153    8.5035    9.9951   11.2990   12.4553
    5.8705    7.6581    9.1594   10.4771   11.6491
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities` (that is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular).

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Call = optByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ...
    ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'ExpandOutput', true) % (5 x 6) matrix output

Call = 5×6

    7.5765    9.7516   11.4387   12.8395   14.0588   15.1361
    6.4020    8.6554   10.3931   11.8344   13.0890   14.1980
    5.3484    7.6432    9.4149   10.8865   12.1693   13.3046
    4.4173    6.7153    8.5035    9.9951   11.2990   12.4553
    3.6073    5.8705    7.6581    9.1594   10.4771   11.6491
```

Compute Option Prices for a Vector of Strikes and a Vector of Asset Prices

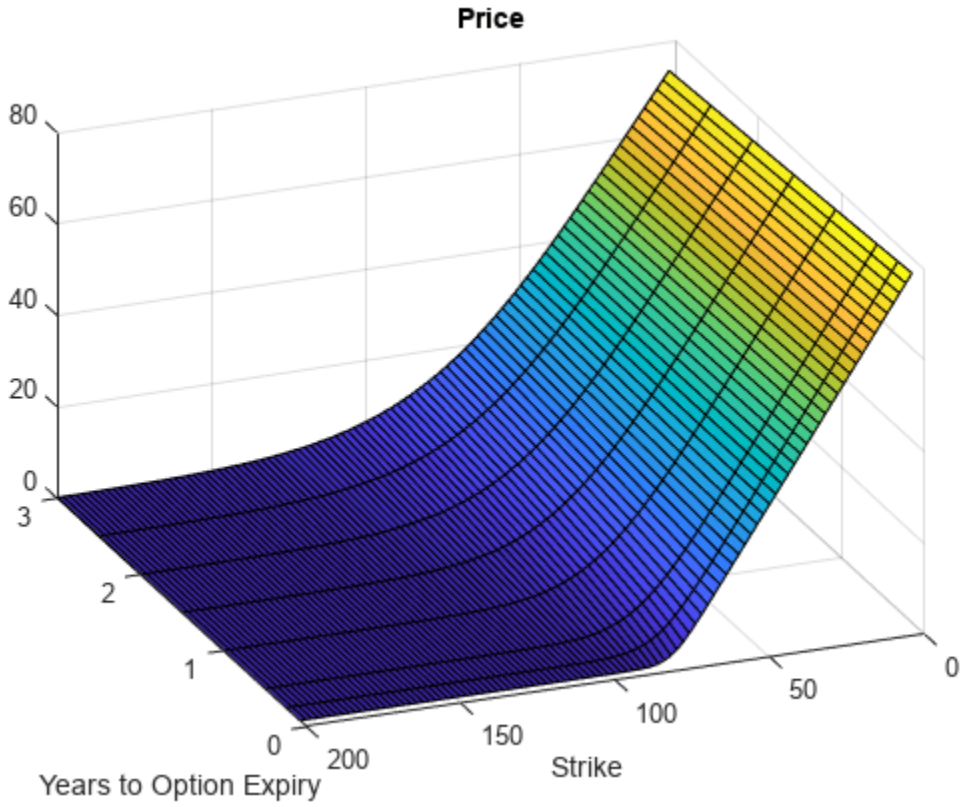
When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Call = optByBatesNI(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, V0, ...
    ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'ExpandOutput', true) % (5 x 4) matrix output

Call = 5×4

    4.2033    6.6918    9.7516   13.2808
    3.5558    5.8111    8.6554   11.9993
    2.9906    5.0181    7.6432   10.7934
    2.5018    4.3096    6.7153    9.6651
    2.0825    3.6818    5.8705    8.6158
```

Plot an Option Price Surface

The `Strike` and `Maturity` inputs can be vectors. Set `ExpandOutput` to `"true"` to output the surface as a `NStrikes`-by-`NMaturities` matrix.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

Call = optByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ...
    ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'ExpandOutput', true);


[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Call);
title('Price');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
zlim([0 80]);
```



Input Arguments

Rate — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

AssetPrice — Current underlying asset price

numeric

Current underlying asset price, specified as numeric value using a scalar or a `NINST`-by-`1` or `NColumns`-by-`1` vector.

For more information on the proper dimensions for `AssetPrice`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

Settle — Option settlement date

datetime array | string array | date character vector

Option settlement date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `optByBatesNI` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Settle`, see the name-value pair argument `ExpandOutput`.

`Maturity` — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optByBatesNI` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Maturity`, see the name-value pair argument `ExpandOutput`.

`OptSpec` — Definition of option
cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

For more information on the proper dimensions for `OptSpec`, see the name-value pair argument `ExpandOutput`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a `NINST`-by-`1`, `NRows`-by-`1`, `NRows`-by-`NColumns` vector of strike prices.

For more information on the proper dimensions for `Strike`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`V0` — Initial variance of underlying asset
numeric

Initial variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
numeric

Long-term variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`Kappa` — Mean revision speed for the variance of underlying asset
numeric

Mean revision speed for the underling asset, specified as a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
numeric

Volatility of the variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric value.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal value where `log`(1+*J*) is normally distributed with mean (`log`(1+`MeanJ`)-0.5*`JumpVol^2`) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log(1+`*`J`*`)`
decimal

Standard deviation of `log`(1+*J*) where `J` is the random percentage jump size, specified as a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
numeric

Annual frequency of Poisson jump process, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optByBatesNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq,'Basis',7)`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric value.

Data Types: `double`

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, specified as the comma-separated pair consisting of `'VolRiskPremium'` and a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating "Little Heston Trap" formulation
`true` (default) | logical with values `true` or `false`

Flag indicating "Little Heston Trap" formulation by Albrecher *et al*, specified as the comma-separated pair consisting of `'LittleTrap'` and a logical:

- `true` — Use the Albrecher *et al* formulation.
- `false` — Use the original Heston formation.

Data Types: `logical`

`AbsTol` — Absolute error tolerance for numerical integration
`1e-10` (default) | numeric

Absolute error tolerance for numerical integration, specified as the comma-separated pair consisting of `'AbsTol'` and a scalar numeric value.

Data Types: `double`

`RelTol` — Relative error tolerance for numerical integration
`1e-6` (default) | numeric

Relative error tolerance for numerical integration, specified as the comma-separated pair consisting of `'RelTol'` and a scalar numeric value.

Data Types: `double`

`IntegrationRange` — Numerical integration range used to approximate continuous integral over `[0 Inf]`
`[1e-9 Inf]` (default) | vector

Numerical integration range used to approximate the continuous integral over `[0 Inf]`, specified as the comma-separated pair consisting of `'IntegrationRange'` and a 1-by-2 vector representing `[LowerLimit UpperLimit]`.

Data Types: `double`

`Framework` — Framework for computing option prices and sensitivities using numerical integration of models
`"heston1993"` (default) | string with values `"heston1993"` or `"lewis2001"` | character vector with values `'heston1993'` or `'lewis2001'`

Framework for computing option prices and sensitivities using numerical integration of models, specified as the comma-separated pair consisting of `'Framework'` and a scalar string or character vector with the following values:

- `"heston1993"` or `'heston1993'` — Method used in Heston (1993)
- `"lewis2001"` or `'lewis2001'` — Method used in Lewis (2001)

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-1 vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-1 vector, or a `NRows`-by-`NColumns` matrix. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-1 vectors.
- `false` — If `false`, the outputs are `NINST`-by-1 vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-1 vectors.

Data Types: `logical`

Output Arguments

`Price` — Option prices

numeric

Option prices, returned as a `NINST`-by-`1`, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Bates Stochastic Volatility Jump Diffusion Model

The Bates model (Bates (1996)) is an extension of the Heston model, where, in addition to stochastic volatility, the jump diffusion parameters similar to Merton (1976) were also added to model sudden asset price movements.

The stochastic differential equation is:

$$dS_t = (r - q - \lambda_p \mu_J) S_t dt + \sqrt{v_t} S_t dW_t + J S_t dP_t$$
$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t$$
$$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$
$$\mathrm{prob}(dP_t = 1) = \lambda_p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

J is the random percentage jump size conditional on the jump occurring, where `ln`$(1+J)$ is normally distributed with mean $\ln(1+\mu_J)-\frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1+J)\delta\sqrt{2\pi}}\exp\left\{\frac{-\left[\ln(1+J)-\left(\ln(1+\mu_J)-\frac{\delta^2}{2}\right]\right]^2}{2\delta^2}\right\}$$

v_0 is the initial variance of the asset price at $t = 0$ ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for ($\kappa > 0$).

σ_v is the volatility of variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and W_t^v for ($-1 \le p \le 1$).

μ_J is the mean of J for ($\mu_J > -1$).

δ is the standard deviation of `ln`$(1+J)$ for ($\delta \ge 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \ge 0$).

The characteristic function $f_{Bates_j(\phi)}$ for $j = 1$ (asset price mean measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Bates(\phi)} = \exp(C_j + D_j v_0 + i\phi \ln S_t)\exp\left(\lambda_p \tau(1+\mu_J)^{m_j+\frac{1}{2}}\left[(1+\mu_j)^{i\phi}e^{\delta^2(m_j i\phi+\frac{(i\phi)^2}{2})}-1\right]-\lambda_p\tau\mu_J i\phi\right)$$

$$m_j = \begin{cases} m_1 = \frac{1}{2} \\ m_2 = -\frac{1}{2} \end{cases}$$

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1-g_j e^{d_j\tau}}{1-g_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1-e^{d_j\tau}}{1-g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

where

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity for ($\tau = T - t$).

i is the unit imaginary number for ($i^2 = -1$).

The definitions for C_j and D_j under "The Little Heston Trap" by Albrecher et al. (2007) are:

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1-\varepsilon_j e^{-d_j\tau}}{1-\varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1-e^{-d_j\tau}}{1-\varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Numerical Integration Method Under Heston (1993) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Heston (1993) framework is based on the following expressions:

$$Call(K) = S_t e^{-q\tau}P_1 - Ke^{-r\tau}P_2$$

$$Put(K) = Call(K) + Ke^{-r\tau} - S_t e^{-q\tau}$$

$$P_j = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \mathrm{Re}\left[\frac{e^{-i\phi\ln(K)}f_j(\phi)}{i\phi}\right]d\phi$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity ($\tau = T$-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K

i is a unit imaginary number ($i^2 = -1$)

ϕ is the characteristic function variable.

$f_j(\phi)$ is the characteristic function for $P_j (j = 1,2)$.

P_1 is the probability of $S_t > K$ under the asset price measure for the model.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

Where $j = 1,2$ so that $f_1(\phi)$ and $f_2(\phi)$ are the characteristic functions for probabilities P_1 and P_2, respectively.

This framework is chosen with the default value "`Heston1993`" for the `Framework` name-value pair argument.

Numerical Integration Method Under Lewis (2001) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Lewis (2001) framework is based on the following expressions:

$$Call(k) = S_t e^{-q\tau} - \frac{\sqrt{K}e^{-\tau t}}{\pi}\int_0^{\infty} \mathrm{Re}\left[K^{-iu} f_2\left(\phi = \left(u - \frac{i}{2}\right)\right)\frac{1}{u^2 + \frac{1}{4}}\right]du$$

$$Put(K) = Call(K) = Ke^{-\tau t} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity ($\tau = T\text{-}t$).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K

i is a unit imaginary number ($i^2 = -1$)

ϕ is the characteristic function variable.

u is the characteristic function variable for integration, where $\phi = \left(u - \frac{i}{2}\right)$.

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

This framework is chosen with the value "`Lewis2001`" for the `Framework` name-value pair argument.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optByBatesNI` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bates, D. S. “Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options.” *The Review of Financial Studies.* Vol 9. No. 1. 1996.

[2] Heston, S. L. “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.” *The Review of Financial Studies.* Vol 6. No. 2. 1993.

[3] Lewis, A. L. “A Simple Option Formula for General Jump-Diffusion and Other Exponential Levy Processes.” Envision Financial Systems and OptionCity.net, 2001.

See Also

`optByHestonFFT` | `optSensByHestonFFT` | `optByHestonNI` | `optSensByHestonNI` | `optByBatesFFT` | `optSensByBatesFFT` | `optSensByBatesNI` | `optByMertonFFT` | `optSensByMertonFFT` | `optByMertonNI` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optSensByBatesNI

Option price or sensitivities by Bates model using numerical integration

Syntax

```
PriceSens = optSensByBatesNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,
V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq)
PriceSens = optSensByBatesNI( ___ ,Name,Value)
```

Description

`PriceSens = optSensByBatesNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike, V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq)` computes vanilla European option price and sensitivities by Bates model, using numerical integration methods.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`PriceSens = optSensByBatesNI( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Sensitivity Surface Using the Bates Model

`optSensByBatesNI` uses numerical integration to compute option sensitivities and then to plot option sensitivity surfaces.

Define Option Variables and Bates Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

V0 = 0.04;
ThetaV = 0.05;
Kappa = 1.0;
SigmaV = 0.2;
RhoSV = -0.7;
MeanJ = 0.02;
JumpVol = 0.08;
JumpFreq = 2;
```

Compute the Option Sensitivity for a Single Strike

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;
```

```
Delta = optSensByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta")
```

```
Delta = 0.5630
```

Compute the Option Sensitivities for a Vector of Strikes

The `Strike` input can be a vector.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Delta = optSensByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta")
```

```
Delta = 5×1

    0.6807
    0.6234
    0.5630
    0.5011
    0.4392
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Delta  = optSensByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta") % Five values in vector output
```

```
Delta = 5×1

    0.6625
    0.6232
    0.5958
    0.5748
    0.5577
```

Expand the Output for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the output into a `NStrikes`-by-`NMaturities` matrix. In this case, it is a square matrix.

```
Delta  = optSensByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
```

```
    'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'ExpandOutput', true) % (5 x 5) matrix output
```

```
Delta = 5×5

    0.6625    0.6556    0.6515    0.6483    0.6455
    0.6222    0.6232    0.6239    0.6241    0.6238
    0.5805    0.5900    0.5958    0.5996    0.6019
    0.5381    0.5564    0.5674    0.5748    0.5798
    0.4954    0.5225    0.5389    0.5499    0.5577
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities`. That is, the output `NStrikes` -by- `NMaturities` matrix can be rectangular.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Delta  = optSensByBatesNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'ExpandOutput', true)  % (5 x 6) matrix output
```

```
Delta = 5×6

    0.6807    0.6625    0.6556    0.6515    0.6483    0.6455
    0.6234    0.6222    0.6232    0.6239    0.6241    0.6238
    0.5630    0.5805    0.5900    0.5958    0.5996    0.6019
    0.5011    0.5381    0.5564    0.5674    0.5748    0.5798
    0.4392    0.4954    0.5225    0.5389    0.5499    0.5577
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Asset Prices

When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

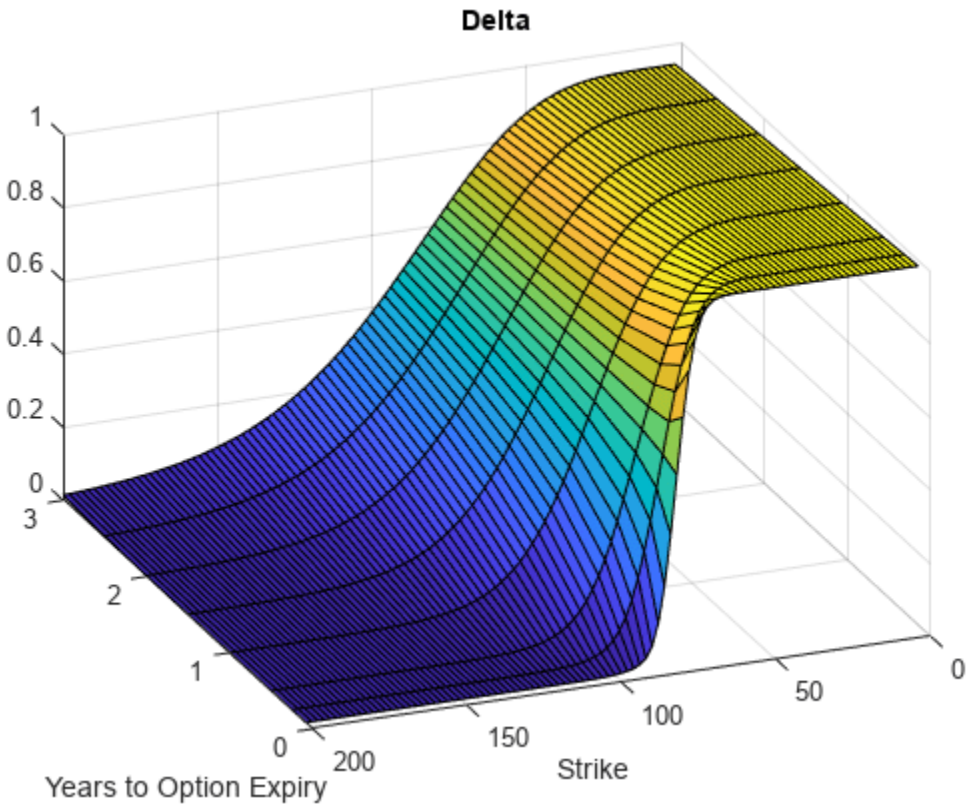
```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Delta  = optSensByBatesNI(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, ...
    'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'ExpandOutput', true) % (5 x 4) matrix output
```

```
Delta = 5×4

    0.4350    0.5579    0.6625    0.7457
    0.3881    0.5124    0.6222    0.7120
    0.3432    0.4670    0.5805    0.6763
    0.3010    0.4223    0.5381    0.6390
    0.2619    0.3789    0.4954    0.6002
```

Plot Option Sensitivity Surfaces

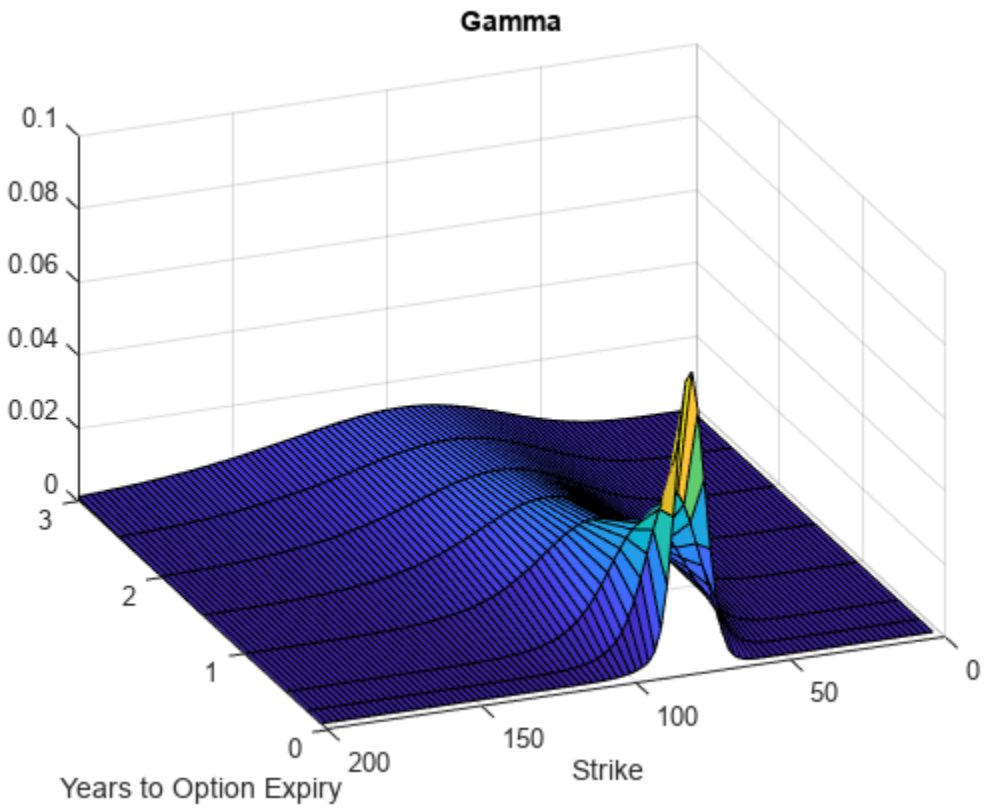
The `Strike` and `Maturity` inputs can be vectors. Set `ExpandOutput` to `"true"` to output the surfaces as `NStrikes`-by-`NMaturities` matrices.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

[Delta, Gamma, Rho, Theta, Vega, VegaLT] = optSensByBatesNI(...
    Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ThetaV, Kappa, ...
    SigmaV, RhoSV, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'OutSpec', ["delta", "gamma", "rho", "theta", "vega", "vegalt"], ...
    'ExpandOutput', true);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Delta);
title('Delta');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
```

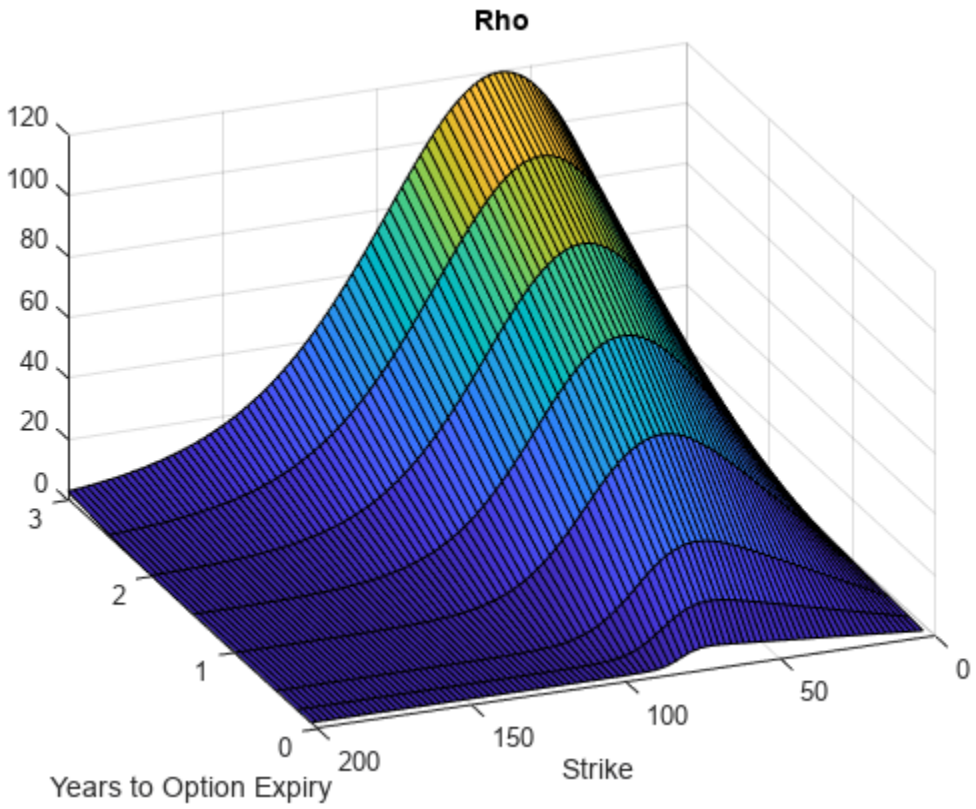



```
figure;
```

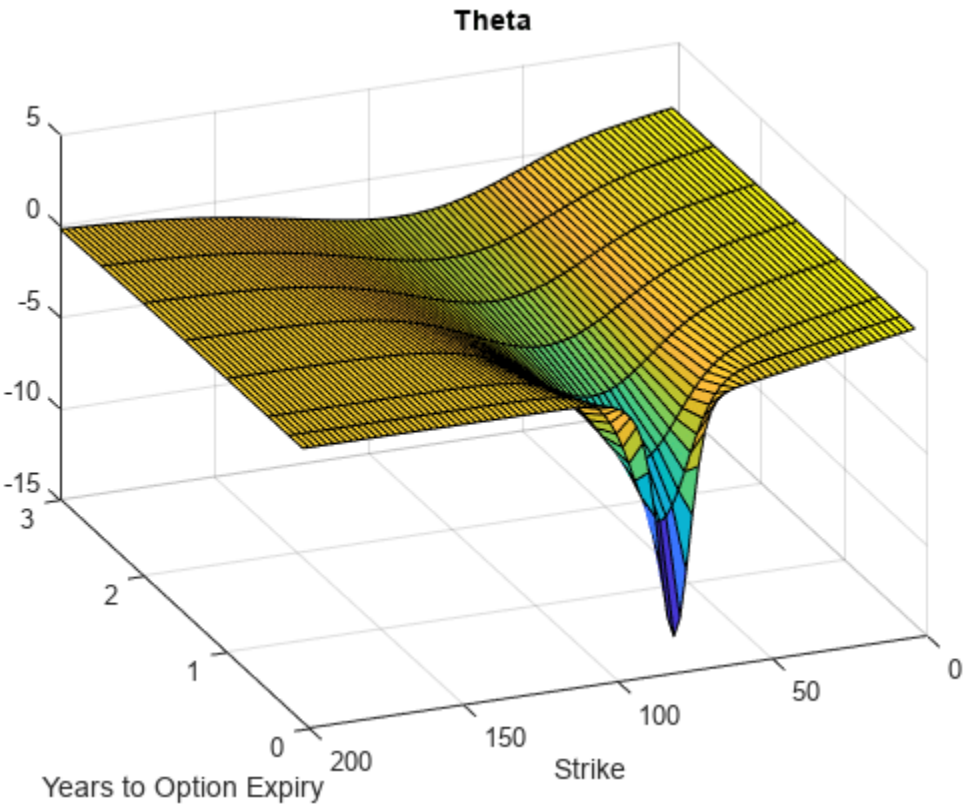
```
surf(X,Y,Gamma)
title('Gamma')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



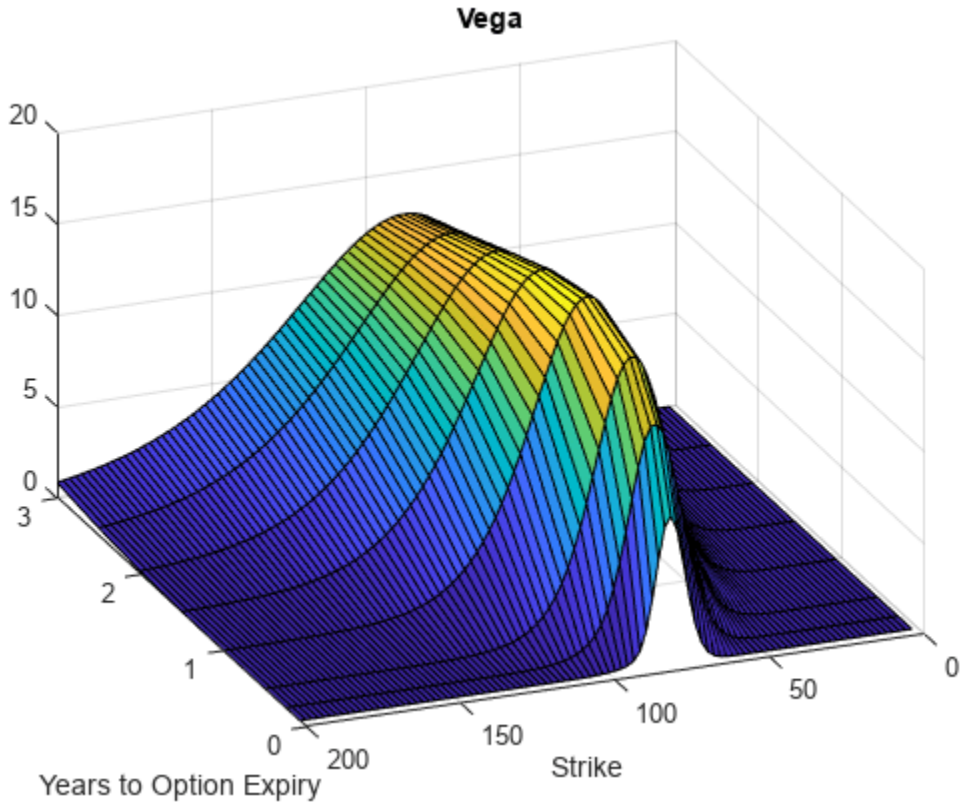

```
figure;
surf(X,Y,Rho)
title('Rho')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



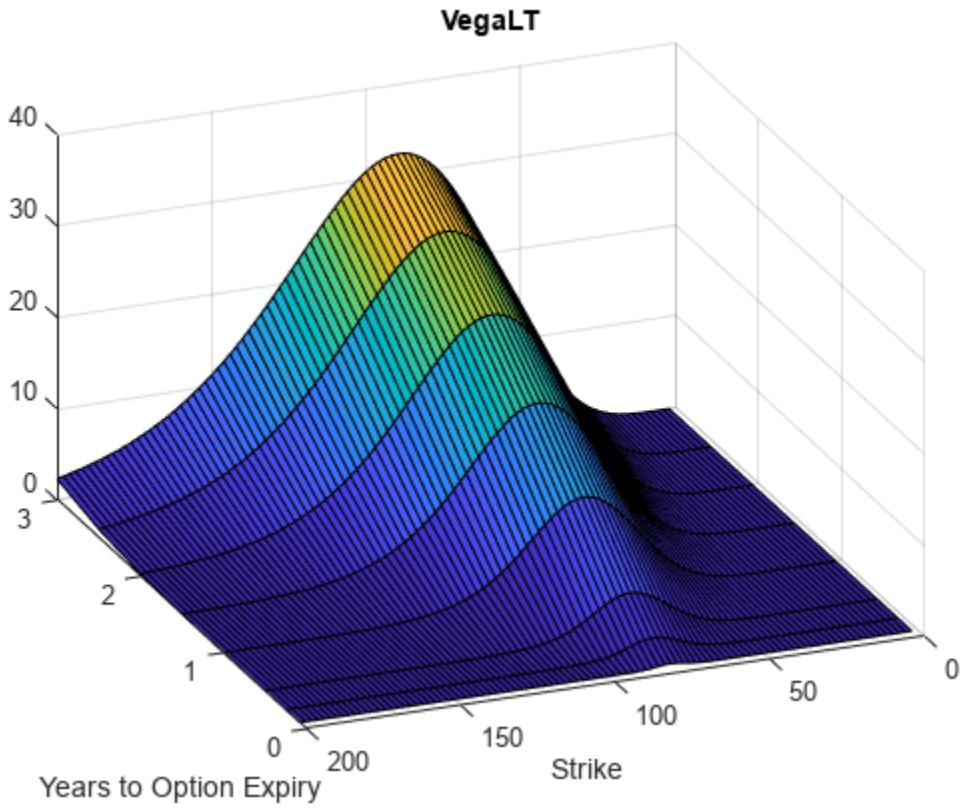

```
figure;
surf(X,Y,Theta)
title('Theta')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,Vega)
title('Vega')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,VegaLT)
title('VegaLT')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



Input Arguments

`Rate` — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

numeric

Current underlying asset price, specified as numeric value using a scalar or a `NINST`-by-`1` or `NColumns`-by-`1` vector.

For more information on the proper dimensions for `AssetPrice`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Settle` — Option settlement date

datetime array | string array | date character vector

Option settlement date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `optSensByBatesNI` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Settle`, see the name-value pair argument `ExpandOutput`.

`Maturity` — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optSensByBatesNI` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Maturity`, see the name-value pair argument `ExpandOutput`.

`OptSpec` — Definition of option
cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

For more information on the proper dimensions for `OptSpec`, see the name-value pair argument `ExpandOutput`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a `NINST`-by-`1`, `NRows`-by-`1`, `NRows`-by-`NColumns` vector of strike prices.

For more information on the proper dimensions for `Strike`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`V0` — Initial variance of underlying asset
numeric

Initial variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
numeric

Long-term variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`Kappa` — Mean revision speed for the variance of underlying asset
numeric

Mean revision speed for the underling asset, specified as a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
numeric

Volatility of the variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric value.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal value where `log`(1+*J*) is normally distributed with mean (`log`(1+`MeanJ`)-0.5*`JumpVol^2`) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log(1+`*`J`*`)`
decimal

Standard deviation of `log`(1+*J*) where `J` is the random percentage jump size, specified as a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
numeric

Annual frequency of Poisson jump process, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = optSensByBatesNI(Rate,AssetPrice,Settle,Maturity, OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,MeanJ,JumpVol,JumpFreq,'Basis',7)`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric value.

Data Types: `double`

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, specified as the comma-separated pair consisting of `'VolRiskPremium'` and a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating Little Heston Trap formulation
`true` (default) | logical with values `true` or `false`

Flag indicating Little Heston Trap formulation by Albrecher *et al*, specified as the comma-separated pair consisting of `'LittleTrap'` and a logical:

- `true` — Use the Albrecher *et al* formulation.
- `false` — Use the original Heston formation.

Data Types: `logical`

`OutSpec` — Define outputs
`["price"]` (default) | string array with values `"price"`, `"delta"`, `"gamma"`, `"vega"`, `"rho"`, `"theta"`, and `"vegalt"` | cell array of character vectors with values `'price'`, `'delta'`, `'gamma'`, `'vega'`, `'rho'`, `'theta'`, and `'vegalt'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` string array or cell array of character vectors with supported values.

Note `"vega"` is the sensitivity with respect the initial volatility sqrt(`V0`). In contrast, `"vegalt"` is the sensitivity with respect to the long-term volatility sqrt(`ThetaV`).

Example: `OutSpec = ["price","delta","gamma","vega","rho","theta","vegalt"]`

Data Types: `string` | `cell`

`AbsTol` — Absolute error tolerance for numerical integration
`1e-10` (default) | numeric

Absolute error tolerance for numerical integration, specified as the comma-separated pair consisting of `'AbsTol'` and a scalar numeric value.

Data Types: `double`

`RelTol` — Relative error tolerance for numerical integration
`1e-6` (default) | numeric

Relative error tolerance for numerical integration, specified as the comma-separated pair consisting of `'RelTol'` and a scalar numeric value.

Data Types: `double`

`IntegrationRange` — Numerical integration range used to approximate continuous integral over `[0 Inf]`
`[1e-9 Inf]` (default) | vector

Numerical integration range used to approximate the continuous integral over `[0 Inf]`, specified as the comma-separated pair consisting of `'IntegrationRange'` and a `1`-by-`2` vector representing `[LowerLimit UpperLimit]`.

Data Types: `double`

`Framework` — Framework for computing option prices and sensitivities using numerical integration of models
`"heston1993"` (default) | string with values `"heston1993"` or `"lewis2001"` | character vector with values `'heston1993'` or `'lewis2001'`

Framework for computing option prices and sensitivities using numerical integration of models, specified as the comma-separated pair consisting of `'Framework'` and a scalar string or character vector with the following values:

- `"heston1993"` or `'heston1993'` — Method used in Heston (1993)
- `"lewis2001"` or `'lewis2001'` — Method used in Lewis (2001)

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-`1` vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-`1` vector, or a `NRows`-by-`NColumns` matrix. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-`1` vectors.
- `false` — If `false`, the outputs are `NINST`-by-`1` vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-`1` vectors.

Data Types: `logical`

Output Arguments

`PriceSens` — Option prices

numeric

Option prices or sensitivities, returned as a `NINST`-by-`1`, or `NRows`-by-`NColumns`, depending on `ExpandOutput`. The name-value pair argument `OutSpec` determines the types and order of the outputs.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Bates Stochastic Volatility Jump Diffusion Model

The Bates model (Bates (1996)) is an extension of the Heston model, where, in addition to stochastic volatility, the jump diffusion parameters similar to Merton (1976) were also added to model sudden asset price movements.

The stochastic differential equation is:

$$dS_t = (r - q - \lambda_p \mu_J) S_t dt + \sqrt{v_t} S_t dW_t + J S_t dP_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$

$$\mathrm{prob}(dP_t = 1) = \lambda_p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

J is the random percentage jump size conditional on the jump occurring, where `ln`$(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1 + J)\delta\sqrt{2\pi}}\exp\left\{\frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2}\right\}$$

v_0 is the initial variance of the asset price at $t = 0$ ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for ($\kappa > 0$).

σ_v is the volatility of variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and W_t^v for ($-1 \le p \le 1$).

μ_J is the mean of J for ($\mu_J > -1$).

δ is the standard deviation of `ln`$(1+J)$ for ($\delta \ge 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \ge 0$).

The characteristic function $f_{Bates_j(\phi)}$ for $j = 1$ (asset price mean measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Bates(\phi)} = \exp(C_j + D_j v_0 + i\phi \ln S_t)\exp\left(\lambda_p \tau (1+\mu_J)^{m_j + \frac{1}{2}}\left[(1+\mu_j)^{i\phi} e^{\delta^2 (m_j i\phi + \frac{(i\phi)^2}{2})} - 1\right] - \lambda_p \tau \mu_J i\phi\right)$$

$$m_j = \begin{Bmatrix} m_1 = \frac{1}{2} \\ m_2 = -\frac{1}{2} \end{Bmatrix}$$

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1-g_j e^{d_j\tau}}{1-g_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1-e^{d_j\tau}}{1-g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

where

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity for ($\tau = T - t$).

i is the unit imaginary number for ($i^2 = -1$).

The definitions for C_j and D_j under “The Little Heston Trap” by Albrecher et al. (2007) are:

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1-\varepsilon_j e^{-d_j\tau}}{1-\varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1-e^{-d_j\tau}}{1-\varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Numerical Integration method Under Heston (1993) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Heston (1993) framework is based on the following expressions:

$$Call(K) = S_t e^{-q\tau} P_1 - K e^{-r\tau} P_2$$

$$Put(K) = Call(K) + K e^{-r\tau} - S_t e^{-q\tau}$$

$$P_j = \frac{1}{2} + \frac{1}{\pi} \int_0^{\infty} \mathrm{Re}\left[\frac{e^{-i\phi \ln(K)} f_j(\phi)}{i\phi} \right] d\phi$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number (i^2= -1).

ϕ is the characteristic function variable.

$f_j(\phi)$ is the characteristic function for $P_j (j = 1,2)$.

P_1 is the probability of $S_t > K$ under the asset price measure for the model.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

Where $j = 1,2$ so that $f_1(\phi)$ and $f_2(\phi)$ are the characteristic functions for probabilities P_1 and P_2, respectively.

This framework is chosen with the default value `"Heston1993"` for the `Framework` name-value pair argument.

Numerical Integration Method Under Lewis (2001) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Lewis (2001) framework is based on the following expressions:

$$Call(k) = S_t e^{-q\tau} - \frac{\sqrt{K} e^{-\tau t}}{\pi} \int_0^{\infty} \mathrm{Re}\left[K^{-iu} f_2\left(\phi = \left(u - \frac{i}{2}\right)\right) \frac{1}{u^2 + \frac{1}{4}} \right] du$$

$$Put(K) = Call(K) = K e^{-\tau t} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number (i^2= -1).

ϕ is the characteristic function variable.

u is the characteristic function variable for integration, where $\phi = \left(u - \frac{i}{2}\right)$.

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

This framework is chosen with the value "`Lewis2001`" for the `Framework` name-value pair argument.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optSensByBatesNI` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bates, D. S. "Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options." *The Review of Financial Studies.* Vol 9. No. 1. 1996.

[2] Heston, S. L. "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options." *The Review of Financial Studies.* Vol 6. No. 2. 1993.

[3] Lewis, A. L. “A Simple Option Formula for General Jump-Diffusion and Other Exponential Levy Processes.” Envision Financial Systems and OptionCity.net, 2001.

See Also

`optByHestonFFT` | `optSensByHestonFFT` | `optByHestonNI` | `optSensByHestonNI` | `optByBatesFFT` | `optSensByBatesFFT` | `optByBatesNI` | `optByMertonFFT` | `optSensByMertonFFT` | `optByMertonNI` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optByHestonFD

Option price by Heston model using finite differences

Syntax

```
[Price,PriceGrid,AssetPrices,Variances,Times] = optByHestonFD(Rate,
AssetPrice,Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV)
[Price,PriceGrid,AssetPrices,Variances,Times] = optByHestonFD( ___ ,Name,Value)
```

Description

`[Price,PriceGrid,AssetPrices,Variances,Times] = optByHestonFD(Rate, AssetPrice,Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV)` computes a vanilla European or American option price by the Heston model, using the alternating direction implicit (ADI) method.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceGrid,AssetPrices,Variances,Times] = optByHestonFD( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Price an American Option Using the Heston Model

Define the option variables and Heston model parameters.

```
AssetPrice = 10;
Strike = 10;
Rate = 0.1;
Settle = datetime(2017,1,1);
ExerciseDates = datetime(2017,4,2);

V0 = 0.0625;
ThetaV = 0.16;
Kappa = 5.0;
SigmaV = 0.9;
RhoSV = 0.1;
```

Compute the American put option price.

```
OptSpec = 'Put';
Price = optByHestonFD(Rate, AssetPrice, Settle, ...
ExerciseDates, OptSpec, Strike, V0, ThetaV, Kappa, SigmaV, RhoSV, 'AmericanOpt', 1)
```

```
Price = 0.5188
```

Input Arguments

`Rate` — Continuously compounded risk-free interest rate
scalar decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal.

Data Types: `double`

`AssetPrice` — Current underlying asset price
scalar numeric

Current underlying asset price, specified as numeric value using a scalar numeric.

Data Types: `double`

`Settle` — Option settlement date
datetime scalar | string scalar | date character vector

Option settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `optByHestonFD` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, there is only one `ExerciseDates` value and this is the option expiry date.
- For an American option, use a 1-by-2 vector of exercise date boundaries. 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, the option can be exercised between the `Settle` date and the single listed `ExerciseDate`.

To support existing code, `optByHestonFD` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
cell array of character vectors with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a scalar using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
scalar numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

`V0` — Initial variance of underlying asset
scalar numeric

Initial variance of the underlying asset, specified as a scalar numeric.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
scalar numeric

Long-term variance of the underlying asset, specified as a scalar numeric.

Data Types: `double`

`Kappa` — Mean revision speed for variance of underlying asset
scalar numeric

Mean revision speed for the variance of the underlying asset, specified as a scalar numeric.

Data Types: `double`

`SigmaV` — Volatility of variance of underlying asset
scalar numeric

Volatility of the variance of the underlying asset, specified as a scalar numeric.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
scalar numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceGrid,AssetPrices,Variances,Times] = optByHestonD(Rate,AssetPrice,Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,'Basis',7)`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,1, 2, 3, 4, 6, 7, 8, 9, `10`, 11, 12, 13

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)

- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | scalar numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric.

Note If you enter a value for `DividendYield`, then set `DividendAmounts` and `ExDividendDates = [ ]` or do not enter them. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield = 0`.

Data Types: `double`

`DividendAmounts` — Cash dividend amounts
`[ ]` (default) | vector

Cash dividend amounts, specified as the comma-separated pair consisting of `'DividendAmounts'` and a `NDIV`-by-`1` vector.

Note Each dividend amount must have a corresponding ex-dividend date. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield = 0`.

Data Types: `double`

`ExDividendDates` — Ex-dividend dates
`[ ]` (default) | datetime array | string array | date character vector

Ex-dividend dates, specified as the comma-separated pair consisting of `'ExDividendDates'` and an `NDIV`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optByHestonFD` also accepts serial date numbers as inputs, but they are not recommended.

`AssetPriceMax` — Maximum price for price grid boundary
if unspecified, value is calculated based on asset price distribution at maturity (default) | positive scalar

Maximum price for the price grid boundary, specified as the comma-separated pair consisting of `'AssetPriceMax'` and a positive scalar.

Data Types: `single` | `double`

`VarianceMax` — Maximum variance to use for variance grid boundary
`1.0` (default) | scalar numeric

Maximum variance to use for the variance grid boundary, specified as the comma-separated pair consisting of `'VarianceMax'` as a scalar numeric.

Data Types: `double`

`AssetGridSize` — Size of asset grid for finite difference grid
`400` (default) | scalar numeric

Size of the asset grid for finite difference grid, specified as the comma-separated pair consisting of `'AssetGridSize'` and a scalar numeric.

Data Types: `double`

`VarianceGridSize` — Number of nodes for variance grid for finite difference grid
`200` (default) | scalar numeric

Number of nodes for the variance grid for finite difference grid, specified as the comma-separated pair consisting of `'VarianceGridSize'` and a scalar numeric.

Data Types: `double`

`TimeGridSize` — Number of nodes of time grid for finite difference grid
`100` (default) | positive numeric scalar

Number of nodes of the time grid for finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive numeric scalar.

Data Types: `double`

`AmericanOpt` — Option type
`0` (European) (default) | scalar with value of `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar flag with one of these values:

- `0` — European
- `1` — American

Data Types: `double`

Output Arguments

`Price` — Option price
numeric

Option price, returned as a scalar numeric.

`PriceGrid` — Grid containing prices calculated by the finite difference method
numeric

Grid containing prices calculated by the finite difference method, returned as a three-dimensional grid with size `AssetGridSize` × `VarianceGridSize` × `TimeGridSize`. The depth is not necessarily equal to the `TimeGridSize`, because exercise and ex-dividend dates are added to the time grid. `PriceGrid(:, :, end)` contains the price for $t = 0$.

`AssetPrices` — Prices of the asset
vector

Prices of the asset corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Variances` — Variances
vector

Variances corresponding to the second dimension of `PriceGrid`, returned as a vector.

`Times` — Times
vector

Times corresponding to the third dimension of `PriceGrid`, returned as a vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Heston Stochastic Volatility Model

The Heston model is an extension of the Black-Scholes model, where the volatility (square root of variance) is no longer assumed to be constant, and the variance now follows a stochastic (CIR) process. This allows modeling the implied volatility smiles observed in the market.

The stochastic differential equation is:

$$dS_t = (r - q)S_t dt + \sqrt{v_t} S_t dW_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = \rho dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t

v_0 is the initial variance of the asset price at $t = 0$ for ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for the variance for ($\kappa > 0$).

σ_v is the volatility of the variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and W^v_t for ($-1 \le p \le 1$).

Version History

Introduced in R2018b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optByHestonFD` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Heston, S. L. "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options." *The Review of Financial Studies.* Vol 6, Number 2, 1993.

See Also

`optstockbyfd` | `optstocksensbyfd` | `optSensByHestonFD` | `optByLocalVolFD` | `optSensByLocalVolFD` | `optByBatesFD` | `optSensByBatesFD` | `optByMertonFD` | `optSensByMertonFD` | `Vanilla`

Topics

"Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers" on page 1-97

“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optSensByHestonFD

Option price and sensitivities by Heston model using finite differences

Syntax

```
[PriceSens,PriceGrid,AssetPrices,Variances,Times] = optByHestonFD(Rate,
AssetPrice,Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV)
[PriceSens,PriceGrid,AssetPrices,Variances,Times] = optByHestonFD( ___ ,
Name,Value)
```

Description

`[PriceSens,PriceGrid,AssetPrices,Variances,Times] = optByHestonFD(Rate, AssetPrice,Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV)` computes a vanilla European or American option price and sensitivities by the Heston model, using the alternating direction implicit (ADI) method.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,PriceGrid,AssetPrices,Variances,Times] = optByHestonFD( ___ , Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute an American Option Price and Sensitivities Using the Heston Model

Define the option variables and Heston model parameters.

```
AssetPrice = 10;
Strike = 10;
Rate = 0.1;
Settle = datetime(2017,1,1);
ExerciseDates = datetime(2017,4,2);

V0 = 0.0625;
ThetaV = 0.16;
Kappa = 5.0;
SigmaV = 0.9;
RhoSV = 0.1;
```

Compute the American put option price and sensitivities.

```
OptSpec = 'Put';
[Price,Delta,Gamma,Rho,Theta,Vega,VegaLT] = optSensByHestonFD(Rate, AssetPrice, Settle, ExerciseD
OptSpec, Strike, V0, ThetaV, Kappa, SigmaV, RhoSV, 'AmericanOpt', 1, ...
'OutSpec', ["Price" "Delta" "Gamma" "Rho" "Theta" "Vega" "VegaLT"])
```

```
Price = 0.5188

Delta = -0.4472

Gamma = 0.2822

Rho = -0.9234

Theta = -1.1614

Vega = 0.8998

VegaLT = 1.0921
```

Input Arguments

`Rate` — Continuously compounded risk-free interest rate

scalar decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal.

Data Types: `double`

`AssetPrice` — Current underlying asset price

scalar numeric

Current underlying asset price, specified as a scalar numeric.

Data Types: `double`

`Settle` — Option settlement date

datetime scalar | string scalar | date character vector

Option settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `optSensByHestonFD` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, there is only one `ExerciseDates` value and this is the option expiry date.
- For an American option, use a 1-by-2 vector of exercise date boundaries. 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, the option can be exercised between the `Settle` date and the single listed `ExerciseDate`.

To support existing code, `optSensByHestonFD` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option

cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a scalar using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
scalar numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

`V0` — Initial variance of underlying asset
scalar numeric

Initial variance of the underlying asset, specified as a scalar numeric.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
scalar numeric

Long-term variance of the underlying asset, specified as a scalar numeric.

Data Types: `double`

`Kappa` — Mean revision speed for the variance of underlying asset
scalar numeric

Mean revision speed for the variance of the underlying asset, specified as a scalar numeric.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
scalar numeric

Volatility of the variance of the underlying asset, specified as a scalar numeric.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
scalar numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PriceSens,PriceGrid,AssetPrices,Variances,Times] = optSensByHestonFD(Rate,AssetPrice,Settle,ExerciseDates,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,'Basis',7)`

Basis — Day-count basis of instrument
0 (default) | numeric values: 0,1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13

Day-count basis of the instrument, specified as the comma-separated pair consisting of 'Basis' and a scalar using a supported value:

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

For more information, see “Basis” on page 2-228.

Data Types: double

DividendYield — Continuously compounded underlying asset yield
0 (default) | scalar numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of 'DividendYield' and a scalar numeric.

Note If you enter a value for DividendYield, then set DividendAmounts and ExDividendDates = [ ] or do not enter them. If you enter values for DividendAmounts and ExDividendDates, then set DividendYield = 0.

Data Types: double

DividendAmounts — Cash dividend amounts
[ ] (default) | vector

Cash dividend amounts, specified as the comma-separated pair consisting of 'DividendAmounts' and a NDIV-by-1 vector.

Note Each dividend amount must have a corresponding ex-dividend date. If you enter values for DividendAmounts and ExDividendDates, then set DividendYield = 0.

Data Types: double

`ExDividendDates` — Ex-dividend dates
`[]` (default) | datetime array | string array | date character vector

Ex-dividend dates, specified as the comma-separated pair consisting of `'ExDividendDates'` and a `NDIV`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optSensByHestonFD` also accepts serial date numbers as inputs, but they are not recommended.

`AssetPriceMax` — Maximum price for price grid boundary
if unspecified, value is calculated based on asset price distribution at maturity (default) | positive scalar

Maximum price for price grid boundary, specified as the comma-separated pair consisting of `'AssetPriceMax'` and a positive scalar.

Data Types: `single` | `double`

`VarianceMax` — Maximum variance to use for variance grid boundary
`1.0` (default) | scalar numeric

Maximum variance to use for variance grid boundary, specified as the comma-separated pair consisting of `'VarianceMax'` as a scalar numeric.

Data Types: `double`

`AssetGridSize` — Size of asset grid for finite difference grid
`400` (default) | scalar numeric

Size of the asset grid for finite difference grid, specified as the comma-separated pair consisting of `'AssetGridSize'` and a scalar numeric.

Data Types: `double`

`VarianceGridSize` — Number of nodes for variance grid for finite difference grid
`200` (default) | scalar numeric

Number of nodes for the variance grid for finite difference grid, specified as the comma-separated pair consisting of `'VarianceGridSize'` and a scalar numeric.

Data Types: `double`

`TimeGridSize` — Number of nodes of time grid for finite difference grid
`100` (default) | positive numeric scalar

Number of nodes of the time grid for finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive numeric scalar.

Data Types: `double`

`AmericanOpt` — Option type
`0` (European) (default) | scalar with values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar flag with one of these values:

- `0` — European

- 1 — American

Data Types: double

OutSpec — Define outputs
["price"] (default) | cell array of character vectors with values 'price', 'delta', 'gamma', 'vega', 'rho', 'theta', and 'vegalt' | string array with values "price", "delta", "gamma", "vega", "rho", "theta", and "vegalt"

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and an NOUT- by-1 or a 1-by-NOUT string array or cell array of character vectors with the supported values.

Note 'vega' is the sensitivity with respect to the initial volatility sqrt(V0). In contrast, 'vegalt' is the sensitivity with respect to the long-term volatility sqrt(ThetaV).

Example: OutSpec = {'price','delta','gamma','vega','rho','theta','vegalt'}

Data Types: string | cell

Output Arguments

PriceSens — Option price and sensitivities
scalar numeric

Option price and sensitivities, returned as a scalar numeric. OutSpec determines the types and order of the outputs.

PriceGrid — Grid containing prices calculated by the finite difference method
numeric

Grid containing prices calculated by the finite difference method, returned as a three-dimensional grid with size AssetGridSize VarianceGridSize TimeGridSize. The depth is not necessarily equal to the TimeGridSize, because exercise and ex-dividend dates are added to the time grid. PriceGrid(:, :, end) contains the price for $t = 0$.

AssetPrices — Prices of the asset
vector

Prices of the asset corresponding to the first dimension of PriceGrid, returned as a vector.

Variances — Variances
vector

Variances corresponding to the second dimension of PriceGrid, returned as a vector.

Times — Times
vector

Times corresponding to the third dimension of PriceGrid, returned as a vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Heston Stochastic Volatility Model

The Heston model is an extension of the Black-Scholes model, where the volatility (square root of variance) is no longer assumed to be constant, and the variance now follows a stochastic (CIR) process. This allows modeling the implied volatility smiles observed in the market.

The stochastic differential equation is:

$$dS_t = (r - q)S_t dt + \sqrt{v_t} S_t dW_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t

v_0 is the initial variance of the asset price at $t = 0$ for ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for the variance for ($\kappa > 0$).

σ_v is the volatility of the variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and $W^v{}_t$ for ($-1 \le p \le 1$).

Version History

Introduced in R2018b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optSensByHestonFD` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Heston, S. L. “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.” *The Review of Financial Studies.* Vol 6, Number 2, 1993.

See Also

`optstockbyfd` | `optstocksensbyfd` | `optByHestonFD` | `optByLocalVolFD` | `optSensByLocalVolFD` | `optByBatesFD` | `optSensByBatesFD` | `optByMertonFD` | `optSensByMertonFD` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optByHestonFFT

Option price by Heston model using FFT and FRFT

Syntax

```
[Price,StrikeOut] = optByHestonFFT(Rate,AssetPrice,Settle,Maturity,OptSpec,
Strike,V0,ThetaV,Kappa,SigmaV,RhoSV)
[Price,StrikeOut] = optByHestonFFT( ___ ,Name,Value)
```

Description

`[Price,StrikeOut] = optByHestonFFT(Rate,AssetPrice,Settle,Maturity,OptSpec, Strike,V0,ThetaV,Kappa,SigmaV,RhoSV)` computes vanilla European option price by Heston model, using Carr-Madan FFT and Chourdakis FRFT methods.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,StrikeOut] = optByHestonFFT( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Price Surface Using the Heston Model

Use `optByHestonFFT` to calibrate a FFT strike grid and then compute option prices and plot an option price surface.

Define Option Variables and Heston Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

V0 = 0.04;
ThetaV = 0.05;
Kappa = 1.0;
SigmaV = 0.2;
RhoSV = -0.7;
```

Compute the Option Prices for the Entire FFT (or FRFT) Strike Grid, Without Specifying "Strike"

Compute option prices and also output the corresponding strikes. If the `Strike` input is empty (`[]`), option prices will be computed on the entire FFT (or FRFT) strike grid. The FFT (or FRFT) strike grid is determined as `exp(log-strike grid)`, where each column of the log-strike grid has `NumFFT`

points with `LogStrikeStep` spacing that are roughly centered around each element of log(`AssetPrice`). The default value for `NumFFT` is 2^12. In addition to the prices in the first output, the optional last output contains the corresponding strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = []; % Strike is not specified (will use the entire FFT strike grid)

% Compute option prices for the entire FFT strike grid
[Call, Kout] =  optByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield);

% Show the lowest and highest strike values on the FFT strike grid
format
MinStrike = Kout(1) % Lowest possible strike in the current FFT strike grid
```

```
MinStrike = 2.9205e-135
```

```
MaxStrike = Kout(end) % Highest possible strike in the current FFT strike grid
```

```
MaxStrike = 1.8798e+138
```

```
% Show a subset of the strikes and corresponding option prices
Range = (2046:2052);
[Kout(Range) Call(Range)]
```

```
ans = 7×2

   50.4929   29.4843
   58.8640   21.3767
   68.6231   12.5614
   80.0000    4.7008
   93.2631    0.6496
  108.7251    0.0144
  126.7505    0.0001
```

Change the Number of FFT (or FRFT) Points, and Compare with optByHestonNI

Try a different number of FFT (or FRFT) points, and compare the results with direct numerical integration. Unlike `optByHestonFFT`, which uses FFT (or FRFT) techniques for fast computation across the whole range of strikes, the `optByHestonNI` function uses direct numerical integration and it is typically slower, especially for multiple strikes. However, the values computed by `optByHestonNI` can serve as a benchmark for adjusting the settings for `optByHestonFFT`.

```
% Try a smaller number of FFT (or FRFT) points
% (e.g. for faster performance or smaller memory footprint)
NumFFT = 2^10; % Smaller than the default value of 2^12
Strike = []; % Strike is not specified (will use the entire FFT strike grid)
[Call, Kout] = optByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'NumFFT', NumFFT);

% Compare with numerical integration method
Range = (510:516);
Strike = Kout(Range);
CallFFT = Call(Range);
CallNI = optByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ...
```

```
    ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield);
Error = abs(CallFFT-CallNI);
table(Strike, CallFFT, CallNI, Error)
```

```
ans=7×4 table
    Strike     CallFFT       CallNI         Error
    ______    _________    ___________    _________

    12.696       66.066          66.696      0.62964
    23.449       55.766          56.103      0.33672
    43.312       36.359          36.539      0.17974
        80       4.7727          4.7007     0.071928
    147.76     0.066156     2.3472e-08     0.066156
    272.93     0.013271    -2.5036e-09     0.013271
    504.11    0.0034504    -3.0876e-07    0.0034508
```

Make Further Adjustments to FFT (or FRFT)

If the values in the output `CallFFT` are significantly different from those in `CallNI`, try making adjustments to `optByHestonFFT` settings, such as `CharacteristicFcnStep`, `LogStrikeStep`, `NumFFT`, `DampingFactor`, and so on. Note that if (`LogStrikeStep` * `CharacteristicFcnStep`) is 2*pi/ `NumFFT`, FFT is used. Otherwise, FRFT is used.

```
Strike = []; % Strike is not specified (will use the entire FFT or FRFT strike grid)
[Call, Kout] = optByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001);

% Compare with numerical integration method
Strike = Kout(Range);
CallFFT = Call(Range);
CallNI = optByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, V0, ...
    ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield);
Error = abs(CallFFT-CallNI);
table(Strike, CallFFT, CallNI, Error)
```

```
ans=7×4 table
    Strike    CallFFT    CallNI      Error
    ______    _______    ______    __________

    79.76      4.826      4.826    2.7708e-08
    79.84     4.7841     4.7841    3.0111e-08
    79.92     4.7423     4.7423    3.2376e-08
       80     4.7007     4.7007    3.4496e-08
    80.08     4.6593     4.6593    3.6457e-08
    80.16     4.6181     4.6181    3.8253e-08
    80.24      4.577      4.577    3.9872e-08
```

```
% Save the final FFT (or FRFT) strike grid for future reference. For
% example, it provides information about the range of Strike inputs for
% which the FFT (or FRFT) operation is valid.
FFTStrikeGrid = Kout;
MinStrike = FFTStrikeGrid(1) % Strike cannot be less than MinStrike
```

```
MinStrike = 47.9437
```

```
MaxStrike = FFTStrikeGrid(end) % Strike cannot be greater than MaxStrike
```

```
MaxStrike = 133.3566
```

Compute the Option Price for a Single Strike

Once the desired FFT (or FRFT) settings are determined, use the `Strike` input to specify the strikes rather than providing an empty array. If the specified strikes do not match a value on the FFT (or FRFT) strike grid, the outputs are interpolated on the specified strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;

Call =  optByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001)
```

```
Call = 4.7007
```

Compute the Option Prices for a Vector of Strikes

Use the `Strike` input to specify the strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Call =  optByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001)
```

```
Call = 5×1

    7.0401
    5.8053
    4.7007
    3.7316
    2.8991
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Call =  optByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001) % Five values in vector output
```

```
Call = 5×1

    8.9560
```

```
    9.3419
    9.6240
    9.8560
   10.0500
```

Expand the Outputs for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the outputs into `NStrikes`-by `NMaturities` matrices. In this case, they are square matrices.

```
[Call, Kout] =  optByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'ExpandOutput', true) % (5 x 5) matrix outputs
```

```
Call = 5×5

    8.9560   10.4543   11.7058   12.8009   13.7728
    7.7946    9.3419   10.6337   11.7644   12.7685
    6.7244    8.3028    9.6240   10.7828   11.8134
    5.7475    7.3379    8.6771    9.8560   10.9074
    4.8645    6.4474    7.7930    8.9840   10.0500


Kout = 5×5

    76    76    76    76    76
    78    78    78    78    78
    80    80    80    80    80
    82    82    82    82    82
    84    84    84    84    84
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities` (that is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular).

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Call =  optByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'ExpandOutput', true) % (5 x 6) matrix output
```

```
Call = 5×6

    7.0401    8.9560   10.4543   11.7058   12.8009   13.7728
    5.8053    7.7946    9.3419   10.6337   11.7644   12.7685
    4.7007    6.7244    8.3028    9.6240   10.7828   11.8134
    3.7316    5.7475    7.3379    8.6771    9.8560   10.9074
    2.8991    4.8645    6.4474    7.7930    8.9840   10.0500
```

Compute the Option Prices for a Vector of Strikes and a Vector of Asset Prices

When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

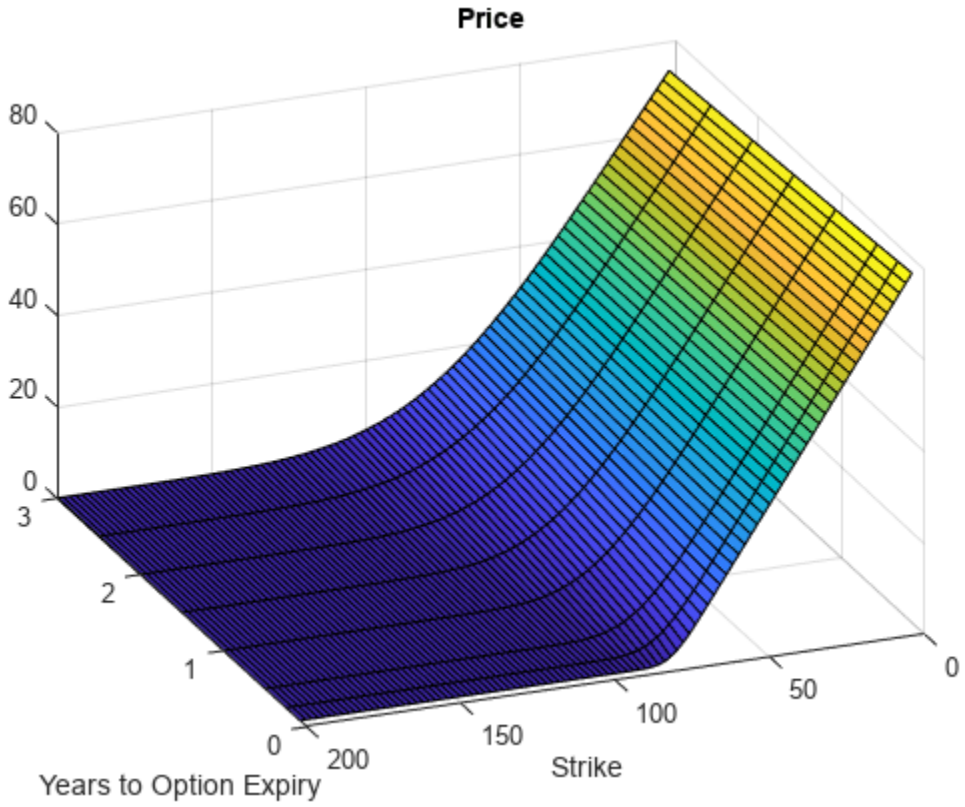
```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Call =  optByHestonFFT(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'ExpandOutput', true) % (5 x 4) matrix output
```

```
Call = 5×4

    3.2944    5.8047    8.9560   12.6052
    2.6413    4.8810    7.7946   11.2507
    2.0864    4.0575    6.7244    9.9738
    1.6230    3.3325    5.7475    8.7783
    1.2429    2.7028    4.8645    7.6676
```

Plot an Option Price Surface

Use the `Strike` input to specify the strikes. Increase the value for `NumFFT` to support a wider range of strikes. Also, the `Maturity` input can be a vector. Set `ExpandOutput` to `"true"` to output the surface as a `NStrikes`-by-`NMaturities` matrix.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

% Increase 'NumFFT' to support a wider range of strikes
NumFFT = 2^13;

Call =  optByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, 'ExpandOutput', true);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Call);
title('Price');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
zlim([0 80]);
```



Input Arguments

`Rate` — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

numeric

Current underlying asset price, specified as numeric value using a scalar or a `NINST`-by-`1` or `NColumns`-by-`1` vector.

For more information on the proper dimensions for `AssetPrice`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Settle` — Option settlement date

datetime array | string array | date character vector

Option settlement date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `optByHestonFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Settle`, see the name-value pair argument `ExpandOutput`.

`Maturity` — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optByHestonFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Maturity`, see the name-value pair argument `ExpandOutput`.

`OptSpec` — Definition of option
cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a cell array of character vectors or string arrays with values `"call"` or `"put"`.

For more information on the proper dimensions for `OptSpec`, see the name-value pair argument `ExpandOutput`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a `NINST`-by-`1`, `NRows`-by-`1`, `NRows`-by-`NColumns` vector of strike prices.

If this input is an empty array (`[]`), option prices are computed on the entire FFT (or FRFT) strike grid, which is determined as `exp(log-strike grid)`. Each column of the log-strike grid has `'NumFFT'` points with `'LogStrikeStep'` spacing that are roughly centered around each element of `log(AssetPrice)`.

For more information on the proper dimensions for `Strike`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`V0` — Initial variance of underlying asset
numeric

Initial variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
numeric

Long-term variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

Kappa — Mean revision speed for the variance of underlying asset
numeric

Mean revision speed for the underling asset, specified as a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
numeric

Volatility of the variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,StrikeOut] = optByHestonFFT(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,'Basis',7)`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric value.

Data Types: `double`

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, specified as the comma-separated pair consisting of `'VolRiskPremium'` and a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating Little Heston Trap formulation
`true` (default) | logical with values `true` or `false`

Flag indicating Little Heston Trap formulation by Albrecher *et al*, specified as the comma-separated pair consisting of `'LittleTrap'` and a logical:

- `true` — Use the Albrecher *et al* formulation.
- `false` — Use the original Heston formation.

Data Types: `logical`

`NumFFT` — Number of grid points in the characteristic function variable
`4096` (default) | numeric

Number of grid points in the characteristic function variable and in each column of the log-strike grid, specified as the comma-separated pair consisting of `'NumFFT'` and a scalar numeric value.

Data Types: `double`

`CharacteristicFcnStep` — Characteristic function variable grid spacing
`0.01` (default) | numeric

Characteristic function variable grid spacing, specified as the comma-separated pair consisting of `'CharacteristicFcnStep'` and a scalar numeric value.

Data Types: `double`

`LogStrikeStep` — Log-strike grid spacing
`2*pi/NumFFT/CharacteristicFcnStep` (default) | numeric

Log-strike grid spacing, specified as the comma-separated pair consisting of `'LogStrikeStep'` and a scalar numeric value.

Note If (`LogStrikeStep*CharacteristicFcnStep`) is `2*pi/NumFFT`, FFT is used. Otherwise, FRFT is used.

Data Types: `double`

`DampingFactor` — Damping factor for Carr-Madan formulation
`1.5` (default) | numeric

Damping factor for Carr-Madan formulation, specified as the comma-separated pair consisting of `'DampingFactor'` and a scalar numeric value.

Data Types: `double`

`Quadrature` — Type of quadrature
`"simpson"` (default) | character vector with values: `'simpson'` or `'trapezoidal'` | string array with values: `"simpson"` or `"trapezoidal"`

Type of quadrature, specified as the comma-separated pair consisting of `'Quadrature'` and a single character vector or string array with a value of `'simpson'` or `'trapezoidal'`.

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-1 vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-1 vector, or a `NRows`-by-`NColumns` matrix. If `Strike` is empty, `NRows` is equal to `NumFFT`. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-1 vectors.
- `false` — If `false`, the outputs are `NINST`-by-1 vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-1 vectors.

Data Types: `logical`

Output Arguments

`Price` — Option prices
numeric

Option prices, returned as a `NINST`-by-1, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

`StrikeOut` — Strikes corresponding to `Price`
numeric

Strikes corresponding to `Price`, returned as a `NINST`-by-1, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Heston Stochastic Volatility Model

The Heston model is an extension of the Black-Scholes model, where the volatility (square root of variance) is no longer assumed to be constant, and the variance now follows a stochastic (CIR) process. This process allows modeling the implied volatility smiles observed in the market.

The stochastic differential equation is:

$$dS_t = (r - q)S_t dt + \sqrt{v_t} S_t dW_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t

v_0 is the initial variance of the asset price at $t = 0$ for ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for the variance for ($\kappa > 0$).

σ_v is the volatility of the variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and $W^v{}_t$ for ($-1 \le p \le 1$).

The characteristic function $f_{Heston_j}(\phi)$ for $j = 1$ (asset price measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Heston_j}(\phi) = \exp(C_j + D_j v_0 + i\phi \ln S_t)$$

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1 - g_j e^{d_j\tau}}{1 - g_j}\right)\right]$$

$$D_j = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1 - e^{d_j\tau}}{1 - g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

where

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

The definitions for C_j and D_j under "The Little Heston Trap" by Albrecher et al. (2007) are:

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1 - \varepsilon_j e^{-d_j\tau}}{1 - \varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1 - e^{-d_j\tau}}{1 - \varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Carr-Madan Formulation

The Carr and Madan (1999) formulation is a popular modified implementation of Heston (1993) framework.

Rather than computing the probabilities P_1 and P_2 as intermediate steps, Carr and Madan developed an alternative expression so that taking its inverse Fourier transform gives the option price itself directly.

$$Call(k) = \frac{e^{-\alpha k}}{\pi}\int_0^{\infty} \mathrm{Re}\left[e^{-iuk}\psi(u)\right]du$$

$$\psi(u) = \frac{e^{-r\tau}f_2(\phi = (u-(\alpha+1)i))}{\alpha^2+\alpha-u^2+iu(2\alpha+1)}$$

$$Put(K) = Call(K) + Ke^{-r\tau} - S_t e^{-q\tau}$$

where

τ is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K

i is a unit imaginary number (i^2= -1)

ϕ is the characteristic function variable.

α is the damping factor.

u is the characteristic function variable for integration, where ϕ = (u - (α+1)i).

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

To apply FFT or FRFT to this formulation, the characteristic function variable for integration, u, is discretized into `NumFFT`(N) points with the step size `CharacteristicFcnStep` (Δu), and the log-strike k is discretized into N points with the step size `LogStrikeStep`(Δk).

The discretized characteristic function variable for integration, u_j(for j = 1,2,3,...,N), has a minimum value of 0 and a maximum value of (N-1) (Δu), and it approximates the continuous integration range from 0 to infinity.

The discretized log-strike grid, k_n(for n = 1, 2, 3, N) is approximately centered around `ln`(S_t), with a minimum value of

$$\ln(S_t) - \frac{N}{2}\Delta k$$

and a maximum value of

$$\ln(S_t) + \left(\frac{N}{2} - 1\right)\Delta k$$

Where the minimum allowable strike is

$$S_t \exp\left(-\frac{N}{2}\Delta k\right)$$

and the maximum allowable strike is

$$S_t \exp\left[\left(\frac{N}{2} - 1\right)\Delta k\right]$$

As a result of the discretization, the expression for the call option becomes

$$Call(k_n) = \Delta u \frac{e^{-\alpha k_n}}{\pi} \sum_{j=1}^{N} \mathrm{Re}\left[e^{-i\Delta k \Delta u (j-1)(n-1)} e^{iuj\left[\frac{N\Delta k}{2} - \ln(S_t)\right]} \psi(u_j)\right] w_j$$

where

Δu is the step size of discretized characteristic function variable for integration.

Δk is the step size of discretized log-strike.

N is the number of FFT or FRFT points.

w_j is the weights for quadrature used for approximating the integral.

FFT is used to evaluate the above expression if Δk and Δu are subject to the following constraint:

$$\Delta k \Delta u = \left(\frac{2\pi}{N}\right)$$

otherwise, the functions use the FRFT method described in Chourdakis (2005).

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optByHestonFFT` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Albrecher, H., Mayer, P., Schoutens, W., and Tistaert, J. "The Little Heston Trap." Working Paper, Linz and Graz University of Technology, K.U. Leuven, ING Financial Markets, 2006.

[2] Carr, P. and D.B. Madan. “Option Valuation using the Fast Fourier Transform.” *Journal of Computational Finance.* Vol 2. No. 4. 1999.

[3] Chourdakis, K. “Option Pricing Using Fractional FFT.” *Journal of Computational Finance.* 2005.

[4] Heston, S. L. “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.” *The Review of Financial Studies.* Vol 6. No. 2. 1993.

See Also

`optSensByHestonFFT` | `optByHestonNI` | `optSensByHestonNI` | `optByBatesFFT` | `optSensByBatesFFT` | `optByBatesNI` | `optSensByBatesNI` | `optByMertonFFT` | `optSensByMertonFFT` | `optByMertonNI` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optSensByHestonFFT

Option price and sensitivities by Heston model using FFT and FRFT

Syntax

```
[PriceSens,StrikeOut] = optSensByHestonFFT(Rate,AssetPrice,Settle,Maturity,
OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV)
[PriceSens,StrikeOut] = optSensByHestonFFT( ___ ,Name,Value)
```

Description

`[PriceSens,StrikeOut] = optSensByHestonFFT(Rate,AssetPrice,Settle,Maturity, OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV)` computes vanilla European option price and sensitivities by Heston model, using Carr-Madan FFT and Chourdakis FRFT methods.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,StrikeOut] = optSensByHestonFFT( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Sensitivity Surface Using the Heston Model

Use `optSensByHestonFFT` to calibrate the FFT strike grid for sensitivities, compute option sensitivities, and plot option sensitivity surfaces.

Define Option Variables and Heston Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

V0 = 0.04;
ThetaV = 0.05;
Kappa = 1.0;
SigmaV = 0.2;
RhoSV = -0.7;
```

Compute the Option Sensitivities for the Entire FFT (or FRFT) Strike Grid, Without Specifying "Strike"

Compute option sensitivities and also output the corresponding strikes. If the `Strike` input is empty (`[]`), option sensitivities will be computed on the entire FFT (or FRFT) strike grid. The FFT (or FRFT) strike grid is determined as `exp(log-strike grid)`, where each column of the log-strike

grid has NumFFT points with LogStrikeStep spacing that are roughly centered around each element of log(AssetPrice). The default value for NumFFT is 2^12. In addition to the sensitivities in the first output, the optional last output contains the corresponding strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = []; % Strike is not specified (will use the entire FFT strike grid)

% Compute option sensitivities for the entire FFT strike grid
[Delta, Kout] = optSensByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'OutSpec', "delta");

% Show the lowest and highest strike values on the FFT strike grid
format
MinStrike = Kout(1) % Lowest possible strike in the current FFT strike grid

MinStrike = 2.9205e-135

MaxStrike = Kout(end) % Highest possible strike in the current FFT strike grid

MaxStrike = 1.8798e+138


% Show a subset of the strikes and corresponding option sensitivities
Range = (2046:2052);
[Kout(Range) Delta(Range)]

ans = 7×2

   50.4929    0.9866
   58.8640    0.9671
   68.6231    0.8724
   80.0000    0.5775
   93.2631    0.1545
  108.7251    0.0059
  126.7505    0.0000
```

Change the Number of FFT (or FRFT) Points, and Compare With optSensByHestonNI

Try a different number of FFT (or FRFT) points, and compare the results with numerical integration. Unlike optSensByHestonFFT, which uses FFT (or FRFT) techniques for fast computation across the whole range of strikes, the optSensByHestonNI function uses direct numerical integration and it is typically slower, especially for multiple strikes. However, the values computed by optSensByHestonNI can serve as a benchmark for adjusting the settings for optSensByHestonFFT.

```
% Try a smaller number of FFT points
% (e.g. for faster performance or smaller memory footprint)
NumFFT = 2^10; % Smaller than the default value of 2^12
Strike = []; % Strike is not specified (will use the entire FFT strike grid)
[Delta, Kout] = optSensByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'NumFFT', NumFFT);

% Compare with numerical integration method
Range = (510:516);
Strike = Kout(Range);
```

```
DeltaFFT = Delta(Range);
DeltaNI = optSensByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'OutSpec', "delta");
Error = abs(DeltaFFT-DeltaNI);
table(Strike, DeltaFFT, DeltaNI, Error)
```

```
ans=7×4 table
    Strike     DeltaFFT      DeltaNI        Error
    ______    __________    __________    _________

    12.696       0.90066       0.99002       0.08936
    23.449       0.93635       0.99002      0.053677
    43.312       0.94796        0.9896      0.041645
        80       0.53274       0.57747      0.044733
    147.76     0.0032769      2.45e-08     0.0032769
    272.93    0.00098029    -1.399e-10    0.00098029
    504.11    0.00028151    5.2868e-10    0.00028151
```

Make Further Adjustments to FFT (or FRFT)

If the values in the output `DeltaFFT` are significantly different from those in `DeltaNI`, try making adjustments to `optSensByHestonFFT` settings, such as `CharacteristicFcnStep`, `LogStrikeStep`, `NumFFT`, `DampingFactor`, and so on. Note that if (`LogStrikeStep` * `CharacteristicFcnStep`) is 2*pi/ `NumFFT`, FFT is used. Otherwise, FRFT is used.

```
Strike = []; % Strike is not specified (will use the entire FFT or FRFT strike grid)
[Delta, Kout] = optSensByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001)

% Compare with numerical integration method
Strike = Kout(Range);
DeltaFFT = Delta(Range);
DeltaNI = optSensByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'OutSpec', "delta");
Error = abs(DeltaFFT-DeltaNI);
table(Strike, DeltaFFT, DeltaNI, Error)
```

```
ans=7×4 table
    Strike    DeltaFFT    DeltaNI      Error
    ______    ________    _______    __________

    79.76     0.58558     0.58558    3.0538e-08
    79.84     0.58289     0.58289    2.8865e-08
    79.92     0.58018     0.58018    2.7053e-08
       80     0.57747     0.57747    2.5111e-08
    80.08     0.57476     0.57476    2.3049e-08
    80.16     0.57203     0.57203    2.0875e-08
    80.24      0.5693      0.5693    1.8601e-08
```

```
% Save the final FFT (or FRFT) strike grid for future reference. For
% example, it provides information about the range of Strike inputs for
% which the FFT (or FRFT) operation is valid.
FFTStrikeGrid = Kout;
MinStrike = FFTStrikeGrid(1) % Strike cannot be less than MinStrike
```

```
MinStrike = 47.9437

MaxStrike = FFTStrikeGrid(end) % Strike cannot be greater than MaxStrike

MaxStrike = 133.3566
```

Compute the Option Sensitivity for a Single Strike

Once you determine FFT (or FRFT) settings, use the `Strike` input to specify the strikes rather than providing an empty array. If the specified strikes do not match a value on the FFT (or FRFT) strike grid, the outputs are interpolated on the specified strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;

Delta = optSensByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001)

Delta = 0.5775
```

Compute the Option Sensitivities for a Vector of Strikes

Use the `Strike` input to specify the strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Delta = optSensByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001)

Delta = 5×1

    0.7043
    0.6433
    0.5775
    0.5083
    0.4377
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Delta = optSensByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
```

```
    'OutSpec', "delta", 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001) % Five values in vector output
```

```
Delta = 5×1

    0.6848
    0.6413
    0.6095
    0.5841
    0.5631
```

Expand the Outputs for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the outputs into `NStrikes`-by-`NMaturities` matrices. In this case, they are square matrices.

```
[Delta, Kout] = optSensByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001, 'ExpandOutput', true) % (5 x 5) matrix output
```

```
Delta = 5×5

    0.6848    0.6762    0.6703    0.6654    0.6609
    0.6416    0.6413    0.6404    0.6390    0.6372
    0.5960    0.6048    0.6095    0.6119    0.6129
    0.5485    0.5671    0.5776    0.5841    0.5882
    0.4997    0.5286    0.5452    0.5559    0.5631


Kout = 5×5

    76    76    76    76    76
    78    78    78    78    78
    80    80    80    80    80
    82    82    82    82    82
    84    84    84    84    84
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities` (that is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular).

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Delta = optSensByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001, 'ExpandOutput', true) % (5 x 6) matrix output
```

```
Delta = 5×6

    0.7043    0.6848    0.6762    0.6703    0.6654    0.6609
    0.6433    0.6416    0.6413    0.6404    0.6390    0.6372
```

```
    0.5775    0.5960    0.6048    0.6095    0.6119    0.6129
    0.5083    0.5485    0.5671    0.5776    0.5841    0.5882
    0.4377    0.4997    0.5286    0.5452    0.5559    0.5631
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Asset Prices

When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

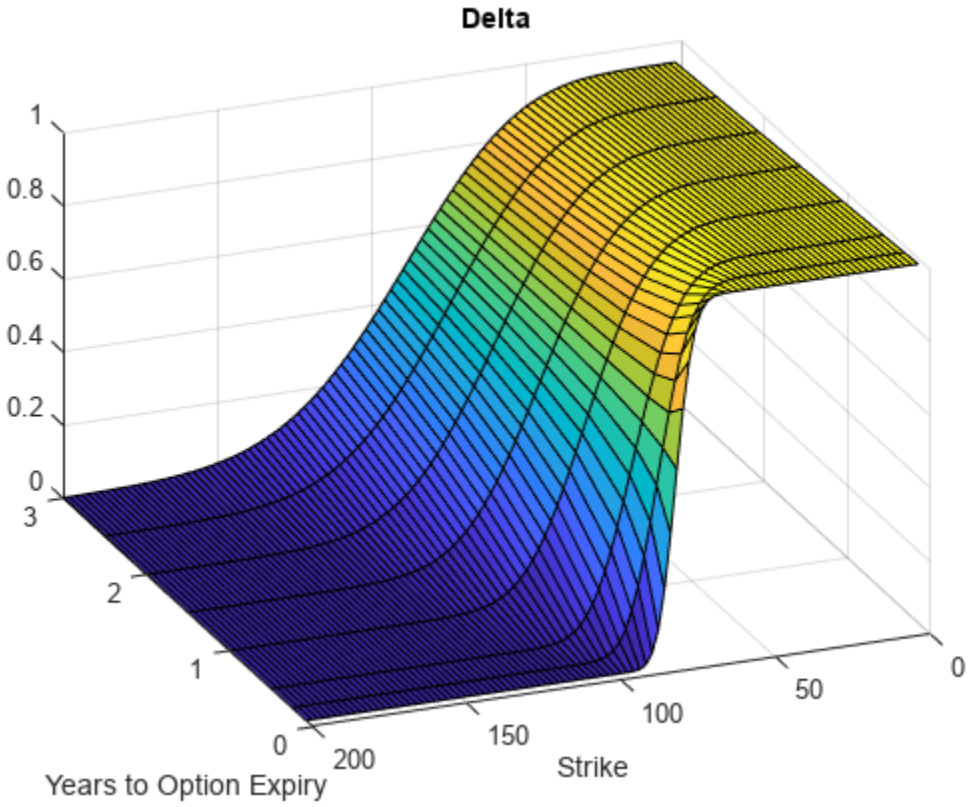
```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Delta = optSensByHestonFFT(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001, 'ExpandOutput', true) % (5 x 4) matrix output
```

```
Delta = 5×4

    0.4293    0.5708    0.6848    0.7705
    0.3737    0.5193    0.6416    0.7364
    0.3200    0.4668    0.5960    0.6994
    0.2693    0.4143    0.5485    0.6597
    0.2226    0.3628    0.4997    0.6177
```

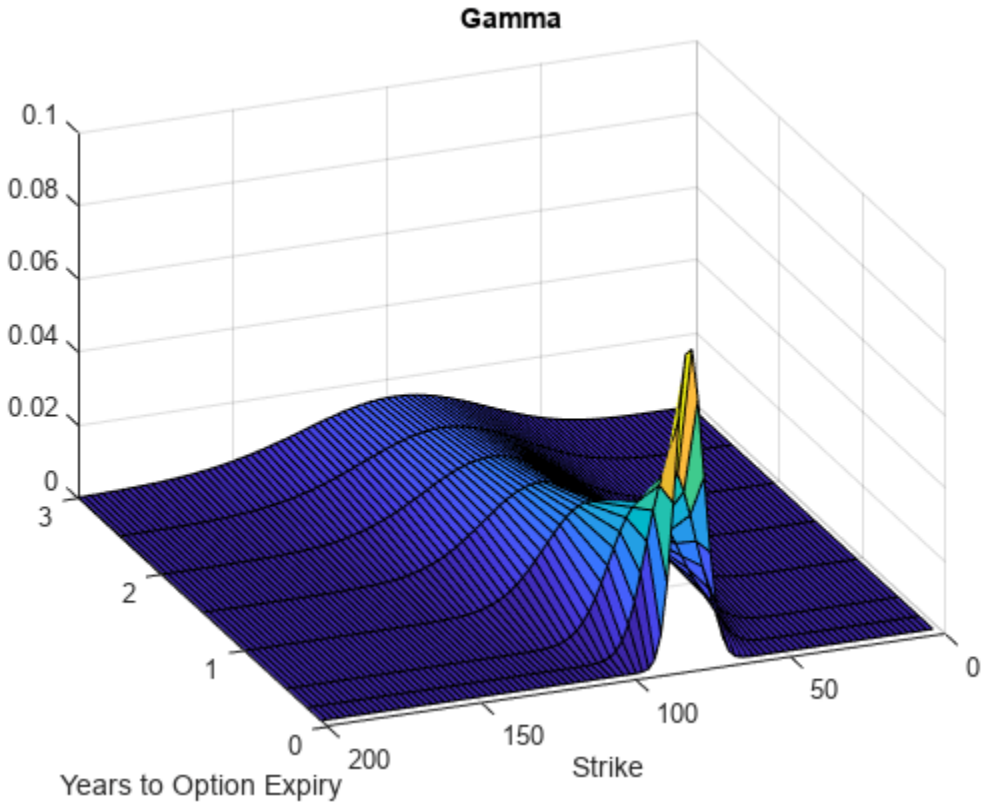
Plot Option Sensitivity Surfaces

Use the `Strike` input to specify the strikes. Increase the value for `NumFFT` to support a wider range of strikes. Also, the `Maturity` input can be a vector. Set `ExpandOutput` to `"true"` to output the surfaces as `NStrikes`-by-`NMaturities` matrices.

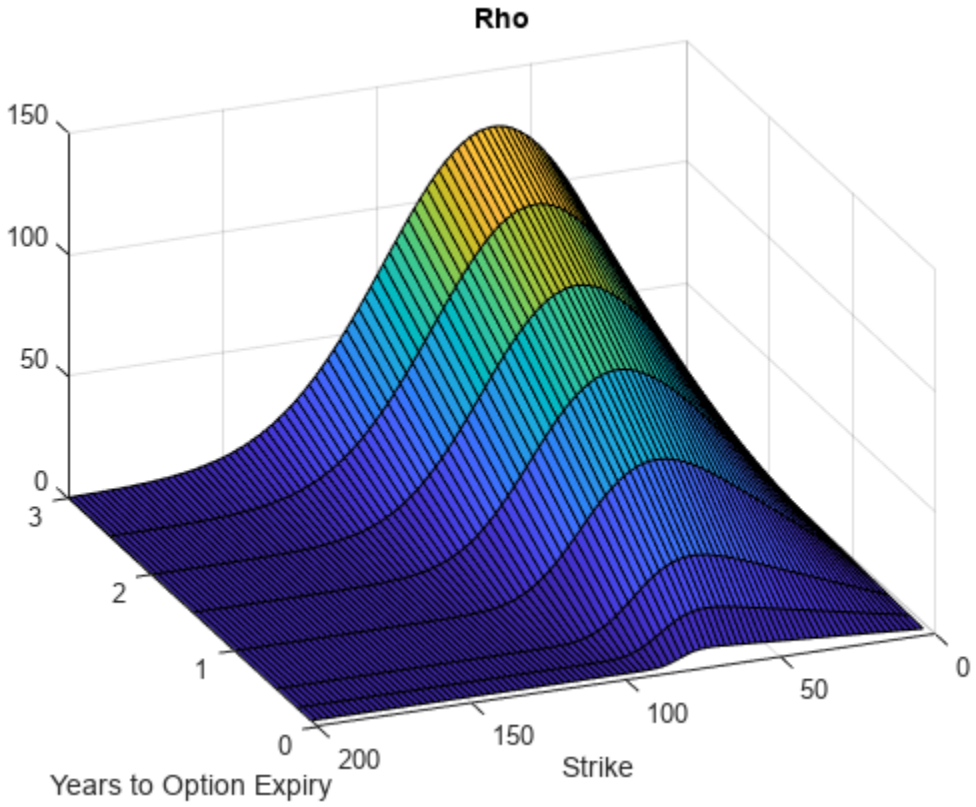
```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

% Increase 'NumFFT' to support a wider range of strikes
NumFFT = 2^13;

[Delta, Gamma, Rho, Theta, Vega, VegaLT] = ...
    optSensByHestonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, 'ExpandOutput', true, ...
    'OutSpec', ["delta", "gamma", "rho", "theta", "vega", "vegalt"]);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Delta);
title('Delta');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
```



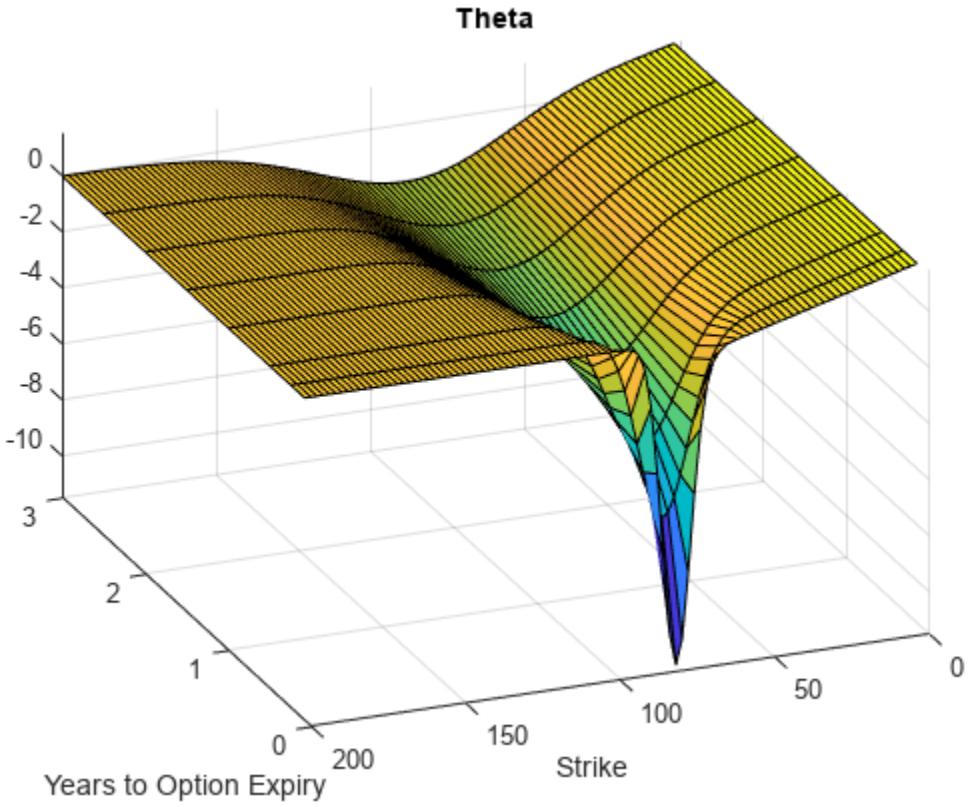

```
figure;
surf(X,Y,Gamma)
title('Gamma')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



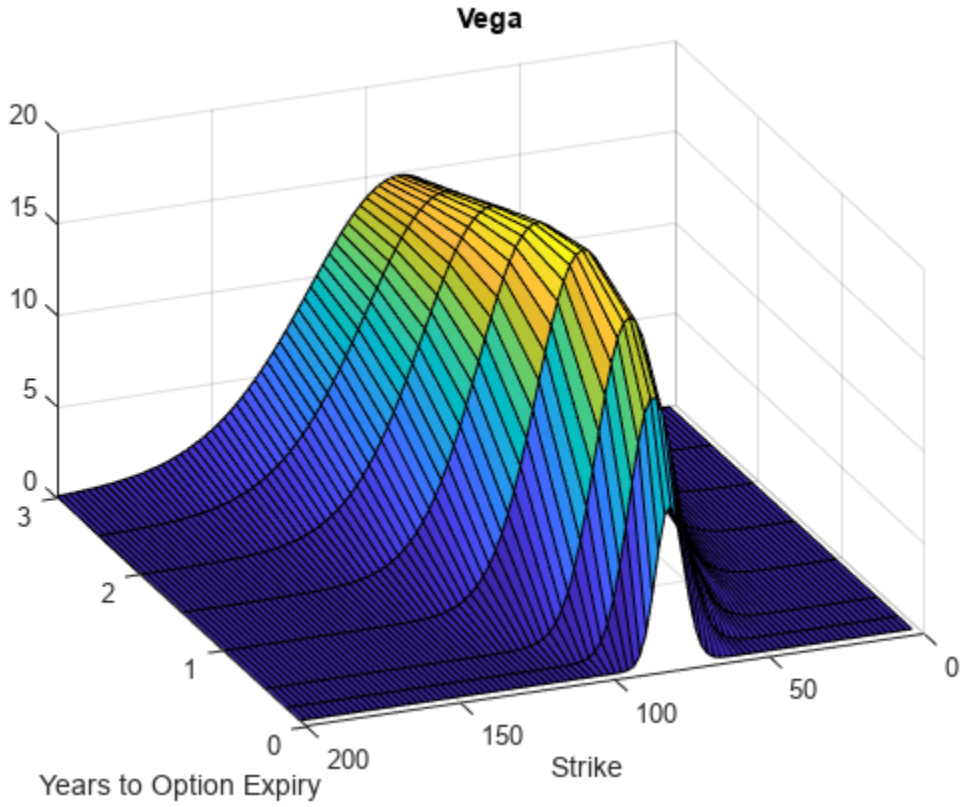

```
figure;
surf(X,Y,Rho)
title('Rho')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



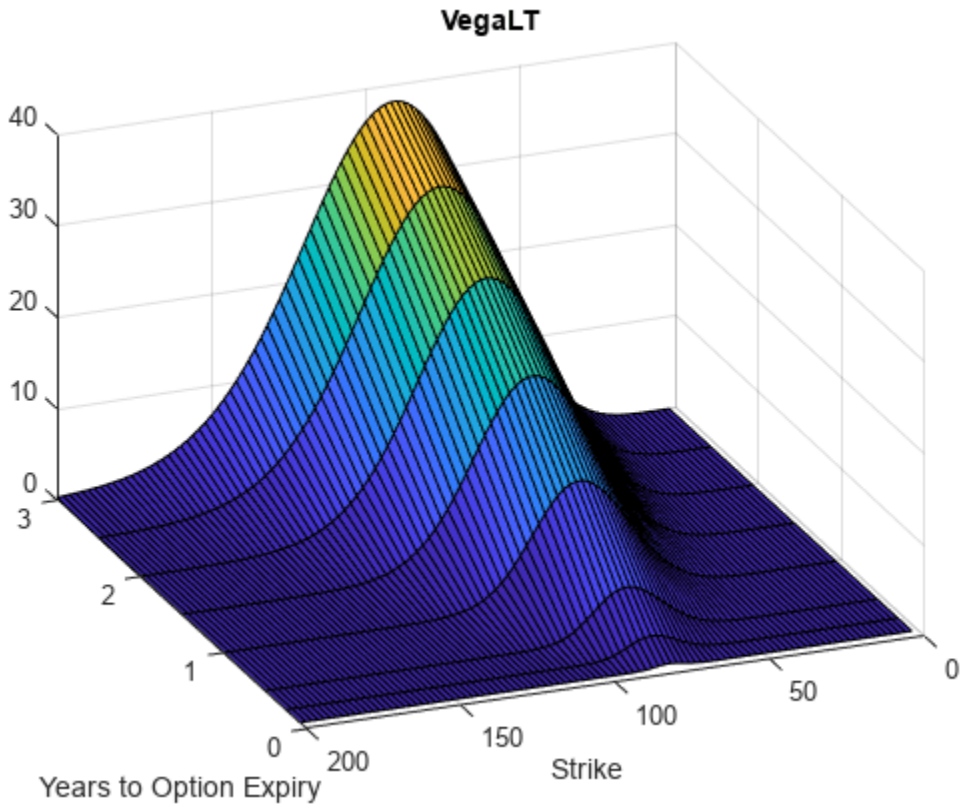

```
figure;
surf(X,Y,Theta)
title('Theta')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,Vega)
title('Vega')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,VegaLT)
title('VegaLT')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



Input Arguments

`Rate` — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

numeric

Current underlying asset price, specified as numeric value using a scalar or a `NINST`-by-`1` or `NColumns`-by-`1` vector.

For more information on the proper dimensions for `AssetPrice`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Settle` — Option settlement date

datetime array | string array | date character vector

Option settlement date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `optSensByHestonFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Settle`, see the name-value pair argument `ExpandOutput`.

`Maturity` — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optSensByHestonFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Maturity`, see the name-value pair argument `ExpandOutput`.

`OptSpec` — Definition of option
cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

For more information on the proper dimensions for `OptSpec`, see the name-value pair argument `ExpandOutput`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a `NINST`-by-`1`, `NRows`-by-`1`, `NRows`-by-`NColumns` vector of strike prices.

If this input is an empty array (`[]`), option prices are computed on the entire FFT (or FRFT) strike grid, which is determined as `exp(log-strike grid)`. Each column of the log-strike grid has `'NumFFT'` points with `'LogStrikeStep'` spacing that are roughly centered around each element of `log(AssetPrice)`.

For more information on the proper dimensions for `Strike`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`V0` — Initial variance of underlying asset
numeric

Initial variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
numeric

Long-term variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

Kappa — Mean revision speed for the variance of underlying asset
numeric

Mean revision speed for the underling asset, specified as a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
numeric

Volatility of the variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

RhoSV — Correlation between Weiner processes for underlying asset and its variance
numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PriceSens,StrikeOut] = optSensByHestonFFT(Rate, AssetPrice,Settle,Maturity,OptSpec,Strike,V0,ThetaV,Kappa,SigmaV,RhoSV,'Basis',7,'OptSpec',"vega")`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric value.

Data Types: `double`

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, specified as the comma-separated pair consisting of `'VolRiskPremium'` and a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating Little Heston Trap formulation
`true` (default) | logical with values `true` or `false`

Flag indicating Little Heston Trap formulation by Albrecher *et al*, specified as the comma-separated pair consisting of `'LittleTrap'` and a logical:

- `true` — Use the Albrecher *et al* formulation.
- `false` — Use the original Heston formation.

Data Types: `logical`

`OutSpec` — Define outputs
`["price"]` (default) | string array with values `"price"`, `"delta"`, `"gamma"`, `"vega"`, `"rho"`, `"theta"`, and `"vegalt"` | cell array of character vectors with values `'price'`, `'delta'`, `'gamma'`, `'vega'`, `'rho'`, `'theta'`, and `'vegalt'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` string array or cell array of character vectors with supported values.

Note `"vega"` is the sensitivity with respect the initial volatility sqrt(`V0`). In contrast, `"vegalt"` is the sensitivity with respect to the long-term volatility sqrt(`ThetaV`).

Example: `OutSpec = ["price","delta","gamma","vega","rho","theta","vegalt"]`

Data Types: `string` | `cell`

`NumFFT` — Number of grid points in the characteristic function variable
`4096` (default) | numeric

Number of grid points in the characteristic function variable and in each column of the log-strike grid, specified as the comma-separated pair consisting of `'NumFFT'` and a scalar numeric value.

Data Types: `double`

`CharacteristicFcnStep` — Characteristic function variable grid spacing
`0.01` (default) | numeric

Characteristic function variable grid spacing, specified as the comma-separated pair consisting of `'CharacteristicFcnStep'` and a scalar numeric value.

Data Types: `double`

`LogStrikeStep` — Log-strike grid spacing
`2*pi/NumFFT/CharacteristicFcnStep` (default) | numeric

Log-strike grid spacing, specified as the comma-separated pair consisting of `'LogStrikeStep'` and a scalar numeric value.

Note If (`LogStrikeStep*CharacteristicFcnStep`) is `2*pi/NumFFT`, FFT is used. Otherwise, FRFT is used.

Data Types: `double`

`DampingFactor` — Damping factor for Carr-Madan formulation
`1.5` (default) | numeric

Damping factor for Carr-Madan formulation, specified as the comma-separated pair consisting of `'DampingFactor'` and a scalar numeric value.

Data Types: `double`

`Quadrature` — Type of quadrature
`simpson` (default) | character vector with values:`simpson` or `trapezoidal` | string array with values: `simpson` or `trapezoidal`

Type of quadrature, specified as the comma-separated pair consisting of `'Quadrature'` and a single character vector or string array with a value of `simpson` or `trapezoidal`.

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-`1` vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-`1` vector, or a `NRows`-by-`NColumns` matrix. If `Strike` is empty, `NRows` is equal to `NumFFT`. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-`1` vectors.
- `false` — If `false`, the outputs are `NINST`-by-`1` vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-`1` vectors.

Data Types: `logical`

Output Arguments

`PriceSens` — Option prices or sensitivities

numeric

Option prices or sensitivities, returned as a `NINST`-by-`1`, or `NRows`-by-`NColumns`, depending on `ExpandOutput`. The name-value pair argument `OutSpec` determines the types and order of the outputs.

`StrikeOut` — Strikes corresponding to `Price`

numeric

Strikes corresponding to `Price`, returned as a `NINST`-by-`1`, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Heston Stochastic Volatility Model

The Heston model is an extension of the Black-Scholes model, where the volatility (square root of variance) is no longer assumed to be constant, and the variance now follows a stochastic (CIR) process. This process allows modeling the implied volatility smiles observed in the market.

The stochastic differential equation is:

$$dS_t = (r - q)S_t dt + \sqrt{v_t} S_t dW_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = \rho dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t

v_0 is the initial variance of the asset price at $t = 0$ for ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for the variance for ($\kappa > 0$).

σ_v is the volatility of the variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and $W^v{}_t$ for ($-1 \le p \le 1$).

The characteristic function $f_{Heston_j}(\phi)$ for $j = 1$ (asset price measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Heston_j}(\phi) = \exp(C_j + D_j v_0 + i\phi \ln S_t)$$

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1 - g_j e^{d_j\tau}}{1 - g_j}\right)\right]$$

$$D_j = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1 - e^{d_j\tau}}{1 - g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

where

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

The definitions for C_j and D_j under “The Little Heston Trap” by Albrecher et al. (2007) are:

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1-\varepsilon_j e^{-d_j\tau}}{1-\varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1-e^{-d_j\tau}}{1-\varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Carr-Madan Formulation

The Carr and Madan (1999) formulation is a popular modified implementation of Heston (1993) framework.

Rather than computing the probabilities P_1 and P_2 as intermediate steps, Carr and Madan developed an alternative expression so that taking its inverse Fourier transform gives the option price itself directly.

$$Call(k) = \frac{e^{-\alpha k}}{\pi}\int_0^\infty \mathrm{Re}\left[e^{-iuk}\psi(u)\right]du$$

$$\psi(u) = \frac{e^{-r\tau}f_2(\phi = (u-(\alpha+1)i))}{\alpha^2+\alpha-u^2+iu(2\alpha+1)}$$

$$Put(K) = Call(K) + Ke^{-r\tau} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number (i^2= -1).

ϕ is the characteristic function variable.

α is the damping factor.

u is the characteristic function variable for integration, where ϕ = (u - (α+1)i).

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

To apply FFT or FRFT to this formulation, the characteristic function variable for integration, u, is discretized into `NumFFT`(N) points with the step size `CharacteristicFcnStep` (Δu), and the log-strike k is discretized into N points with the step size `LogStrikeStep`(Δk).

The discretized characteristic function variable for integration, u_j(for j = 1,2,3,...,N), has a minimum value of 0 and a maximum value of (N-1) (Δu), and it approximates the continuous integration range from 0 to infinity.

The discretized log-strike grid, k_n(for n = 1, 2, 3, N) is approximately centered around `ln`(S_t), with a minimum value of

$$\ln(S_t) - \frac{N}{2}\Delta k$$

and a maximum value of

$$\ln(S_t) + \left(\frac{N}{2} - 1\right)\Delta k$$

Where the minimum allowable strike is

$$S_t \exp\left(-\frac{N}{2}\Delta k\right)$$

and the maximum allowable strike is

$$S_t \exp\left[\left(\frac{N}{2} - 1\right)\Delta k\right]$$

As a result of the discretization, the expression for the call option becomes

$$Call(k_n) = \Delta u \frac{e^{-\alpha k_n}}{\pi} \sum_{j=1}^{N} \mathrm{Re}\left[e^{-i\Delta k \Delta u (j-1)(n-1)} e^{iu_j\left[\frac{N\Delta k}{2} - \ln(S_t)\right]} \psi(u_j)\right] w_j$$

where

Δu is the step size of discretized characteristic function variable for integration.

Δk is the step size of discretized log-strike.

N is the number of FFT or FRFT points.

w_j is the weights for quadrature used for approximating the integral.

FFT is used to evaluate the above expression if Δk and Δu are subject to the following constraint:

$$\Delta k \Delta u = \left(\frac{2\pi}{N}\right)$$

otherwise, the functions use the FRFT method described in Chourdakis (2005).

Version History

Introduced in R2018a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `optSensByHestonFFT` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Albrecher, H., Mayer, P., Schoutens, W., and Tistaert, J. "The Little Heston Trap." Working Paper, Linz and Graz University of Technology, K.U. Leuven, ING Financial Markets, 2006.

[2] Carr, P. and D.B. Madan. “Option Valuation using the Fast Fourier Transform.” *Journal of Computational Finance.* Vol 2. No. 4. 1999.

[3] Chourdakis, K. “Option Pricing Using Fractional FFT.” *Journal of Computational Finance.* 2005.

[4] Heston, S. L. “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.” *The Review of Financial Studies.* Vol 6. No. 2. 1993.

See Also

`optByHestonFFT` | `optByHestonNI` | `optSensByHestonNI` | `optByBatesFFT` | `optSensByBatesFFT` | `optByBatesNI` | `optSensByBatesNI` | `optByMertonFFT` | `optSensByMertonFFT` | `optByMertonNI` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optByHestonNI

Option price by Heston model using numerical integration

Syntax

```
Price = optByHestonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,V0,
ThetaV,Kappa,SigmaV,RhoSV)
Price = optByHestonNI( ___ ,Name,Value)
```

Description

`Price = optByHestonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,V0, ThetaV,Kappa,SigmaV,RhoSV)` computes vanilla European option price by Heston model, using numerical integration methods.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = optByHestonNI( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Price Surface Using the Heston Model

`optByHestonNI` uses numerical integration to compute option prices and then to plot an option price surface.

Define Option Variables and Heston Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

V0 = 0.04;
ThetaV = 0.05;
Kappa = 1.0;
SigmaV = 0.2;
RhoSV = -0.7;
```

Compute the Option Price for a Single Strike

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;

Call = optByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield)
```

```
Call = 4.7007
```

Compute the Option Prices for a Vector of Strikes

The `Strike` input can be a vector.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Call = optByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield)
```

```
Call = 5×1

    7.0401
    5.8053
    4.7007
    3.7316
    2.8991
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Call = optByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield)
```

```
Call = 5×1

    8.9560
    9.3419
    9.6240
    9.8560
   10.0500


    % Five values in vector output
```

Expand the Output for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the output into a `NStrikes`-by-`NMaturities` matrix. In this case, it is a square matrix.

```
Call = optByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'ExpandOutput', true) % (5 x 5) matrix output
```

```
Call = 5×5

    8.9560   10.4543   11.7058   12.8009   13.7728
```

```
    7.7946    9.3419   10.6337   11.7644   12.7685
    6.7244    8.3028    9.6240   10.7828   11.8134
    5.7474    7.3378    8.6771    9.8560   10.9074
    4.8645    6.4474    7.7930    8.9840   10.0500
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities` (that is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular).

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Call = optByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'ExpandOutput', true) % (5 x 6) matrix output
```

```
Call = 5×6

    7.0401    8.9560   10.4543   11.7058   12.8009   13.7728
    5.8053    7.7946    9.3419   10.6337   11.7644   12.7685
    4.7007    6.7244    8.3028    9.6240   10.7828   11.8134
    3.7316    5.7474    7.3378    8.6771    9.8560   10.9074
    2.8991    4.8645    6.4474    7.7930    8.9840   10.0500
```

Compute the Option Prices for a Vector of Strikes and a Vector of Asset Prices

When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Call = optByHestonNI(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'ExpandOutput', true) % (5 x 4) matrix output
```

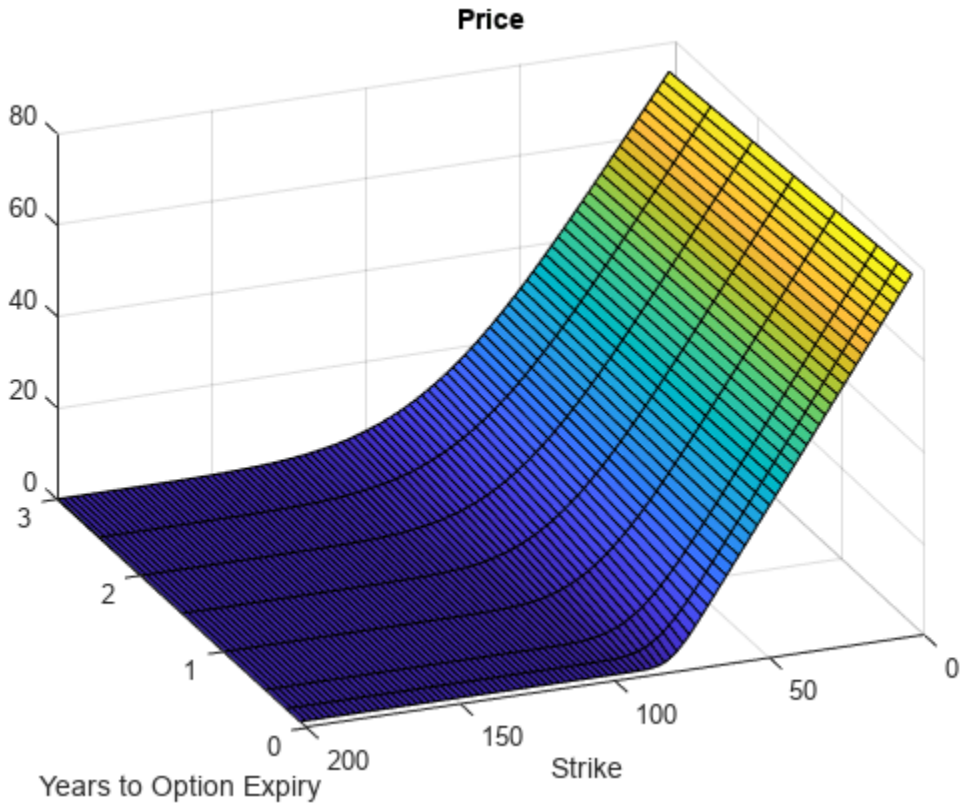
```
Call = 5×4

    3.2944    5.8047    8.9560   12.6052
    2.6413    4.8810    7.7946   11.2507
    2.0864    4.0575    6.7244    9.9738
    1.6230    3.3325    5.7474    8.7783
    1.2429    2.7028    4.8645    7.6676
```

Plot an Option Price Surface

The `Strike` and `Maturity` inputs can be vectors. Set `ExpandOutput` to `"true"` to output the surface as a `NStrikes`-by-`NMaturities` matrix.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
```

```
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

Call = optByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'ExpandOutput', true);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Call);
title('Price');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
zlim([0 80]);
```



Input Arguments

Rate — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

AssetPrice — Current underlying asset price
numeric

Current underlying asset price, specified as numeric value using a scalar or a NINST-by-1 or NColumns-by-1 vector.

For more information on the proper dimensions for AssetPrice, see the name-value pair argument ExpandOutput.

Data Types: double

Settle — Option settlement date
datetime array | string array | date character vector

Option settlement date, specified as a NINST-by-1 or NColumns-by-1 vector using a datetime array, string array, or date character vectors. The Settle date must be before the Maturity date.

To support existing code, optByHestonNI also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for Settle, see the name-value pair argument ExpandOutput.

Maturity — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a NINST-by-1 or NColumns-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optByHestonNI also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for Maturity, see the name-value pair argument ExpandOutput.

OptSpec — Definition of option
cell array of character vector with values 'call' or 'put' | string array with values 'call' or 'put'

Definition of the option as 'call' or 'put', specified as a NINST-by-1 or NColumns-by-1 vector using a cell array of character vectors or string arrays with values "call" or "put".

For more information on the proper dimensions for OptSpec, see the name-value pair argument ExpandOutput.

Data Types: cell | string

Strike — Option strike price value
numeric

Option strike price value, specified as a NINST-by-1, NRows-by-1, NRows-by-NColumns vector of strike prices.

For more information on the proper dimensions for Strike, see the name-value pair argument ExpandOutput.

Data Types: double

`V0` — Initial variance of underlying asset
numeric

Initial variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
numeric

Long-term variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`Kappa` — Mean revision speed for the variance of underlying asset
numeric

Mean revision speed for the underling asset, specified as a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
numeric

Volatility of the variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optByHestonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,V0,ThetaV,Kappa, SigmaV,RhoSV,'Basis',7,'Framework',"lewis2001")`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360

- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric value.

Data Types: `double`

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, specified as the comma-separated pair consisting of `'VolRiskPremium'` and a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating Little Heston Trap formulation
`true` (default) | logical with values `true` or `false`

Flag indicating Little Heston Trap formulation by Albrecher *et al*, specified as the comma-separated pair consisting of `'LittleTrap'`and a logical:

- `true` — Use the Albrecher *et al* formulation.
- `false` — Use the original Heston formation.

Data Types: `logical`

`AbsTol` — Absolute error tolerance for numerical integration
`1e-10` (default) | numeric

Absolute error tolerance for numerical integration, specified as the comma-separated pair consisting of `'AbsTol'` and a scalar numeric value.

Data Types: `double`

`RelTol` — Relative error tolerance for numerical integration
`1e-6` (default) | numeric

Relative error tolerance for numerical integration, specified as the comma-separated pair consisting of `'RelTol'` and a scalar numeric value.

Data Types: `double`

`IntegrationRange` — Numerical integration range used to approximate continuous integral over `[0 Inf]`
`[1e-9 Inf]` (default) | vector

Numerical integration range used to approximate the continuous integral over `[0 Inf]`, specified as the comma-separated pair consisting of `'IntegrationRange'` and a 1-by-2 vector representing `[LowerLimit UpperLimit]`.

Data Types: `double`

`Framework` — Framework for computing option prices and sensitivities using numerical integration of models
`"heston1993"` (default) | string with values `"heston1993"` or `"lewis2001"` | character vector with values `'heston1993'` or `'lewis2001'`

Framework for computing option prices and sensitivities using numerical integration of models, specified as the comma-separated pair consisting of `'Framework'` and a scalar string or character vector with the following values:

- `"heston1993"` or `'heston1993'` — Method used in Heston (1993)
- `"lewis2001"` or `'lewis2001'` — Method used in Lewis (2001)

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-1 vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-1 vector, or a `NRows`-by-`NColumns` matrix. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-1 vectors.
- `false` — If `false`, the outputs are `NINST`-by-1 vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-1 vectors.

Data Types: `logical`

Output Arguments

`Price` — Option prices
numeric

Option prices, returned as a `NINST`-by-1, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Heston Stochastic Volatility Model

The Heston model is an extension of the Black-Scholes model, where the volatility (square root of variance) is no longer assumed to be constant, and the variance now follows a stochastic (CIR) process. This allows modeling the implied volatility smiles observed in the market.

The stochastic differential equation is:

$$dS_t = (r - q)S_t dt + \sqrt{v_t} S_t dW_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t

v_0 is the initial variance of the asset price at $t = 0$ for ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for the variance for ($\kappa > 0$).

σ_v is the volatility of the variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and $W^v{}_t$ for ($-1 \le p \le 1$).

The characteristic function $f_{Heston_j}(\phi)$ for $j = 1$ (asset price measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Heston_j}(\phi) = \exp(C_j + D_j v_0 + i\phi \ln S_t)$$

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1 - g_j e^{d_j\tau}}{1 - g_j}\right)\right]$$

$$D_j = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1 - e^{d_j\tau}}{1 - g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

where

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

The definitions for C_j and D_j under "The Little Heston Trap" by Albrecher et al. (2007) are:

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v 2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1 - \varepsilon_j e^{-d_j\tau}}{1 - \varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1 - e^{-d_j\tau}}{1 - \varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Numerical Integration Method Under Heston (1993) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Heston (1993) framework is based on the following expressions:

$$Call(K) = S_t e^{-q\tau} P_1 - K e^{-r\tau} P_2$$

$$Put(K) = Call(K) + K e^{-r\tau} - S_t e^{-q\tau}$$

$$P_j = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \mathrm{Re}\left[\frac{e^{-i\phi\ln(K)} f_j(\phi)}{i\phi}\right] d\phi$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number (i^2= -1).

ϕ is the characteristic function variable.

$f_j(\phi)$ is the characteristic function for $P_j(j = 1,2)$.

P_1 is the probability of $S_t > K$ under the asset price measure for the model.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

Where $j = 1,2$ so that $f_1(\phi)$ and $f_2(\phi)$ are the characteristic functions for probabilities P_1 and P_2, respectively.

This framework is chosen with the default value "`Heston1993`" for the `Framework` name-value pair argument.

Numerical Integration Method Under Lewis (2001) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Lewis (2001) framework is based on the following expressions:

$$Call(k) = S_t e^{-q\tau} - \frac{\sqrt{K}e^{-\tau t}}{\pi}\int_0^{\infty} \mathrm{Re}\left[K^{-iu} f_2\left(\phi = \left(u - \frac{i}{2}\right)\right)\frac{1}{u^2 + \frac{1}{4}}\right]du$$

$$Put(K) = Call(K) = Ke^{-\tau t} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity (τ = T-t).

Call(*K*) is the call price at strike *K*.

Put(*K*) is the put price at strike *K*.

i is a unit imaginary number (i^2= -1).

ϕ is the characteristic function variable.

u is the characteristic function variable for integration, where $\phi = \left(u - \frac{i}{2}\right)$.

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

This framework is chosen with the value "`Lewis2001`" for the `Framework` name-value pair argument.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optByHestonNI` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Heston, S. L. "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options." *The Review of Financial Studies.* Vol 6. No. 2. 1993.

[2] Lewis, A. L. "A Simple Option Formula for General Jump-Diffusion and Other Exponential Levy Processes." Envision Financial Systems and OptionCity.net, 2001.

See Also

`optByHestonFFT` | `optSensByHestonFFT` | `optSensByHestonNI` | `optByBatesFFT` | `optSensByBatesFFT` | `optByBatesNI` | `optSensByBatesNI` | `optByMertonFFT` | `optSensByMertonFFT` | `optByMertonNI` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optSensByHestonNI

Option price and sensitivities by Heston model using numerical integration

Syntax

```
PriceSens = optSensByHestonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,
V0,ThetaV,Kappa,SigmaV,RhoSV)
PriceSens = optSensByHestonNI( ___ ,Name,Value)
```

Description

`PriceSens = optSensByHestonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike, V0,ThetaV,Kappa,SigmaV,RhoSV)` computes vanilla European option price and sensitivities by Heston model, using numerical integration methods.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = optSensByHestonNI( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Sensitivity Surface Using the Heston Model

`optSensByHestonNI` uses numerical integration to compute option sensitivities and then to plot option sensitivity surfaces.

Define Option Variables and Heston Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

V0 = 0.04;
ThetaV = 0.05;
Kappa = 1.0;
SigmaV = 0.2;
RhoSV = -0.7;
```

Compute the Option Sensitivity for a Single Strike

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;
```

```
Delta = optSensByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'OutSpec', "delta")
```

```
Delta = 0.5775
```

Compute the Option Sensitivities for a Vector of Strikes

The `Strike` input can be a vector.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Delta = optSensByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, 'OutSpec', "delta")
```

```
Delta = 5×1

    0.7043
    0.6433
    0.5775
    0.5083
    0.4377
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Delta = optSensByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta") % Five values in vector output
```

```
Delta = 5×1

    0.6848
    0.6413
    0.6095
    0.5841
    0.5631
```

Expand the Output for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the output into a `NStrikes`-by-`NMaturities` matrix. In this case, it is a square matrix.

```
Delta = optSensByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'ExpandOutput', true) % (5 x 5) matrix output
```

```
Delta = 5×5
```

```
    0.6848    0.6762    0.6703    0.6654    0.6609
    0.6416    0.6413    0.6404    0.6390    0.6372
    0.5960    0.6048    0.6095    0.6119    0.6129
    0.5485    0.5671    0.5776    0.5841    0.5882
    0.4997    0.5286    0.5452    0.5559    0.5631
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities` (that is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular).

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Delta = optSensByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'ExpandOutput', true)  % (5 x 6) matrix output
```

```
Delta = 5×6

    0.7043    0.6848    0.6762    0.6703    0.6654    0.6609
    0.6433    0.6416    0.6413    0.6404    0.6390    0.6372
    0.5775    0.5960    0.6048    0.6095    0.6119    0.6129
    0.5083    0.5485    0.5671    0.5776    0.5841    0.5882
    0.4377    0.4997    0.5286    0.5452    0.5559    0.5631
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Asset Prices

When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

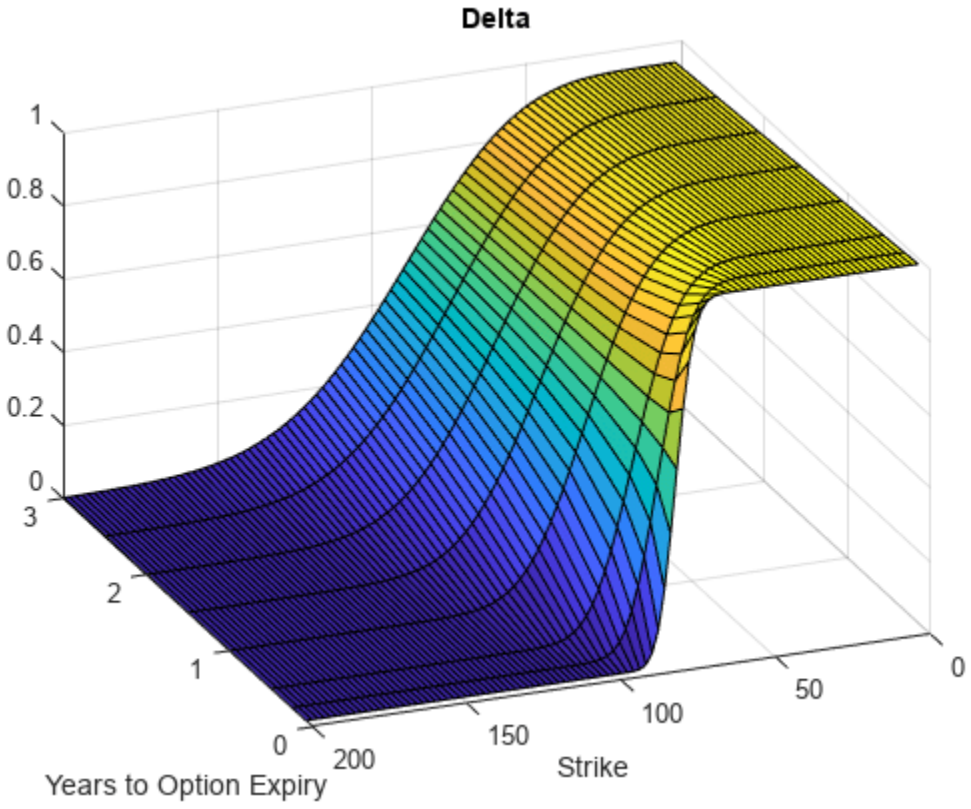
```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Delta = optSensByHestonNI(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'ExpandOutput', true) % (5 x 4) matrix output
```

```
Delta = 5×4

    0.4293    0.5708    0.6848    0.7705
    0.3737    0.5193    0.6416    0.7364
    0.3200    0.4668    0.5960    0.6994
    0.2693    0.4143    0.5485    0.6597
    0.2226    0.3628    0.4997    0.6177
```

Plot Option Sensitivity Surfaces

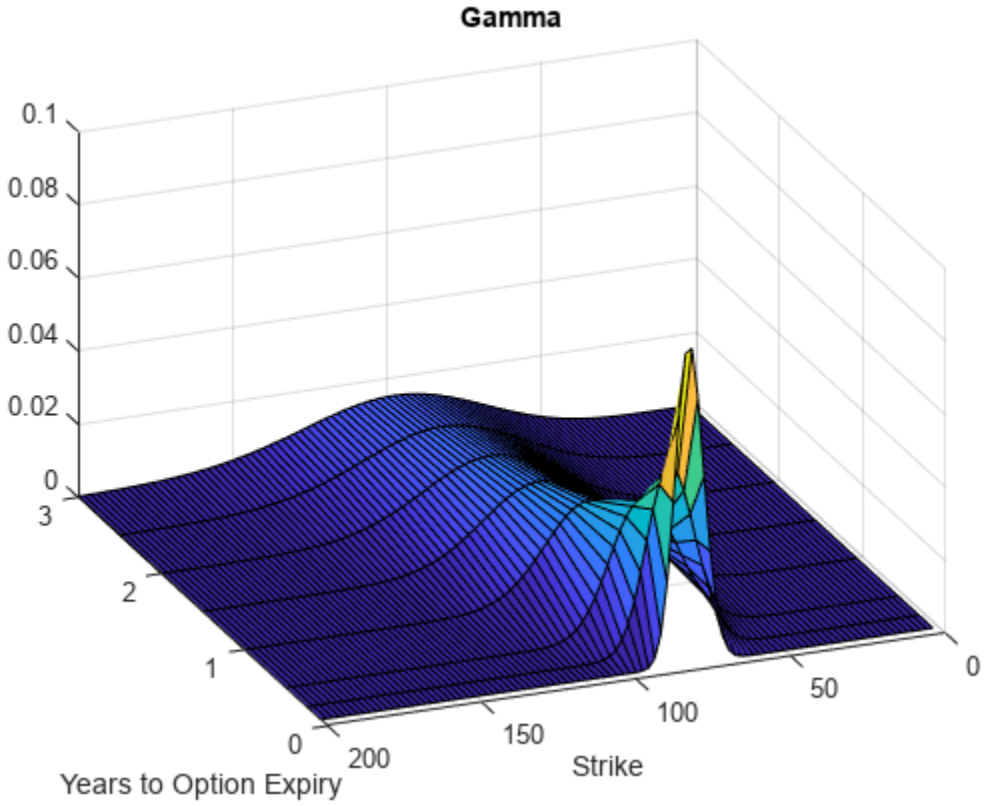
The `Strike` and `Maturity` inputs can be vectors. Set `ExpandOutput` to `"true"` to output the surfaces as `NStrikes`-by-`NMaturities` matrices.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

[Delta, Gamma, Rho, Theta, Vega, VegaLT] = ...
    optSensByHestonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    V0, ThetaV, Kappa, SigmaV, RhoSV, 'DividendYield', DividendYield, ...
    'OutSpec', ["delta", "gamma", "rho", "theta", "vega", "vegalt"], ...
    'ExpandOutput', true);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Delta);
title('Delta');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
```

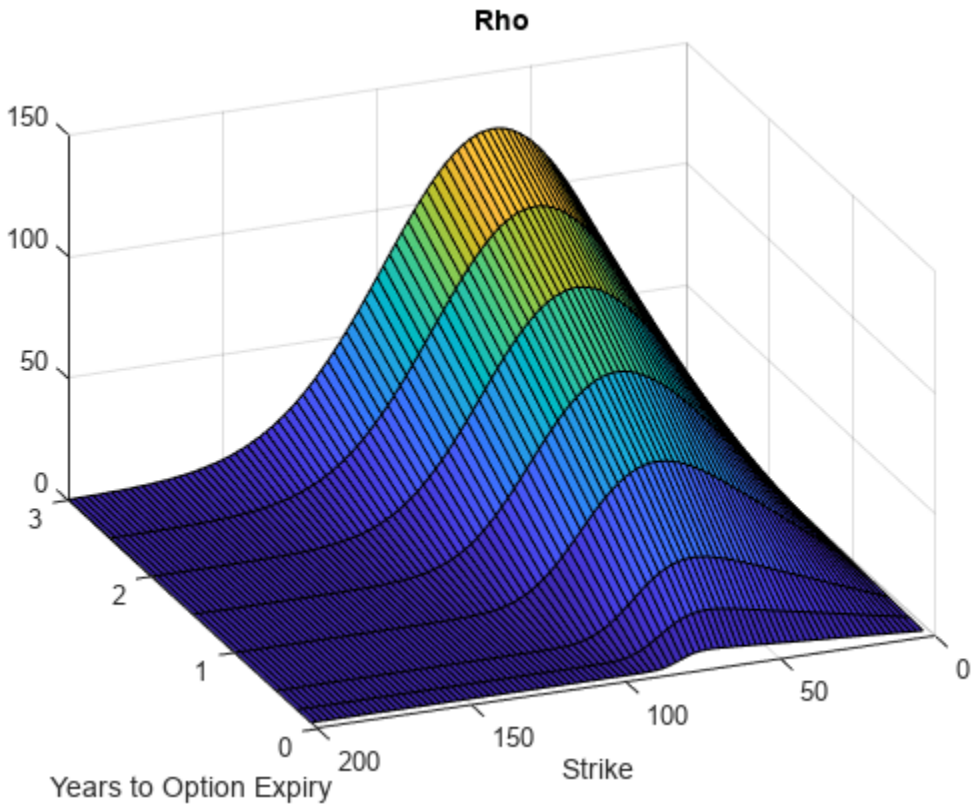



```
figure;
surf(X,Y,Gamma)
title('Gamma')
xlabel('Years to Option Expiry')
ylabel('Strike')
```

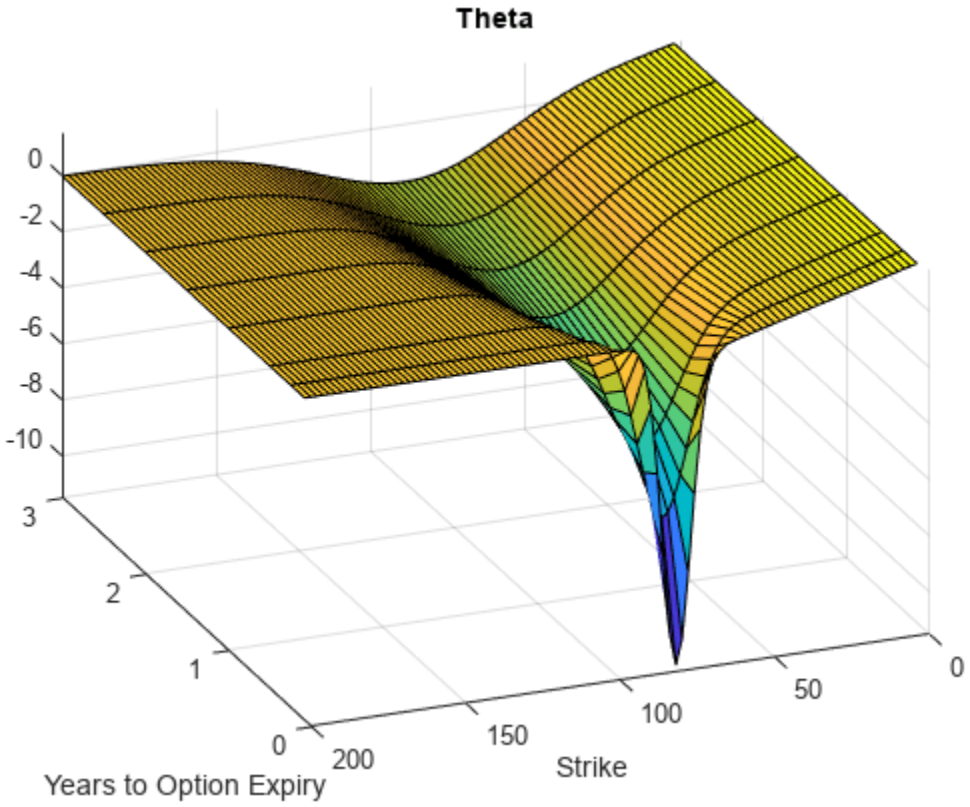
```
view(-112,34);
xlim([0 Times(end)]);
```



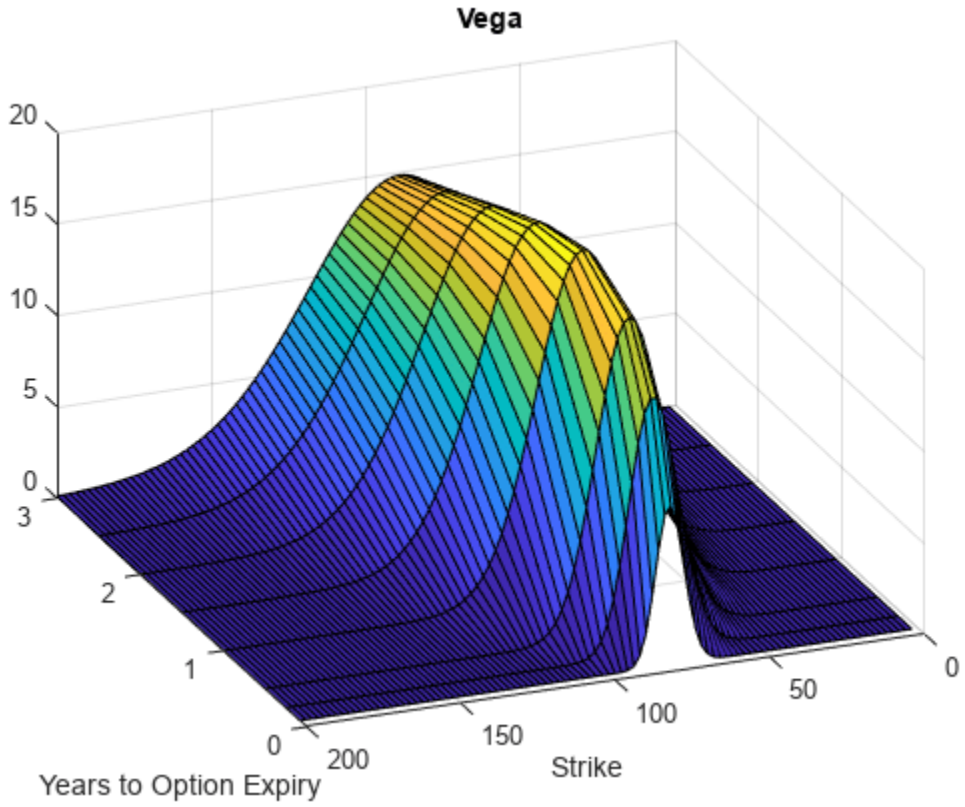

```
figure;
surf(X,Y,Rho)
title('Rho')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



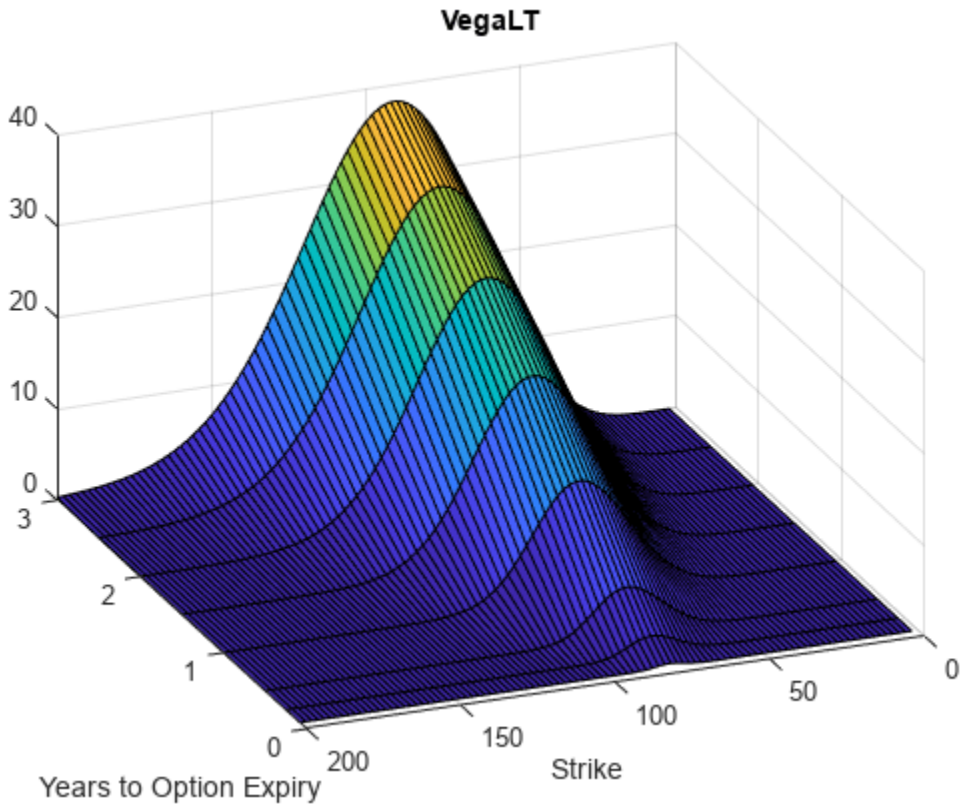

```
figure;
surf(X,Y,Theta)
title('Theta')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,Vega)
title('Vega')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,VegaLT)
title('VegaLT')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



Input Arguments

`Rate` — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

numeric

Current underlying asset price, specified as numeric value using a scalar or a `NINST`-by-`1` or `NColumns`-by-`1` vector.

For more information on the proper dimensions for `AssetPrice`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Settle` — Option settlement date

datetime array | string array | date character vector

Option settlement date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `optSensByHestonNI` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Settle`, see the name-value pair argument `ExpandOutput`.

`Maturity` — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optSensByHestonNI` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Maturity`, see the name-value pair argument `ExpandOutput`.

`OptSpec` — Definition of option
cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

For more information on the proper dimensions for `OptSpec`, see the name-value pair argument `ExpandOutput`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a `NINST`-by-`1`, `NRows`-by-`1`, `NRows`-by-`NColumns` vector of strike prices.

For more information on the proper dimensions for `Strike`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`V0` — Initial variance of underlying asset
numeric

Initial variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
numeric

Long-term variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`Kappa` — Mean revision speed for the variance of underlying asset
numeric

Mean revision speed for the underling asset, specified as a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
numeric

Volatility of the variance of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = optSensByHestonFFT(Rate, AssetPrice, Settle,Maturity, OptSpec, Strike, V0, ThetaV, Kappa, SigmaV, RhoSV,'Basis',7)`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

For more information, see “Basis” on page 2-228.

Data Types: double

DividendYield — Continuously compounded underlying asset yield
0 (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of 'DividendYield' and a scalar numeric value.

Data Types: double

VolRiskPremium — Volatility risk premium
0 (default) | numeric

Volatility risk premium, specified as the comma-separated pair consisting of 'VolRiskPremium' and a scalar numeric value.

Data Types: double

LittleTrap — Flag indicating Little Heston Trap formulation
true (default) | logical with values true or false

Flag indicating Little Heston Trap formulation by Albrecher *et al*, specified as the comma-separated pair consisting of 'LittleTrap' and a logical:

- true — Use the Albrecher *et al* formulation.
- false — Use the original Heston formation.

Data Types: logical

OutSpec — Define outputs
["price"] (default) | string array with values "price", "delta", "gamma", "vega", "rho", "theta", and "vegalt" | cell array of character vectors with values 'price', 'delta', 'gamma', 'vega', 'rho', 'theta', and 'vegalt'

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and a NOUT- by-1 or a 1-by-NOUT string array or cell array of character vectors with supported values.

Note "vega" is the sensitivity with respect the initial volatility sqrt(V0). In contrast, "vegalt" is the sensitivity with respect to the long-term volatility sqrt(ThetaV).

Example: OutSpec = ["price","delta","gamma","vega","rho","theta","vegalt"]

Data Types: string | cell

AbsTol — Absolute error tolerance for numerical integration
1e-10 (default) | numeric

Absolute error tolerance for numerical integration, specified as the comma-separated pair consisting of 'AbsTol' and a scalar numeric value.

Data Types: double

RelTol — Relative error tolerance for numerical integration
1e-6 (default) | numeric

Relative error tolerance for numerical integration, specified as the comma-separated pair consisting of `'RelTol'` and a scalar numeric value.

Data Types: `double`

`IntegrationRange` — Numerical integration range used to approximate continuous integral over `[0 Inf]`
`[1e-9 Inf]` (default) | vector

Numerical integration range used to approximate the continuous integral over `[0 Inf]`, specified as the comma-separated pair consisting of `'IntegrationRange'` and a 1-by-2 vector representing `[LowerLimit UpperLimit]`.

Data Types: `double`

`Framework` — Framework for computing option prices and sensitivities using numerical integration of models
`"heston1993"` (default) | string with values `"heston1993"` or `"lewis2001"` | character vector with values `'heston1993'` or `'lewis2001'`

Framework for computing option prices and sensitivities using numerical integration of models, specified as the comma-separated pair consisting of `'Framework'` and a scalar string or character vector with the following values:

- `"heston1993"` or `'heston1993'` — Method used in Heston (1993)
- `"lewis2001"` or `'lewis2001'` — Method used in Lewis (2001)

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-1 vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-1 vector, or a `NRows`-by-`NColumns` matrix. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-1 vectors.
- `false` — If `false`, the outputs are `NINST`-by-1 vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-1 vectors.

Data Types: `logical`

Output Arguments

`PriceSens` — Option prices or sensitivities
numeric

Option prices or sensitivities, returned as a `NINST`-by-1, or `NRows`-by-`NColumns`, depending on `ExpandOutput`. The name-value pair argument `OutSpec` determines the types and order of the outputs.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Heston Stochastic Volatility Model

The Heston model is an extension of the Black-Scholes model, where the volatility (square root of variance) is no longer assumed to be constant, and the variance now follows a stochastic (CIR) process. This allows modeling the implied volatility smiles observed in the market.

The stochastic differential equation is:

$$dS_t = (r - q)S_t dt + \sqrt{v_t} S_t dW_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

v_0 is the initial variance of the asset price at $t = 0$ for ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for the variance for ($\kappa > 0$).

σ_v is the volatility of the variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and $W^v{}_t$ for ($-1 \le p \le 1$).

The characteristic function $f_{Heston_j}(\phi)$ for $j = 1$ (asset price measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Heston_j}(\phi) = \exp(C_j + D_j v_0 + i\phi \ln S_t)$$

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1 - g_j e^{d_j\tau}}{1 - g_j}\right)\right]$$

$$D_j = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1 - e^{d_j\tau}}{1 - g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

where

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

The definitions for C_j and D_j under "The Little Heston Trap" by Albrecher et al. (2007) are:

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v 2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1 - \varepsilon_j e^{-d_j\tau}}{1 - \varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1 - e^{-d_j\tau}}{1 - \varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Numerical Integration Method Under Heston (1993) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Heston (1993) framework is based on the following expressions:

$$Call(K) = S_t e^{-q\tau} P_1 - K e^{-r\tau} P_2$$

$$Put(K) = Call(K) + K e^{-r\tau} - S_t e^{-q\tau}$$

$$P_j = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \mathrm{Re}\left[\frac{e^{-i\phi\ln(K)} f_j(\phi)}{i\phi}\right] d\phi$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number (i^2= -1).

ϕ is the characteristic function variable.

$f_j(\phi)$ is the characteristic function for $P_j(j = 1,2)$.

P_1 is the probability of $S_t > K$ under the asset price measure for the model.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

Where $j = 1,2$ so that $f_1(\phi)$ and $f_2(\phi)$ are the characteristic functions for probabilities P_1 and P_2, respectively.

This framework is chosen with the default value "`Heston1993`" for the `Framework` name-value pair argument.

Numerical Integration Method Under Lewis (2001) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Lewis (2001) framework is based on the following expressions:

$$Call(k) = S_t e^{-q\tau} - \frac{\sqrt{K}e^{-\tau t}}{\pi}\int_0^{\infty} \mathrm{Re}\left[K^{-iu} f_2\left(\phi = \left(u - \frac{i}{2}\right)\right)\frac{1}{u^2 + \frac{1}{4}}\right]du$$

$$Put(K) = Call(K) = Ke^{-\tau t} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity (τ = T-t).

Call(*K*) is the call price at strike *K*.

Put(*K*) is the put price at strike *K*.

i is a unit imaginary number (i^2= -1).

ϕ is the characteristic function variable.

u is the characteristic function variable for integration, where $\phi = \left(u - \frac{i}{2}\right)$.

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

This framework is chosen with the value "`Lewis2001`" for the `Framework` name-value pair argument.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optSensByHestonNI` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Heston, S. L. "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options." *The Review of Financial Studies.* Vol 6. No. 2. 1993.

[2] Lewis, A. L. "A Simple Option Formula for General Jump-Diffusion and Other Exponential Levy Processes." Envision Financial Systems and OptionCity.net, 2001.

See Also

`optByHestonFFT` | `optSensByHestonFFT` | `optByHestonNI` | `optByBatesFFT` | `optSensByBatesFFT` | `optByBatesNI` | `optSensByBatesNI` | `optByMertonFFT` | `optSensByMertonFFT` | `optByMertonNI` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optByLocalVolFD

Option price by local volatility model, using finite differences

Syntax

```
[Price,PriceGrid,AssetPrices,Times] = optByLocalVolFD(Rate,AssetPrice,Settle,
ExerciseDates,OptSpec,Strike,ImpliedVolData)
[Price,PriceGrid,AssetPrices,Times] = optByLocalVolFD( ___ ,Name,Value)
```

Description

`[Price,PriceGrid,AssetPrices,Times] = optByLocalVolFD(Rate,AssetPrice,Settle, ExerciseDates,OptSpec,Strike,ImpliedVolData)` compute a Vanilla European or American option price by the local volatility model, using the Crank-Nicolson method.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceGrid,AssetPrices,Times] = optByLocalVolFD( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Price a European Option Using the Local Volatility Model

Define the option variables.

```
AssetPrice = 590;
Strike = 590;
Rate = 0.06;
DividendYield = 0.0262;
Settle = '01-Jan-2018';
ExerciseDates = '01-Jan-2020';
```

Define the implied volatility surface data.

```
Maturity = ["06-Mar-2018" "05-Jun-2018" "12-Sep-2018" "10-Dec-2018" "01-Jan-2019" ...
"02-Jul-2019" "01-Jan-2020" "01-Jan-2021" "01-Jan-2022" "01-Jan-2023"];
Maturity = repmat(Maturity,10,1);
Maturity = Maturity(:);

ExercisePrice = AssetPrice.*[0.85 0.90 0.95 1.00 1.05 1.10 1.15 1.20 1.30 1.40];
ExercisePrice = repmat(ExercisePrice,1,10)';

ImpliedVol = [...
    0.190; 0.168; 0.133; 0.113; 0.102; 0.097; 0.120; 0.142; 0.169; 0.200; ...
```

```
    0.177; 0.155; 0.138; 0.125; 0.109; 0.103; 0.100; 0.114; 0.130; 0.150; ...
    0.172; 0.157; 0.144; 0.133; 0.118; 0.104; 0.100; 0.101; 0.108; 0.124; ...
    0.171; 0.159; 0.149; 0.137; 0.127; 0.113; 0.106; 0.103; 0.100; 0.110; ...
    0.171; 0.159; 0.150; 0.138; 0.128; 0.115; 0.107; 0.103; 0.099; 0.108; ...
    0.169; 0.160; 0.151; 0.142; 0.133; 0.124; 0.119; 0.113; 0.107; 0.102; ...
    0.169; 0.161; 0.153; 0.145; 0.137; 0.130; 0.126; 0.119; 0.115; 0.111; ...
    0.168; 0.161; 0.155; 0.149; 0.143; 0.137; 0.133; 0.128; 0.124; 0.123; ...
    0.168; 0.162; 0.157; 0.152; 0.148; 0.143; 0.139; 0.135; 0.130; 0.128; ...
    0.168; 0.164; 0.159; 0.154; 0.151; 0.147; 0.144; 0.140; 0.136; 0.132];

ImpliedVolData = table(Maturity, ExercisePrice, ImpliedVol);
```

Compute the European call option price.

```
OptSpec = 'Call';
Price = optByLocalVolFD(Rate, AssetPrice, ...
Settle, ExerciseDates, OptSpec, Strike, ImpliedVolData, 'DividendYield',DividendYield)
```

```
Price = 65.1319
```

Input Arguments

`Rate` — Continuously compounded risk-free interest rate

scalar numeric

Continuously compounded risk-free interest rate, specified by a scalar numeric.

Data Types: `double`

`AssetPrice` — Current underlying asset price

scalar numeric

Current underlying asset price, specified as a scalar numeric.

Data Types: `double`

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `optByLocalVolFD` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, there is only one `ExerciseDates` value and this is the option expiry date.
- For an American option, use a 1-by-2 vector of dates. The American option can be exercised on any date between or including the pair of dates. If only one non-`NaN` date is listed, the option can be exercised between `Settle` and the single listed date in `ExerciseDates`.

To support existing code, `optByLocalVolFD` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a character vector or string array with values `'call'` or `'put'`.

Data Types: `char` | `string`

`Strike` — Option strike price value
nonnegative scalar

Option strike price value, specified as a nonnegative scalar.

Data Types: `double`

`ImpliedVolData` — Table of maturity dates, strike or exercise prices, and corresponding implied volatilities
table

Table of maturity dates, strike or exercise prices, and their corresponding implied volatilities,specified as a `NVOL`-by-`3` table.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optByLocalVolFD(Rate,AssetPrice,Settle, ExerciseDates,OptSpec,Strike,ImpliedVolData,'AssetGridSize',1000)`

`Basis` — Day-count basis
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar using one of the supported values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)

- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric.

Note If you enter a value for `DividendYield`, then set `DividendAmounts` and `ExDividendDates = [ ]` or do not enter them. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield = 0`.

Data Types: `double`

`DividendAmounts` — Cash dividend amounts
`[ ]` (default) | vector

Cash dividend amounts, specified as the comma-separated pair consisting of `'DividendAmounts'` and a `NDIV`-by-`1` vector.

For each dividend amount, there must be a corresponding `ExDividendDates` date. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield = 0`.

Note If you enter a value for `DividendYield`, then set `DividendAmounts` and `ExDividendDates = [ ]` or do not enter them.

Data Types: `double`

`ExDividendDates` — Ex-dividend dates
`[ ]` (default) | datetime array | string array | date character vector

Ex-dividend dates, specified as the comma-separated pair consisting of `'ExDividendDates'` and a `NDIV`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optByLocalVolFD` also accepts serial date numbers as inputs, but they are not recommended.

`AssetPriceMax` — Maximum price for price grid boundary
if unspecified, `AssetPriceMax` values are calculated using asset distributions at maturity (default) | positive scalar

Maximum price for price grid boundary, specified as the comma-separated pair consisting of `'AssetPriceMax'` and a positive scalar.

Data Types: `double`

`AssetGridSize` — Size of asset grid for finite difference grid
`400` (default) | positive scalar

Size of the asset grid for a finite difference grid, specified as the comma-separated pair consisting of `'AssetGridSize'` and a positive scalar.

Data Types: `double`

`TimeGridSize` — Size of time grid for finite difference grid
`100` (default) | positive scalar

Size of the time grid for a finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive scalar.

Data Types: `double`

`AmericanOpt` — Option type
`0` (European) (default) | scalar with values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a positive integer scalar flag with one of these values:

- `0` — European
- `1` — American

Data Types: `double`

`InterpMethod` — Method of interpolation for estimating the implied volatility surface from `ImpliedVolData`
`'linear'` (default) | character vector with values `'linear'`, `'makima'`, `'spline'`, or `'tpaps'` | string with values `"linear"`, `"makima"`, `"spline"`, or `"tpaps"`

Method of interpolation for estimating the implied volatility surface from `ImpliedVolData`, specified as the comma-separated pair consisting of `'InterpMethod'` and a character vector or string array with one of the following values:

- `'linear'` — Linear interpolation
- `'makima'` — Modified Akima cubic Hermite interpolation
- `'spline'` — Cubic spline interpolation
- `'tpaps'` — Thin-plate smoothing spline interpolation

Note The `'tpaps'` method uses the thin-plate smoothing spline functionality from Curve Fitting Toolbox.

The `'makima'` and `'spline'` methods work only for gridded data. For scattered data, use the `'linear'` or `'tpaps'` methods.

For more information on gridded or scattered data and details on interpolation methods, see “Gridded and Scattered Sample Data” and “Interpolating Gridded Data”.

Data Types: `char` | `string`

Output Arguments

`Price` — Option price
scalar numeric

Option price, returned as a scalar numeric.

`PriceGrid` — Grid containing prices calculated by finite difference method
grid

Grid containing prices calculated by the finite difference method, returned as a grid that is two-dimensional with size `AssetGridSize` × `TimeGridSize`. The number of columns does not have to be equal to the `TimeGridSize`, because `ExerciseDates` and `ExDividendDates` are added to the time grid. `PriceGrid(:, :, end)` contains the price for $t = 0$.

`AssetPrices` — Prices of asset
vector

Prices of the asset corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Times` — Times
vector

Times corresponding to second dimension of the `PriceGrid`, returned as a vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Local Volatility Model

A local volatility model treats volatility as a function both of the current asset level and of time.

The local volatility can be estimated by using the Dupire formula [2]:

$$\sigma_{loc}^2(K,\tau) = \frac{\sigma_{imp}^2 + 2\tau\sigma_{imp}\frac{\partial\sigma_{imp}}{\partial\tau} + 2(\tau - d)K\tau\sigma_{imp}\frac{\partial\sigma_{imp}}{\partial K}}{\left(1 + Kd_1\sqrt{\tau}\frac{\partial\sigma_{imp}}{\partial K}\right)^2 + K^2\tau\sigma_{imp}\left(\frac{\partial^2\sigma_{imp}}{\partial K^2} - d_1\sqrt{\tau}\left(\frac{\partial\sigma_{imp}}{\partial K}\right)^2\right)}$$

$$d_1 = \frac{\ln(S_0/K) + \left((\tau - d) + \sigma_{imp}^2/2\right)\tau}{\sigma_{imp}\sqrt{\tau}}$$

Version History

Introduced in R2018b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optByLocalVolFD` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Andersen, L. B., and R. Brotherton-Ratcliffe. "The Equity Option Volatility Smile: An Implicit Finite-Difference Approach." *Journal of Computational Finance.* Vol. 1, Number 2, 1997, pp. 5–37.

[2] Dupire, B. "Pricing with a Smile." *Risk*. Vol. 7, Number 1, 1994, pp. 18–20.

See Also

`optstockbyfd` | `optstocksensbyfd` | `optSensByLocalVolFD` | `optByHestonFD` | `optSensByHestonFD` | `optByBatesFD` | `optSensByBatesFD` | `optByMertonFD` | `optSensByMertonFD` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19

optSensByLocalVolFD

Option price and sensitivities by local volatility model, using finite differences

Syntax

```
[PriceSens,PriceGrid,AssetPrices,Times] = optSensByLocalVolFD(Rate,
AssetPrice,Settle,ExerciseDates,OptSpec,Strike,ImpliedVolData)
[PriceSens,PriceGrid,AssetPrices,Times] = optSensByLocalVolFD( ___ ,Name,Value)
```

Description

`[PriceSens,PriceGrid,AssetPrices,Times] = optSensByLocalVolFD(Rate, AssetPrice,Settle,ExerciseDates,OptSpec,Strike,ImpliedVolData)` compute option price and sensitivities by the local volatility model, using the Crank-Nicolson method.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,PriceGrid,AssetPrices,Times] = optSensByLocalVolFD( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute European Option Price and Sensitivities Using the Local Volatility Model

Define the option variables.

```
AssetPrice = 590;
Strike = 590;
Rate = 0.06;
DividendYield = 0.0262;
Settle = datetime(2018,1,1);
ExerciseDates = datetime(2020,1,1);
```

Define the implied volatility surface data.

```
Maturity = [datetime(2018,3,6) datetime(2018,6,5) datetime(2018,9,12) datetime(2018,12,10) dateti
Maturity = repmat(Maturity,10,1);
Maturity = Maturity(:);

ExercisePrice = AssetPrice.*[0.85 0.90 0.95 1.00 1.05 1.10 1.15 1.20 1.30 1.40];
ExercisePrice = repmat(ExercisePrice,1,10)';

ImpliedVol = [...
    0.190; 0.168; 0.133; 0.113; 0.102; 0.097; 0.120; 0.142; 0.169; 0.200; ...
    0.177; 0.155; 0.138; 0.125; 0.109; 0.103; 0.100; 0.114; 0.130; 0.150; ...
```

```
    0.172; 0.157; 0.144; 0.133; 0.118; 0.104; 0.100; 0.101; 0.108; 0.124; ...
    0.171; 0.159; 0.149; 0.137; 0.127; 0.113; 0.106; 0.103; 0.100; 0.110; ...
    0.171; 0.159; 0.150; 0.138; 0.128; 0.115; 0.107; 0.103; 0.099; 0.108; ...
    0.169; 0.160; 0.151; 0.142; 0.133; 0.124; 0.119; 0.113; 0.107; 0.102; ...
    0.169; 0.161; 0.153; 0.145; 0.137; 0.130; 0.126; 0.119; 0.115; 0.111; ...
    0.168; 0.161; 0.155; 0.149; 0.143; 0.137; 0.133; 0.128; 0.124; 0.123; ...
    0.168; 0.162; 0.157; 0.152; 0.148; 0.143; 0.139; 0.135; 0.130; 0.128; ...
    0.168; 0.164; 0.159; 0.154; 0.151; 0.147; 0.144; 0.140; 0.136; 0.132];

ImpliedVolData = table(Maturity, ExercisePrice, ImpliedVol);
```

Compute the European call option price and sensitivities.

```
OptSpec = 'Call';
[Delta,Gamma,Lambda,Theta,Price]  = optSensByLocalVolFD(Rate, AssetPrice, ...
Settle, ExerciseDates, OptSpec, Strike, ImpliedVolData, 'DividendYield',DividendYield, ...
'OutSpec',["Delta" "Gamma" "Lambda" "Theta" "Price"])

Delta = 0.5519

Gamma = 0.0091

Lambda = 4.9994

Theta = -20.9529

Price = 65.1319
```

Input Arguments

`Rate` — Continuously compounded risk-free interest rate

scalar numeric

Continuously compounded risk-free interest rate, specified by a scalar numeric.

Data Types: `double`

`AssetPrice` — Current underlying asset price

scalar numeric

Current underlying asset price, specified as a scalar numeric.

Data Types: `double`

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `optSensByLocalVolFD` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, there is only one `ExerciseDates` value and this is the option expiry date.
- For an American option, use a 1-by-2 vector dates. The American option can be exercised on any date between or including the pair of dates. If only one non-`NaN` date is listed, the option can be exercised between `Settle` and the single listed date in `ExerciseDates`.

To support existing code, `optSensByLocalVolFD` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with value of `'call'` or `'put'` | string array with value of `"call"` or `"put"`

Definition of the option, specified as a character vector or string array with a value of `'call'` or `'put'`.

Data Types: `char` | `string`

`Strike` — Option strike price value
nonnegative scalar

Option strike price value, specified as a nonnegative scalar.

Data Types: `double`

`ImpliedVolData` — Table of maturity dates, strike or exercise prices, and corresponding implied volatilities
table

A table of maturity dates, strike or exercise prices, and their corresponding implied volatilities,specified as a `NVOL`-by-3 table.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = Price = optByLocalVolFD(Rate,AssetPrice,Settle, ExerciseDates,OptSpec,Strike,ImpliedVolData,'AssetGridSize',1000,'OutSpec', {'delta','gamma','vega','lambda','rho','theta','price'})`

`Basis` — Day-count basis
`0` (default) | numeric values: `0`,1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar using one of these supported values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | scalar numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric.

Note If you enter a value for `DividendYield`, then set `DividendAmounts` and `ExDividendDates = [ ]` or do not enter them. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield = 0`.

Data Types: `double`

`DividendAmounts` — Cash dividend amounts
`[ ]` (default) | vector

Cash dividend amounts, specified as the comma-separated pair consisting of `'DividendAmounts'` and a `NDIV`-by-`1` vector.

For each dividend amount, there must be a corresponding `ExDividendDates` date. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield = 0`.

Note If you enter a value for `DividendYield`, then set `DividendAmounts` and `ExDividendDates = [ ]` or do not enter them.

Data Types: `double`

`ExDividendDates` — Ex-dividend dates
`[ ]` (default) | datetime array | string array | date character vector

Ex-dividend dates, specified as the comma-separated pair consisting of `'ExDividendDates'` and a `NDIV`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optSensByLocalVolFD` also accepts serial date numbers as inputs, but they are not recommended.

AssetPriceMax — Maximum price for price grid boundary
if unspecified, AssetPriceMax values are calculated using asset distributions at maturity (default) | positive scalar

Maximum price for price grid boundary, specified as the comma-separated pair consisting of 'AssetPriceMax' and a positive scalar.

Data Types: double

AssetGridSize — Size of asset grid for finite difference grid
400 (default) | positive scalar

Size of the asset grid for finite difference grid, specified as the comma-separated pair consisting of 'AssetGridSize' and a positive scalar.

Data Types: double

TimeGridSize — Size of time grid for finite difference grid
100 (default) | positive scalar

Size of the time grid for finite difference grid, specified as the comma-separated pair consisting of 'TimeGridSize' and a positive scalar.

Data Types: double

AmericanOpt — Option type
0 (European) (default) | scalar with values [0,1]

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and a positive integer scalar flag with one of these values:

- 0 — European
- 1 — American

Data Types: double

InterpMethod — Method of interpolation for estimating the implied volatility surface from ImpliedVolData
'linear' (default) | character vector with values 'linear', 'makima', 'spline', or 'tpaps' | string with values "linear", "makima", "spline", or "tpaps"

Method of interpolation for estimating the implied volatility surface from ImpliedVolData, specified as the comma-separated pair consisting of 'InterpMethod' and a character vector or string with one of the following values:

- 'linear' — Linear interpolation
- 'makima' — Modified Akima cubic Hermite interpolation
- 'spline' — Cubic spline interpolation
- 'tpaps' — Thin-plate smoothing spline interpolation

Note The 'tpaps' method uses the thin-plate smoothing spline functionality from Curve Fitting Toolbox.

The 'makima' and 'spline' methods work only for gridded data. For scattered data, use the 'linear' or 'tpaps' methods.

For more information on gridded or scattered data and details on interpolation methods, see “Gridded and Scattered Sample Data” and “Interpolating Gridded Data”.

Data Types: `char` | `string`

`OutSpec` — Define outputs
`{'price'}` (default) | cell array of character vectors with values `'price'`, `'delta'`, `'gamma'`, `'vega'`, `'lambda'`, `'rho'`, `'theta'` | string array with values `"price"`, `"delta"`, `"gamma"`, `"vega"`, `"rho"`, `"theta"`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and an `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'price'`, `'delta'`, `'gamma'`, `'vega'`, `'lambda'`, `'rho'`, and `'theta'`.

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `cell` | `string`

Output Arguments

`PriceSens` — Option price and sensitivities
scalar numeric

Option price and sensitivities, returned as a scalar numeric. `OutSpec` determines the types and order of the output.

`PriceGrid` — Grid containing prices calculated by finite difference method
grid

Grid containing prices calculated by the finite difference method, returned as a grid that is two-dimensional with size `AssetGridSize` × `TimeGridSize`. The number of columns does not have to be equal to the `TimeGridSize`, because `ExerciseDates` and `ExDividendDates` are added to the time grid. `PriceGrid(:, :, end)` contains the price for $t = 0$.

`AssetPrices` — Prices of asset
vector

Prices of the asset corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Times` — Times
vector

Times corresponding to second dimension of the `PriceGrid`, returned as a vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Local Volatility Model

A local volatility model treats volatility as a function both of the current asset level and of time.

The local volatility can be estimated by using the Dupire formula [2]:

$$\sigma_{loc}^2(K,\tau) = \frac{\sigma_{imp}^2 + 2\tau\sigma_{imp}\frac{\partial\sigma_{imp}}{\partial\tau} + 2(\tau - d)K\tau\sigma_{imp}\frac{\partial\sigma_{imp}}{\partial K}}{\left(1 + Kd_1\sqrt{\tau}\frac{\partial\sigma_{imp}}{\partial K}\right)^2 + K^2\tau\sigma_{imp}\left(\frac{\partial^2\sigma_{imp}}{\partial K^2} - d_1\sqrt{\tau}\left(\frac{\partial\sigma_{imp}}{\partial K}\right)^2\right)}$$

$$d_1 = \frac{\ln(S_0/K) + \left((\tau - d) + \sigma_{imp}^2/2\right)\tau}{\sigma_{imp}\sqrt{\tau}}$$

Version History

Introduced in R2018b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optSensByLocalVolFD` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Andersen, L. B., and R. Brotherton-Ratcliffe. "The Equity Option Volatility Smile: An Implicit Finite-Difference Approach." *Journal of Computational Finance.* Vol. 1, Number 2, 1997, pp. 5–37.

[2] Dupire, B. "Pricing with a Smile." *Risk*. Vol. 7, Number 1, 1994, pp. 18–20.

See Also

`optstockbyfd` | `optstocksensbyfd` | `optByLocalVolFD` | `optByHestonFD` | `optSensByHestonFD` | `optByBatesFD` | `optSensByBatesFD` | `optByMertonFD` | `optSensByMertonFD` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optByMertonFD

Option price by Merton76 model using finite differences

Syntax

```
[Price,PriceGrid,AssetPrices,Times] = optByMertonFD(Rate,AssetPrice,Settle,
ExerciseDates,OptSpec,Strike,Sigma,MeanJ,JumpVol,JumpFreq)
[Price,PriceGrid,AssetPrices,Times] = optByMertonFD( ___ ,Name,Value)
```

Description

`[Price,PriceGrid,AssetPrices,Times] = optByMertonFD(Rate,AssetPrice,Settle, ExerciseDates,OptSpec,Strike,Sigma,MeanJ,JumpVol,JumpFreq)` computes a vanilla European or American option price by the Merton76 model, using the Crank-Nicolson Adams-Bashforth (CNAB) IMEX method.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceGrid,AssetPrices,Times] = optByMertonFD( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute Price for an American Option Using the Merton Model

Define the option variables and Merton model parameters.

```
AssetPrice = 90;
Strike = 100;
Rate = 0.06;
DividendYield = 0.1;
Settle = datetime(2018,1,1);
ExerciseDates = datetime(2018,4,2);

Sigma = 0.40;
MeanJ = -0.10;
JumpVol = 0.01;
JumpFreq = 1.00;
```

Compute the American call option price using the finite differences method.

```
OptSpec = 'Call';

Price = optSensByMertonFD(Rate, AssetPrice, Settle, ExerciseDates, OptSpec, Strike,...
Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'AmericanOpt', 1)
```

```
Price = 3.4551
```

Input Arguments

`Rate` — Continuously compounded risk-free interest rate

scalar decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

scalar numeric

Current underlying asset price, specified as a scalar numeric.

Data Types: `double`

`Settle` — Option settlement date

datetime scalar | string scalar | date character vector

Option settlement date, specified as a scalar datetime, string, or data character vector.

To support existing code, `optByMertonFD` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a scalar date. For a European option, `ExerciseDates` contains only one value: the option expiry date.
- For an American option, use a 1-by-2 vector of dates to specify the exercise date boundaries. An American option can be exercised on any date between or including the pair of dates. If only one non-`NaN` date is listed, then the option can be exercised between `Settle` date and the single listed value in `ExerciseDates`.

To support existing code, `optByMertonFD` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option

character vector with value of `'call'` or `'put'` | string array with value of `"call"` or `"put"`

Definition of the option, specified as a scalar using a character vector or string array with a value of `'call'` or `'put'`.

Data Types: `cell` | `string`

`Strike` — Option strike price value

scalar numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

Sigma — Volatility of underlying asset
scalar numeric

Volatility of the underling asset, specified as a scalar numeric.

Data Types: double

MeanJ — Mean of the random percentage jump size
scalar decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal value where log(1+*J*) is normally distributed with the mean (log(1+MeanJ)-0.5*JumpVol^2) and the standard deviation JumpVol.

Data Types: double

JumpVol — Standard deviation of log(1+*J*)
scalar decimal

Standard deviation of log(1+*J*) where J is the random percentage jump size, specified as a scalar decimal.

Data Types: double

JumpFreq — Annual frequency of Poisson jump process
numeric

Annual frequency of the Poisson jump process, specified as a scalar numeric.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceGrid] = optByMertonFD(Rate,AssetPrice,Settle,ExerciseDates,OptSpec,Strike,MeanJ,JumpVol,JumpFreq,'Basis',7)

Basis — Day-count basis of instrument
0 (default) | numeric values: 0,1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13

Day-count basis of the instrument, specified as the comma-separated pair consisting of 'Basis' and a scalar using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)

- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | scalar numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric.

Note If you enter a value for `DividendYield`, then set `DividendAmounts` and `ExDividendDates = [ ]` or do not enter them. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield = 0`.

Data Types: `double`

`DividendAmounts` — Cash dividend amounts
`[ ]` (default) | vector

Cash dividend amounts, specified as the comma-separated pair consisting of `'DividendAmounts'` and an `NDIV`-by-`1` vector.

Note Each dividend amount must have a corresponding ex-dividend date. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield = 0`.

Data Types: `double`

`ExDividendDates` — Ex-dividend dates
`[ ]` (default) | datetime array | string array | date character vector

Ex-dividend dates, specified as the comma-separated pair consisting of `'ExDividendDates'` and an `NDIV`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optByMertonFD` also accepts serial date numbers as inputs, but they are not recommended.

`AssetPriceMax` — Maximum price for price grid boundary
if unspecified, value is calculated based on asset price distribution at maturity (default) | positive scalar numeric

Maximum price for the price grid boundary, specified as the comma-separated pair consisting of `'AssetPriceMax'` and a positive scalar numeric.

Data Types: `double`

`AssetGridSize` — Size of asset grid for finite difference grid
`400` (default) | scalar numeric

Size of the asset grid for the finite difference grid, specified as the comma-separated pair consisting of `'AssetGridSize'` and a scalar numeric.

Data Types: `double`

`TimeGridSize` — Number of nodes of time grid for finite difference grid
`100` (default) | positive numeric scalar

Number of nodes of the time grid for the finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive numeric scalar.

Data Types: `double`

`AmericanOpt` — Option type
`0` (European) (default) | scalar with value of `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar flag with one of these values:

- `0` — European
- `1` — American

Data Types: `double`

Output Arguments

`Price` — Option price
scalar numeric

Option price, returned as a scalar numeric.

`PriceGrid` — Grid containing prices calculated by the finite difference method
grid

Grid containing prices calculated by the finite difference method, returned as a two-dimensional grid with size `AssetGridSize` × `TimeGridSize`. The number of columns is not necessarily equal to the `TimeGridSize` because exercise and ex-dividend dates are added to the time grid. `PriceGrid(:, :, end)` contains the price for $t = 0$.

`AssetPrices` — Prices of the asset
vector

Prices of the asset corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Times` — Times
vector

Times corresponding to the second dimension of `PriceGrid`, returned as a vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Merton Jump Diffusion Model

The Merton jump diffusion model [2] extends the Black-Scholes model by using the Poisson process to include jump diffusion parameters in the modeling of sudden asset price movements (both up and down).

The stochastic differential equation is

$$dS_t = (r - q - \lambda_p \mu_j) S_t dt + \sigma S_t dW_t + J S_t dP_t$$
$$\text{prob}(dP_t = 1) = \lambda_p dt$$

where:

r is the continuous risk-free rate.

q is the continuous dividend yield.

W_t is the Weiner process.

J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1 + J)\delta\sqrt{2\pi}} \exp\left\{ \frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2} \right\}$$

where:

μ_J is the mean of J for (μ_J > -1).

δ is the standard deviation of $\ln(1+J)$ for ($\delta \geq 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \geq 0$).

σ is the volatility of the asset price for ($\sigma > 0$).

Version History

Introduced in R2019a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optByMertonFD` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cont, R., and E. Voltchkova. “A Finite Difference Scheme for Option Pricing in Jump Diffusion and Exponential Lévy Models.” *SIAM Journal on Numerical Analysis.* Vol. 43, Number 4, 2005, pp. 1596-1626.

[2] Merton, R. "Option Pricing When Underlying Stock Returns Are Discontinuous." *The Journal of Financial Economics.* Vol 3. 1976, pp. 125-144.

See Also

`optByLocalVolFD` | `optSensByLocalVolFD` | `optByHestonFD` | `optSensByHestonFD` | `optSensByMertonFD` | `optSensByBatesFD` | `optByBatesFD` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optSensByMertonFD

Option price and sensitivities by Merton76 model using finite differences

Syntax

```
[PriceSens,PriceGrid,AssetPrices,Times] = optSensByMertonFD(Rate,AssetPrice,
Settle,ExerciseDates,OptSpec,Strike,Sigma,MeanJ,JumpVol,JumpFreq)
[PriceSens,PriceGrid,AssetPrices,Times] = optSensByMertonFD( ___ ,Name,Value)
```

Description

`[PriceSens,PriceGrid,AssetPrices,Times] = optSensByMertonFD(Rate,AssetPrice, Settle,ExerciseDates,OptSpec,Strike,Sigma,MeanJ,JumpVol,JumpFreq)` computes a vanilla European or American option price and sensitivities by the Merton76 model, using the Crank-Nicolson Adams-Bashforth (CNAB) IMEX method.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,PriceGrid,AssetPrices,Times] = optSensByMertonFD( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute Price and Sensitivities for an American Option Using the Merton Model

Define the option variables and Merton model parameters.

```
AssetPrice = 90;
Strike = 100;
Rate = 0.06;
DividendYield = 0.1;
Settle = datetime(2018,1,1);
ExerciseDates = datetime(2018,4,2);

Sigma = 0.40;
MeanJ = -0.10;
JumpVol = 0.01;
JumpFreq = 1.00;
```

Compute the American call option price and sensitivities using the finite differences method.

```
OptSpec = 'Call';

[Price, Delta, Gamma, Rho, Theta, Vega] = optSensByMertonFD(Rate, AssetPrice, Settle, ExerciseDa
Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'AmericanOpt', 1,...
'OutSpec', ["Price" "Delta" "Gamma" "Rho" "Theta" "Vega"])
```

```
Price = 3.4551

Delta = 0.3211

Gamma = 0.0195

Rho = 5.6610

Theta = -11.9877

Vega = 15.5156
```

Input Arguments

`Rate` — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

numeric

Current underlying asset price, specified as a scalar numeric.

Data Types: `double`

`Settle` — Option settlement date

datetime scalar | string scalar | date character vector

Option settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `optSensByMertonFD` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a scalar date. For a European option, `ExerciseDates` contains only one value: the option expiry date.
- For an American option, use a 1-by-2 vector of dates to specify the exercise date boundaries. An American option can be exercised on any date between or including the pair of dates. If only one non-`NaN` date is listed, then the option can be exercised between `Settle` date and the single listed value in `ExerciseDates`.

  To support existing code, `optSensByMertonFD` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option

character vector with value of `'call'` or `'put'` | string array with value of `"call"` or `"put"`

Definition of the option, specified as a scalar using a character vector or string array with a value of `'call'` or `'put'`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
scalar numeric

Option strike price value, specified as a scalar numeric.

Data Types: `double`

`Sigma` — Volatility of underlying asset
scalar numeric

Volatility of the underling asset, specified as a scalar numeric.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
scalar decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal value where `log`(1+*J*) is normally distributed with the mean (`log`(1+`MeanJ`)-0.5*`JumpVol`^2) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log`(1+*J*)
scalar decimal

Standard deviation of `log`(1+*J*) where `J` is the random percentage jump size, specified as a scalar decimal.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
scalar numeric

Annual frequency of the Poisson jump process, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceGrid] = optByMertonFD(Rate,AssetPrice,Settle,ExerciseDates,OptSpec,Strike,MeanJ,JumpVol,JumpFreq,'Basis',7,'OutSpec','delta')`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | scalar numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric.

Note If you enter a value for `DividendYield`, then set `DividendAmounts` and `ExDividendDates = [ ]` or do not enter them. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield = 0`.

Data Types: `double`

`DividendAmounts` — Cash dividend amounts
`[ ]` (default) | vector

Cash dividend amounts, specified as the comma-separated pair consisting of `'DividendAmounts'` and an `NDIV`-by-`1` vector.

Note Each dividend amount must have a corresponding ex-dividend date. If you enter values for `DividendAmounts` and `ExDividendDates`, then set `DividendYield = 0`.

Data Types: `double`

`ExDividendDates` — Ex-dividend dates
`[]` (default) | datetime array | string array | date character vector

Ex-dividend dates, specified as the comma-separated pair consisting of `'ExDividendDates'` and an `NDIV`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optSensByMertonFD` also accepts serial date numbers as inputs, but they are not recommended.

`AssetPriceMax` — Maximum price for price grid boundary
if unspecified, value is calculated based on asset price distribution at maturity (default) | positive scalar numeric

Maximum price for the price grid boundary, specified as the comma-separated pair consisting of `'AssetPriceMax'` and a positive scalar numeric.

Data Types: `double`

`AssetGridSize` — Size of asset grid for the finite difference grid
`400` (default) | scalar numeric

Size of the asset grid for finite difference grid, specified as the comma-separated pair consisting of `'AssetGridSize'` and a scalar numeric.

Data Types: `double`

`TimeGridSize` — Number of nodes of time grid for the finite difference grid
`100` (default) | positive numeric scalar

Number of nodes of the time grid for finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive numeric scalar.

Data Types: `double`

`AmericanOpt` — Option type
`0` (European) (default) | scalar with value of `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar flag with one of these values:

- `0` — European
- `1` — American

Data Types: `double`

`OutSpec` — Define outputs
`['price']` (default) | cell array of character vectors with values `'price'`, `'delta'`, `'gamma'`, `'vega'`, `'rho'`, and `'theta'` | string array with values `"price"`, `"delta"`, `"gamma"`, `"vega"`, `"rho"`, and `"theta"`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` string array or cell array of character vectors with supported values.

Example: `OutSpec = ['price','delta','gamma','vega','rho','theta']`

Data Types: `string` | `cell`

Output Arguments

`PriceSens` — Option price or sensitivities
numeric

Option price or sensitivities, returned as a numeric. The name-value pair argument `OutSpec` determines the types and order of the outputs.

`PriceGrid` — Grid containing prices calculated by the finite difference method
grid

Grid containing prices calculated by the finite difference method, returned as a two-dimensional grid with size `AssetGridSize` × `TimeGridSize`. The number of columns is not necessarily equal to the `TimeGridSize` because exercise and ex-dividend dates are added to the time grid. `PriceGrid(:, :, end)` contains the price for $t = 0$.

`AssetPrices` — Prices of the asset
vector

Prices of the asset corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Times` — Times
vector

Times corresponding to the second dimension of `PriceGrid`, returned as a vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Merton Jump Diffusion Model

The Merton jump diffusion model [2] extends the Black-Scholes model by using the Poisson process to include jump diffusion parameters in the modeling of sudden asset price movements (both up and down).

The stochastic differential equation is

$$dS_t = (r - q - \lambda_p \mu_j) S_t dt + \sigma S_t dW_t + J S_t dP_t$$
$$\text{prob}(dP_t = 1) = \lambda_p dt$$

where:

r is the continuous risk-free rate.

q is the continuous dividend yield.

W_t is the Weiner process.

J is the random percentage jump size conditional on the jump occurring, where `ln`$(1+J)$ is normally distributed with mean $\ln(1+\mu_J)-\frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1+J)\delta\sqrt{2\pi}}\exp\left\{\frac{-\left[\ln(1+J)-\left(\ln(1+\mu_J)-\frac{\delta^2}{2}\right]\right]^2}{2\delta^2}\right\}$$

where:

μ_J is the mean of J for $(\mu_J > -1)$.

δ is the standard deviation of `ln`$(1+J)$ for $(\delta \geq 0)$.

λ_p is the annual frequency (intensity) of Poisson process $P_t$for $(\lambda_p \geq 0)$.

σ is the volatility of the asset price for $(\sigma > 0)$.

Version History

Introduced in R2019a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optSensByMertonFD` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cont, R., and E. Voltchkova. “A Finite Difference Scheme for Option Pricing in Jump Diffusion and Exponential Lévy Models.” *SIAM Journal on Numerical Analysis*. Vol. 43, Number 4, 2005, pp. 1596-1626.

[2] Merton, R. "Option Pricing When Underlying Stock Returns Are Discontinuous." *The Journal of Financial Economics.* Vol 3. 1976, pp. 125-144.

See Also

`optByLocalVolFD` | `optSensByLocalVolFD` | `optByHestonFD` | `optSensByHestonFD` | `optByBatesFD` | `optSensByBatesFD` | `optByMertonFD` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optByMertonFFT

Option price by Merton76 model using FFT and FRFT

Syntax

```
[Price,StrikeOut] = optByMertonFFT(Rate,AssetPrice,Settle,Maturity,OptSpec,
Strike,Sigma,MeanJ,JumpVol,JumpFreq)
[Price,StrikeOut] = optByMertonFFT( ___ ,Name,Value)
```

Description

`[Price,StrikeOut] = optByMertonFFT(Rate,AssetPrice,Settle,Maturity,OptSpec, Strike,Sigma,MeanJ,JumpVol,JumpFreq)` computes vanilla European option price by Merton76 model, using Carr-Madan FFT and Chourdakis FRFT methods.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,StrikeOut] = optByMertonFFT( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Price Surface Using the Merton76 Model

Use `optByMertonFFT` to calibrate the FFT strike grid, compute option prices, and plot an option price surface.

Define Option Variables and Merton76 Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

Sigma = 0.16;
MeanJ = 0.02;
JumpVol = 0.08;
JumpFreq = 2;
```

Compute the Option Prices for the Entire FFT (or FRFT) Strike Grid, Without Specifying "Strike"

Compute option prices and also output the corresponding strikes. If the `Strike` input is empty (`[]`), option prices will be computed on the entire FFT (or FRFT) strike grid. The FFT (or FRFT) strike grid is determined as `exp(log-strike grid)`, where each column of the log-strike grid has `NumFFT` points with `LogStrikeStep` spacing that are roughly centered around each element of

log(AssetPrice). The default value for NumFFT is 2^12. In addition to the prices in the first output, the optional last output contains the corresponding strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = []; % Strike is not specified (will use the entire FFT strike grid)

% Compute option prices for the entire FFT strike grid
[Call, Kout] = optByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield);

% Show the lowest and highest strike values on the FFT strike grid
format
MinStrike = Kout(1) % Lowest possible strike in the current FFT strike grid

MinStrike = 2.9205e-135

MaxStrike = Kout(end) % Highest possible strike in the current FFT strike grid

MaxStrike = 1.8798e+138


% Show a subset of the strikes and corresponding option prices
Range = (2046:2052);
[Kout(Range) Call(Range)]

ans = 7×2

   50.4929   29.4645
   58.8640   21.2601
   68.6231   12.2218
   80.0000    4.5600
   93.2631    0.9579
  108.7251    0.1236
  126.7505    0.0113
```

Change the Number of FFT (or FRFT) Points and Compare with optByMertonNI

Try a different number of FFT(or FRFT) points, and compare the results with direct numerical integration. Unlike optByMertonFFT, which uses FFT (or FRFT) techniques for fast computation across the whole range of strikes, the optByMertonNI function uses direct numerical integration and it is typically slower, especially for multiple strikes. However, the values computed by optByMertonNI can serve as a benchmark for adjusting the settings for optByMertonFFT.

```
% Try a smaller number of FFT (or FRFT) points
% (e.g. for faster performance or smaller memory footprint)
NumFFT = 2^10; % Smaller than the default value of 2^12
Strike = []; % Strike is not specified (will use the entire FFT strike grid)
[Call, Kout] = optByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'NumFFT', NumFFT);

% Compare with numerical integration method
Range = (510:516);
Strike = Kout(Range);
CallFFT = Call(Range);
CallNI = optByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield);
```

```
Error = abs(CallFFT-CallNI);
table(Strike, CallFFT, CallNI, Error)
```

```
ans=7×4 table
    Strike     CallFFT       CallNI         Error
    ______    _________    ___________    _________

    12.696       66.328         66.696      0.36786
    23.449       55.922         56.103      0.18071
    43.312       36.481         36.536     0.055233
        80       4.7387           4.56      0.17867
    147.76     0.046602      0.0008089     0.045793
    272.93    0.0092842    -7.0709e-08    0.0092842
    504.11    0.0024041    -2.4515e-07    0.0024044
```

Make Further Adjustments to FFT (or FRFT)

If the values in the output `CallFFT` are significantly different from those in `CallNI`, try making adjustments to `optByMertonFFT` settings, such as `CharacteristicFcnStep`, `LogStrikeStep`, `NumFFT`, `DampingFactor`, and so on. Note that if (`LogStrikeStep` * `CharacteristicFcnStep`) is 2*pi/ `NumFFT`, FFT is used. Otherwise, FRFT is used.

```
Strike = []; % Strike is not specified (will use the entire FFT or FRFT strike grid)
[Call, Kout] = optByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001);

% Compare with numerical integration method
Strike = Kout(Range);
CallFFT = Call(Range);
CallNI = optByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield);
Error = abs(CallFFT-CallNI);
table(Strike, CallFFT, CallNI, Error)
```

```
ans=7×4 table
    Strike    CallFFT    CallNI      Error
    ______    _______    ______    __________

    79.76      4.674      4.674    4.9664e-10
    79.84     4.6358     4.6358    4.9651e-10
    79.92     4.5978     4.5978    4.9642e-10
       80       4.56       4.56    4.9641e-10
    80.08     4.5224     4.5224    4.9642e-10
    80.16      4.485      4.485     4.965e-10
    80.24     4.4478     4.4478     4.966e-10
```

```
% Save the final FFT (or FRFT) strike grid for future reference. For
% example, it provides information about the range of Strike inputs for
% which the FFT (or FRFT) operation is valid.
FFTStrikeGrid = Kout;
MinStrike = FFTStrikeGrid(1) % Strike cannot be less than MinStrike
```

```
MinStrike = 47.9437
```

```
MaxStrike = FFTStrikeGrid(end) % Strike cannot be greater than MaxStrike
```

```
MaxStrike = 133.3566
```

Compute Option Price for a Single Strike

Once the desired FFT (or FRFT) settings are determined, use the `Strike` input to specify the strikes rather than providing an empty array. If the specified strikes do not match a value on the FFT (or FRFT) strike grid, the outputs are interpolated on the specified strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;

Call = optByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001)
```

```
Call = 4.5600
```

Compute the Option Prices for a Vector of Strikes

Use the `Strike` input to specify the strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Call = optByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001)
```

```
Call = 5×1

    6.7411
    5.5762
    4.5600
    3.6891
    2.9551
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Call = optByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001) % Five values in vector output
```

```
Call = 5×1

    8.5589
    8.9439
    9.2316
```

```
    9.4653
    9.6565
```

Expand the Outputs for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the outputs into `NStrikes`-by-`NMaturities` matrices. In this case, they are square matrices.

```
[Call, Kout] = optByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'ExpandOutput', true) % (5 x 5) matrix output
```

```
Call = 5×5

    8.5589    9.9675   11.1343   12.1492   13.0464
    7.4844    8.9439   10.1481   11.1939   12.1181
    6.5125    8.0023    9.2316   10.2999   11.2449
    5.6401    7.1402    8.3827    9.4653   10.4249
    4.8630    6.3545    7.5990    8.6881    9.6565


Kout = 5×5

    76    76    76    76    76
    78    78    78    78    78
    80    80    80    80    80
    82    82    82    82    82
    84    84    84    84    84
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities`. That is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Call = optByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'ExpandOutput', true) % (5 x 6) matrix output
```

```
Call = 5×6

    6.7411    8.5589    9.9675   11.1343   12.1492   13.0464
    5.5762    7.4844    8.9439   10.1481   11.1939   12.1181
    4.5600    6.5125    8.0023    9.2316   10.2999   11.2449
    3.6891    5.6401    7.1402    8.3827    9.4653   10.4249
    2.9551    4.8630    6.3545    7.5990    8.6881    9.6565
```

Compute the Option Prices for a Vector of Strikes and a Vector of Asset Prices

When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

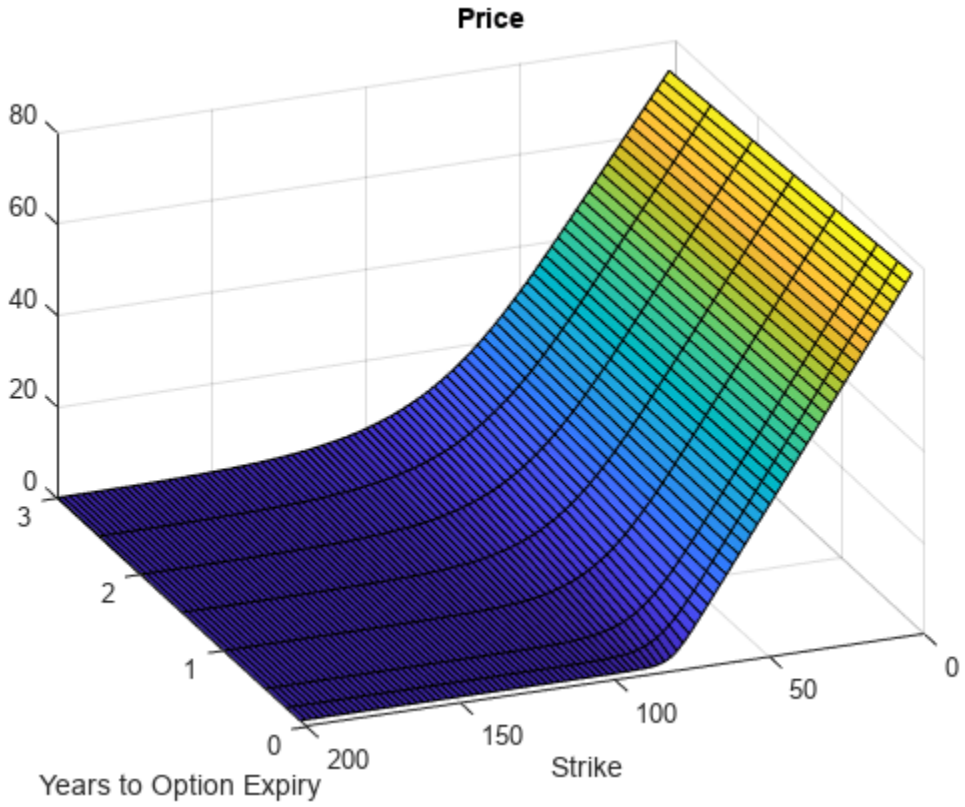
```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Call = optByMertonFFT(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'ExpandOutput', true) % (5 x 4) matrix output
```

```
Call = 5×4

    3.4187    5.6579    8.5589   12.0417
    2.8538    4.8401    7.4844   10.7343
    2.3718    4.1205    6.5125    9.5230
    1.9635    3.4922    5.6401    8.4090
    1.6198    2.9476    4.8630    7.3921
```

Plot an Option Price Surface

Use the `Strike` input to specify the strikes. Increase the value for `NumFFT` to support a wider range of strikes. Also, the `Maturity` input can be a vector. Set `ExpandOutput` to `"true"` to output the surface as a `NStrikes`-by-`NMaturities` matrix.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

% Increase 'NumFFT' to support a wider range of strikes
NumFFT = 2^13;

Call = optByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'NumFFT', NumFFT, ...
    'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, 'ExpandOutput', true);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Call);
title('Price');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
zlim([0 80]);
```



Input Arguments

`Rate` — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

numeric

Current underlying asset price, specified as numeric value using a scalar or a `NINST`-by-`1` or `NColumns`-by-`1` vector.

For more information on the proper dimensions for `AssetPrice`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Settle` — Option settlement date

datetime array | string array | date character vector

Option settlement date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `optByMertonFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Settle`, see the name-value pair argument `ExpandOutput`.

`Maturity` — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optByMertonFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Maturity`, see the name-value pair argument `ExpandOutput`.

`OptSpec` — Definition of option
cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

For more information on the proper dimensions for `OptSpec`, see the name-value pair argument `ExpandOutput`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a `NINST`-by-`1`, `NRows`-by-`1`, `NRows`-by-`NColumns` vector of strike prices.

If this input is an empty array (`[]`), option prices are computed on the entire FFT (or FRFT) strike grid, which is determined as `exp(log-strike grid)`. Each column of the log-strike grid has`'NumFFT'` points with `'LogStrikeStep'` spacing that are roughly centered around each element of `log(AssetPrice)`.

For more information on the proper dimensions for `Strike`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Sigma` — Volatility of underlying asset
numeric

Volatility of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal value where `log`(1+*J*) is normally distributed with mean (`log`(1+`MeanJ`)-0.5*`JumpVol`^2) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log`(1+*J*)
decimal

Standard deviation of `log`(1+*J*) where `J` is the random percentage jump size, specified as a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
numeric

Annual frequency of Poisson jump process, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,StrikeOut] = optByMertonFFT(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,Sigma,MeanJ,JumpVol,JumpFreq,'Basis',7)`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)

- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric value.

Data Types: `double`

`NumFFT` — Number of grid points in the characteristic function variable
`4096` (default) | numeric

Number of grid points in the characteristic function variable and in each column of the log-strike grid, specified as the comma-separated pair consisting of `'NumFFT'` and a scalar numeric value.

Data Types: `double`

`CharacteristicFcnStep` — Characteristic function variable grid spacing
`0.01` (default) | numeric

Characteristic function variable grid spacing, specified as the comma-separated pair consisting of `'CharacteristicFcnStep'` and a scalar numeric value.

Data Types: `double`

`LogStrikeStep` — Log-strike grid spacing
`2*pi/NumFFT/CharacteristicFcnStep` (default) | numeric

Log-strike grid spacing, specified as the comma-separated pair consisting of `'LogStrikeStep'` and a scalar numeric value.

Note If (`LogStrikeStep*CharacteristicFcnStep`) is `2*pi/NumFFT`, FFT is used. Otherwise, FRFT is used.

Data Types: `double`

`DampingFactor` — Damping factor for Carr-Madan formulation
`1.5` (default) | numeric

Damping factor for Carr-Madan formulation, specified as the comma-separated pair consisting of `'DampingFactor'` and a scalar numeric value.

Data Types: `double`

`Quadrature` — Type of quadrature
`"simpson"` (default) | character vector with values: `'simpson'` or `'trapezoidal'` | string array with values: `"simpson"` or `"trapezoidal"`

Type of quadrature, specified as the comma-separated pair consisting of `'Quadrature'` and a single character vector or string array with a value of `'simpson'` or `'trapezoidal'`.

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-1 vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-1 vector, or a `NRows`-by-`NColumns` matrix. If `Strike` is empty, `NRows` is equal to `NumFFT`. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-1 vectors.
- `false` — If `false`, the outputs are `NINST`-by-1 vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-1 vectors.

Data Types: `logical`

Output Arguments

`Price` — Option prices
numeric

Option prices, returned as a `NINST`-by-1, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

`StrikeOut` — Strikes corresponding to `Price`
numeric

Strikes corresponding to `Price`, returned as a `NINST`-by-1, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Merton Jump Diffusion Model

The Merton jump diffusion model (Merton (1976)) is a different extension of the Black-Scholes model, where sudden asset price movements (both up and down) are modeled by adding the jump diffusion parameters with the Poisson process.

The stochastic differential equation is:

$$dS_t = (r - q - \lambda_p \mu_j) S_t dt + \sigma S_t dW_t + J S_t dP_t$$
$$\text{prob}(dP_t = 1) = \lambda_p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

W_t is the Weiner process.

J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1 + J)\delta\sqrt{2\pi}} \exp\left\{ \frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2} \right\}$$

μ_J is the mean of J for (μ_J > -1).

δ is the standard deviation of $\ln(1+J)$ for ($\delta \geq 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \geq 0$).

σ is the volatility of the asset price for ($\sigma > 0$).

The characteristic function $f_{Merton76_j}(\phi)$ for $j = 1$ (asset prices measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Merton76_j} = f_{BS_j} \exp\left(\lambda_p \tau \left(1 + \mu_j\right)^{m_j + \frac{1}{2}} \left[\left(1 + \mu_j\right)^{i\phi} e^{\delta^2 \left(m_j i\phi + \frac{(i\phi)^2}{2} \right)} - 1 \right] - \lambda_p \tau \mu_j i\phi \right)$$

where for $j = 1, 2$:

$$f_{BS_1}(\phi) = \frac{f_{BS_2}(\phi - i)}{f_{BS_2}(-i)}$$

$$f_{BS2}(\phi) = \exp\left(i\phi \left[\ln S_t + \left(r - q - \frac{\sigma^2}{2} \right) \tau \right] - \frac{\phi^2 \sigma^2}{2} \tau \right)$$

$$m_1 = \frac{1}{2}, m_2 = -\frac{1}{2}$$

where

ϕ is the characteristic function variable.

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

Carr-Madan Formulation

The Carr and Madan (1999) formulation is a popular modified implementation of Heston (1993) framework.

Rather than computing the probabilities P_1 and P_2 as intermediate steps, Carr and Madan developed an alternative expression so that taking its inverse Fourier transform gives the option price itself directly.

$$Call(k) = \frac{e^{-\alpha k}}{\pi}\int_0^\infty \mathrm{Re}\left[e^{-iuk}\psi(u)\right]du$$

$$\psi(u) = \frac{e^{-r\tau}f_2(\phi = (u - (\alpha + 1)i))}{\alpha^2 + \alpha - u^2 + iu(2\alpha + 1)}$$

$$Put(K) = Call(K) + Ke^{-r\tau} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

τ is time to maturity ($\tau = T-t$).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K

i is a unit imaginary number ($i^2 = -1$)

ϕ is the characteristic function variable.

α is the damping factor.

u is the characteristic function variable for integration, where $\phi = (u - (\alpha+1)i)$.

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

To apply FFT or FRFT to this formulation, the characteristic function variable for integration, u, is discretized into `NumFFT`(N) points with the step size `CharacteristicFcnStep` (Δu), and the log-strike k is discretized into N points with the step size `LogStrikeStep`(Δk).

The discretized characteristic function variable for integration, u_j(for $j = 1,2,3,\ldots,N$), has a minimum value of 0 and a maximum value of (N-1) (Δu), and it approximates the continuous integration range from 0 to infinity.

The discretized log-strike grid, k_n(for n = 1, 2, 3, N) is approximately centered around `ln`(S_t), with a minimum value of

$$\ln(S_t) - \frac{N}{2}\Delta k$$

and a maximum value of

$$\ln(S_t) + \left(\frac{N}{2} - 1\right)\Delta k$$

Where the minimum allowable strike is

$$S_t \exp\left(-\frac{N}{2}\Delta k\right)$$

and the maximum allowable strike is

$$S_t \exp\left[\left(\frac{N}{2} - 1\right)\Delta k\right]$$

As a result of the discretization, the expression for the call option becomes

$$Call(k_n) = \Delta u \frac{e^{-\alpha k_n}}{\pi} \sum_{j=1}^{N} \mathrm{Re}\left[e^{-i\Delta k \Delta u (j-1)(n-1)} e^{iu_j\left[\frac{N\Delta k}{2} - \ln(S_t)\right]} \psi(u_j)\right] w_j$$

where

Δu is the step size of discretized characteristic function variable for integration.

Δk is the step size of discretized log-strike.

N is the number of FFT or FRFT points.

w_j is the weights for quadrature used for approximating the integral.

FFT is used to evaluate the above expression if Δk and Δu are subject to the following constraint:

$$\Delta k \Delta u = \left(\frac{2\pi}{N}\right)$$

otherwise, the functions use the FRFT method described in Chourdakis (2005).

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optByMertonFFT` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bates, D. S. “Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options.” *The Review of Financial Studies*. Vol 9. No. 1. 1996.

[2] Carr, P., and D.B. Madan. “Option Valuation Using the Fast Fourier Transform.” *Journal of Computational Finance.* Vol 2. No. 4. 1999.

[3] Cont, R. and P. Tankov. *Financial Modeling with Jump Processes*. Chapman & Hall/CRC Press, 2004.

[4] Chourdakis, K. “Option Pricing Using Fractional FFT.” *Journal of Computational Finance*. 2005.

[5] Merton, R. “Option Pricing When Underlying Stock Returns are Discontinuous.” *Journal of Financial Economics.* Vol 3. 1976.

See Also

`optByHestonFFT` | `optSensByHestonFFT` | `optByHestonNI` | `optSensByHestonNI` | `optByBatesFFT` | `optSensByBatesFFT` | `optByBatesNI` | `optSensByBatesNI` | `optSensByMertonFFT` | `optByMertonNI` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optSensByMertonFFT

Option price and sensitivities by Merton76 model using FFT and FRFT

Syntax

```
[PriceSens,StrikeOut] = optSensByMertonFFT(Rate,AssetPrice,Settle,Maturity,
OptSpec,Strike,Sigma,MeanJ,JumpVol,JumpFreq)
[PriceSens,StrikeOut] = optSensByMertonFFT( ___ ,Name,Value)
```

Description

`[PriceSens,StrikeOut] = optSensByMertonFFT(Rate,AssetPrice,Settle,Maturity, OptSpec,Strike,Sigma,MeanJ,JumpVol,JumpFreq)` computes vanilla European option price and sensitivities by Merton76 model, using Carr-Madan FFT and Chourdakis FRFT methods.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[PriceSens,StrikeOut] = optSensByMertonFFT( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Sensitivity Surface Using the Merton76 Model

Use `optSensByMertonFFT` to calibrate the FFT strike grid for sensitivities, compute option sensitivities, and plot option sensitivity surfaces.

Define Option Variables and Merton76 Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

Sigma = 0.16;
MeanJ = 0.02;
JumpVol = 0.08;
JumpFreq = 2;
```

Compute the Option Prices for the Entire FFT (or FRFT) Strike Grid, Without Specifying "Strike"

Compute option sensitivities and also output the corresponding strikes. If the `Strike` input is empty (`[]`), option sensitivities will be computed on the entire FFT (or FRFT) strike grid. The FFT (or FRFT) strike grid is determined as `exp(log-strike grid)`, where each column of the log-strike grid has `NumFFT` points with `LogStrikeStep` spacing that are roughly centered around each

element of `log(AssetPrice)`. The default value for `NumFFT` is 2^12. In addition to the sensitivities in the first output, the optional last output contains the corresponding strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = []; % Strike is not specified

[Delta, Kout] = optSensByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta");

% Show the lowest and highest strike values on the FFT strike grid
format
[Kout(1) Kout(end)]
```

```
ans = 1×2
10^138 ×

    0.0000    1.8798
```

```
% Show a subset of the strikes and corresponding option sensitivities
Range = (2046:2052);
[Kout(Range) Delta(Range)]
```

```
ans = 7×2

   50.4929    0.9895
   58.8640    0.9801
   68.6231    0.8816
   80.0000    0.5283
   93.2631    0.1551
  108.7251    0.0241
  126.7505    0.0025
```

Change the Number of FFT (or FRFT) Points and Compare with optSensByMertonNI

Try a different number of FFT (or FRFT) points, and compare the results with numerical integration. Unlike `optSensByMertonFFT`, which uses FFT (or FRFT) techniques for fast computation across the whole range of strikes, the `optSensByMertonNI` function uses direct numerical integration and it is typically slower, especially for multiple strikes. However, the values computed by `optSensByMertonNI` can serve as a benchmark for adjusting the settings for `optSensByMertonFFT`.

```
% Try a smaller number of FFT points
% (e.g. for faster performance or smaller memory footprint)
NumFFT = 2^10; % Smaller than the default value of 2^12
Strike = []; % Strike is not specified (will use the entire FFT strike grid)
[Delta, Kout] = optSensByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'NumFFT', NumFFT);

% Compare with numerical integration method
Range = (510:516);
Strike = Kout(Range);
DeltaFFT = Delta(Range);
DeltaNI = optSensByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
```

```
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta");
Error = abs(DeltaFFT-DeltaNI);
table(Strike, DeltaFFT, DeltaNI, Error)
```

```
ans=7×4 table
    Strike     DeltaFFT      DeltaNI        Error
    ______    __________    __________    _________

    12.696       0.89726       0.99002      0.092766
    23.449       0.93421       0.99002       0.05581
    43.312       0.94691       0.99001      0.043093
        80       0.50983       0.52827      0.018446
    147.76      0.004147    0.00019101      0.003956
    272.93      0.001071     1.547e-09      0.001071
    504.11    0.00030521    5.7578e-10    0.00030521
```

Make Further Adjustments to FFT (or FRFT)

If the values in the output `DeltaFFT` are significantly different from those in `DeltaNI`, try making adjustments to `optSensByMertonFFT` settings, such as `CharacteristicFcnStep`, `LogStrikeStep`, `NumFFT`, `DampingFactor`, and so on. Note that if (`LogStrikeStep` * `CharacteristicFcnStep`) is 2*pi/ `NumFFT`, FFT is used. Otherwise, FRFT is used.

```
Strike = []; % Strike is not specified (will use the entire FFT or FRFT strike grid)
[Delta, Kout] = optSensByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001);

% Compare with numerical integration method
Strike = Kout(Range);
DeltaFFT = Delta(Range);
DeltaNI = optSensByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta");
Error = abs(DeltaFFT-DeltaNI);
table(Strike, DeltaFFT, DeltaNI, Error)
```

```
ans=7×4 table
    Strike    DeltaFFT    DeltaNI      Error
    ______    ________    _______    __________

    79.76     0.53701     0.53701    5.6407e-12
    79.84      0.5341      0.5341    5.3257e-12
    79.92     0.53119     0.53119    5.0099e-12
       80     0.52827     0.52827    4.6956e-12
    80.08     0.52536     0.52536    4.3811e-12
    80.16     0.52245     0.52245    4.0653e-12
    80.24     0.51953     0.51953    3.7503e-12
```

```
% Save the final FFT (or FRFT) strike grid for future reference. For
% example, it provides information about the range of Strike inputs for
% which the FFT (or FRFT) operation is valid.
FFTStrikeGrid = Kout;
MinStrike = FFTStrikeGrid(1) % Strike cannot be less than MinStrike
```

```
MinStrike = 47.9437
```

```
MaxStrike = FFTStrikeGrid(end) % Strike cannot be greater than MaxStrike
```

```
MaxStrike = 133.3566
```

Compute the Option Sensitivity for a Single Strike

Once the desired FFT (or FRFT) settings are determined, use the `Strike` input to specify the strikes rather than providing an empty array. If the specified strikes do not match a value on the FFT (or FRFT) strike grid, the outputs are interpolated on the specified strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;

Delta = optSensByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001)
```

```
Delta = 0.5283
```

Compute the Option Sensitivities for a Vector of Strikes

Use the `Strike` input to specify the strikes.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Delta = optSensByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001)
```

```
Delta = 5×1

    0.6727
    0.6013
    0.5283
    0.4565
    0.3883
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Delta = optSensByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001) % Five values in vector output
```

```
Delta = 5×1
```

```
    0.6419
    0.5907
    0.5565
    0.5311
    0.5110
```

Expand the Outputs for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the outputs into `NStrikes`-by-`NMaturities` matrices. In this case, they are square matrices.

```
[Delta, Kout] = optSensByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001, 'ExpandOutput', true) % (5 x 5) matrix output
```

```
Delta = 5×5

    0.6419    0.6305    0.6245    0.6204    0.6173
    0.5922    0.5907    0.5905    0.5905    0.5905
    0.5422    0.5507    0.5565    0.5607    0.5637
    0.4927    0.5112    0.5229    0.5311    0.5372
    0.4447    0.4725    0.4898    0.5020    0.5110


Kout = 5×5

    76    76    76    76    76
    78    78    78    78    78
    80    80    80    80    80
    82    82    82    82    82
    84    84    84    84    84
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities`. That is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Delta = optSensByMertonFFT(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001, 'ExpandOutput', true) % (5 x 6) matrix output
```

```
Delta = 5×6

    0.6727    0.6419    0.6305    0.6245    0.6204    0.6173
    0.6013    0.5922    0.5907    0.5905    0.5905    0.5905
    0.5283    0.5422    0.5507    0.5565    0.5607    0.5637
    0.4565    0.4927    0.5112    0.5229    0.5311    0.5372
    0.3883    0.4447    0.4725    0.4898    0.5020    0.5110
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Asset Prices

When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

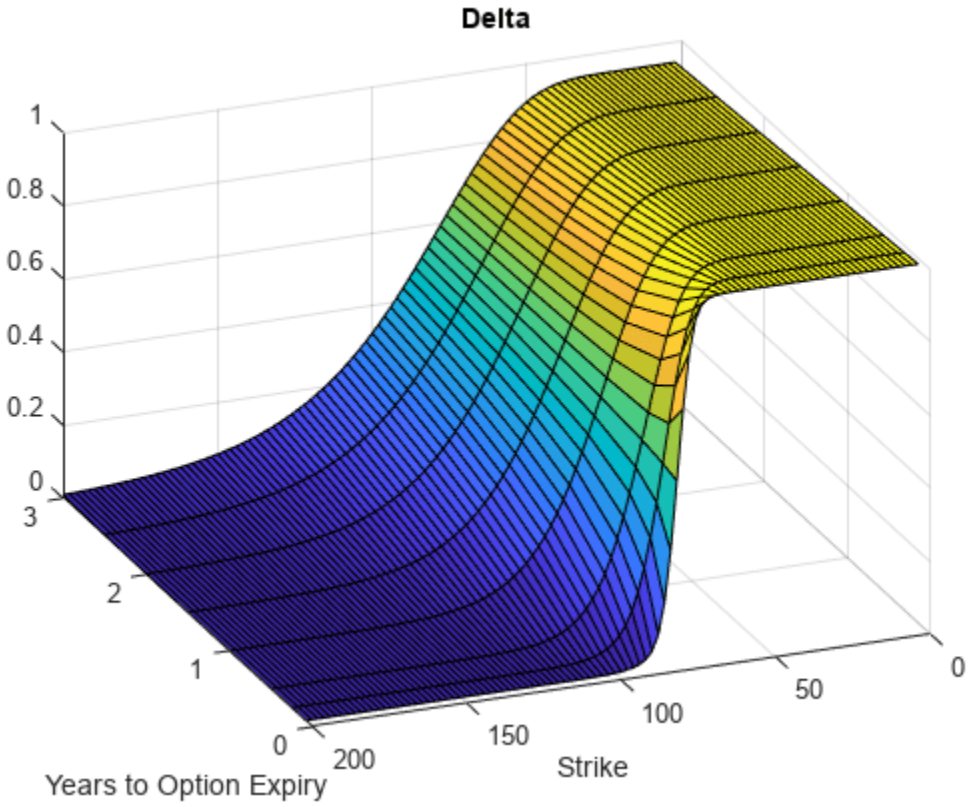
```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Delta = optSensByMertonFFT(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, 'OutSpec', "delta", ...
    'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, ...
    'LogStrikeStep', 0.001, 'ExpandOutput', true) % (5 x 4) matrix output
```

```
Delta = 5×4

    0.3796    0.5157    0.6419    0.7472
    0.3315    0.4637    0.5922    0.7043
    0.2874    0.4137    0.5422    0.6592
    0.2474    0.3664    0.4927    0.6128
    0.2117    0.3224    0.4447    0.5657
```

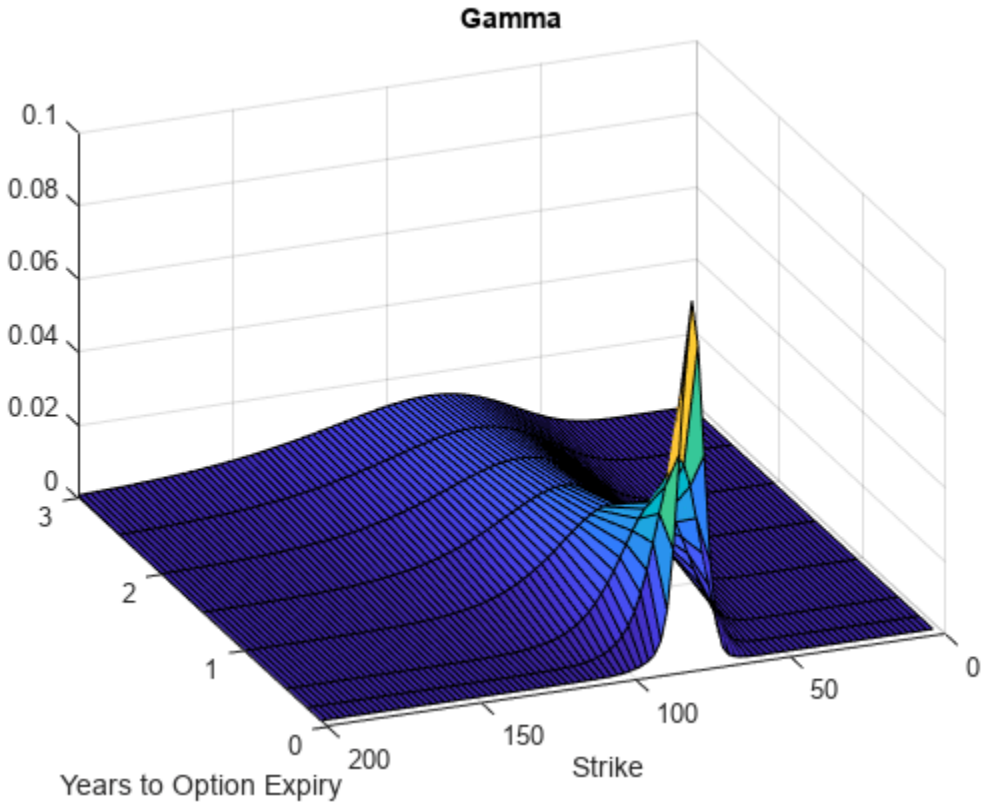
Plot Option Sensitivity Surfaces

Use the `Strike` input to specify the strikes. Increase the value for `NumFFT` to support a wider range of strikes. Also, the `Maturity` input can be a vector. Set `ExpandOutput` to `"true"` to output the surfaces as `NStrikes`-by-`NMaturities` matrices.

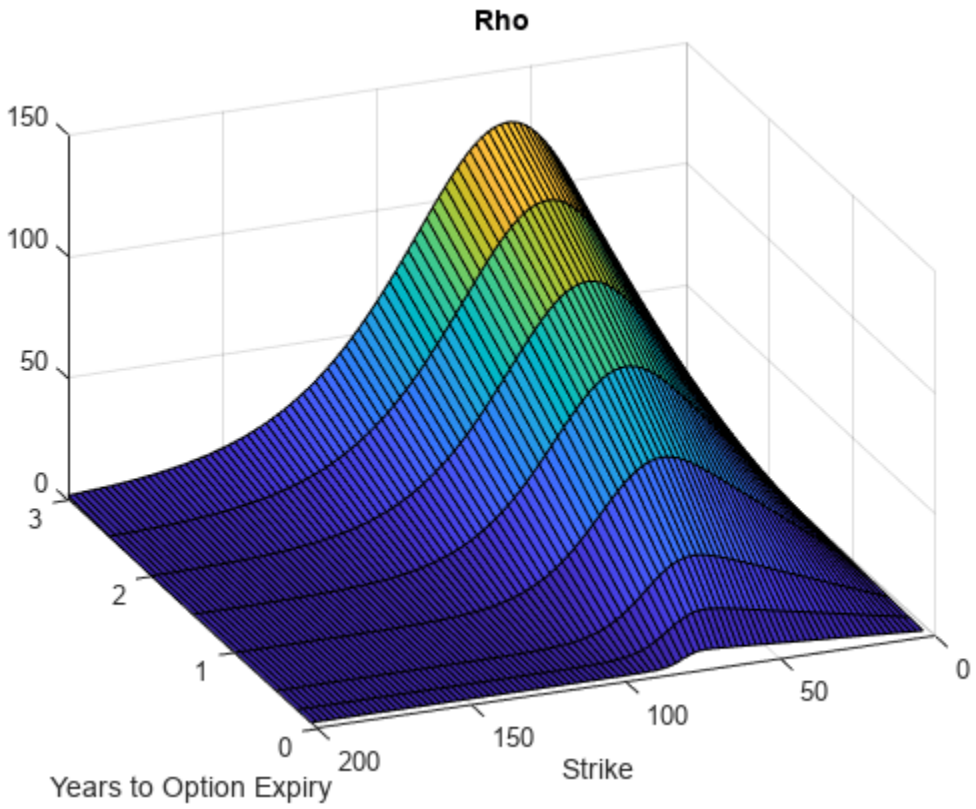
```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

% Increase 'NumFFT' to support a wider range of strikes
NumFFT = 2^13;

[Delta, Gamma, Rho, Theta, Vega] = optSensByMertonFFT(...
    Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'NumFFT', NumFFT, 'CharacteristicFcnStep', 0.065, 'LogStrikeStep', 0.001, ...
    'OutSpec', ["delta", "gamma", "rho", "theta", "vega"], ...
    'ExpandOutput', true);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Delta);
title('Delta');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
```



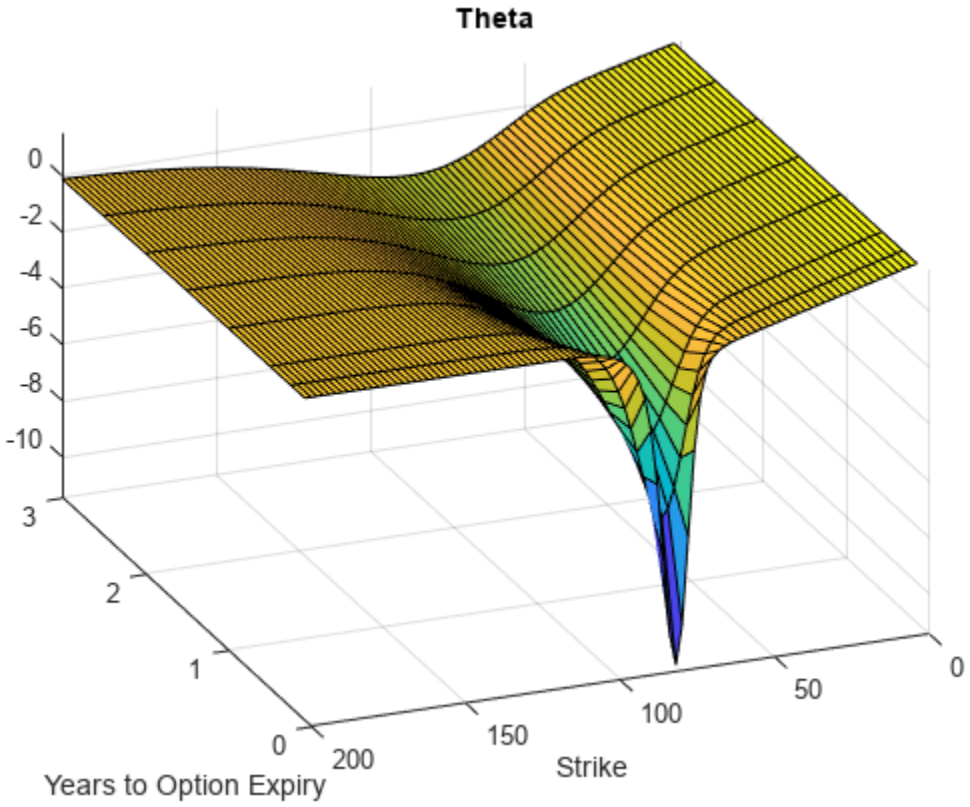

```
figure;
surf(X,Y,Gamma)
title('Gamma')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



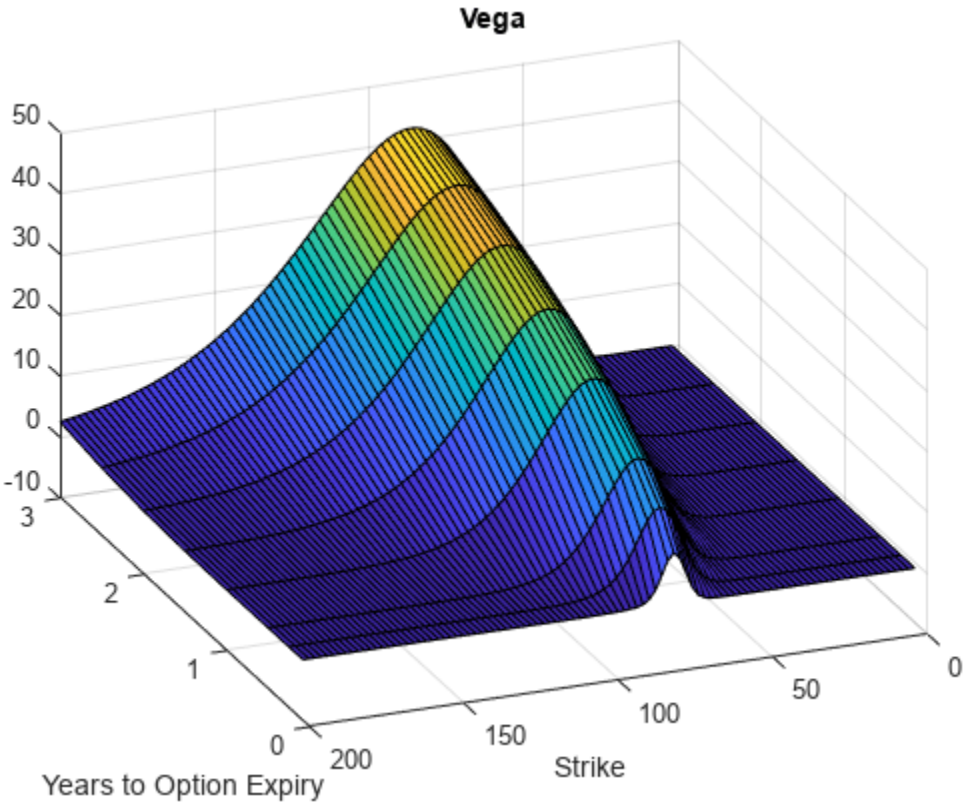

```
figure;
surf(X,Y,Rho)
title('Rho')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,Theta)
title('Theta')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,Vega)
title('Vega')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



Input Arguments

`Rate` — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

numeric

Current underlying asset price, specified as numeric value using a scalar or a `NINST`-by-`1` or `NColumns`-by-`1` vector.

For more information on the proper dimensions for `AssetPrice`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Settle` — Option settlement date

datetime array | string array | date character vector

Option settlement date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `optSensByMertonFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Settle`, see the name-value pair argument `ExpandOutput`.

`Maturity` — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optSensByMertonFFT` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Maturity`, see the name-value pair argument `ExpandOutput`.

`OptSpec` — Definition of option
cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

For more information on the proper dimensions for `OptSpec`, see the name-value pair argument `ExpandOutput`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a `NINST`-by-`1`, `NRows`-by-`1`, `NRows`-by-`NColumns` vector of strike prices.

If this input is an empty array (`[]`), option prices are computed on the entire FFT (or FRFT) strike grid, which is determined as `exp(log-strike grid)`. Each column of the log-strike grid has`'NumFFT'` points with `'LogStrikeStep'` spacing that are roughly centered around each element of `log(AssetPrice)`.

For more information on the proper dimensions for `Strike`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Sigma` — Volatility of underlying asset
numeric

Volatility of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal value where `log`(1+*J*) is normally distributed with mean (`log`(1+`MeanJ`)-0.5*`JumpVol^2`) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log(1+J)`
decimal

Standard deviation of `log`(1+*J*) where `J` is the random percentage jump size, specified as a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
numeric

Annual frequency of Poisson jump process, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PriceSens,StrikeOut] = optSensByMertonFFT(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,Sigma,MeanJ,JumpVol,JumpFreq,'Basis',7)`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)

- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric value.

Data Types: `double`

`OutSpec` — Define outputs
`["price"]` (default) | string array with values `"price"`, `"delta"`, `"gamma"`, `"vega"`, `"rho"`, and `"theta"` | cell array of character vectors with values `'price'`, `'delta'`, `'gamma'`, `'vega'`, `'rho'`, and `'theta'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` string array or cell array of character vectors with supported values.

Note `"vega"` is the sensitivity with respect the initial volatility sqrt(`V0`).

Example: `OutSpec = ["price","delta","gamma","vega","rho","theta"]`

Data Types: `string` | `cell`

`NumFFT` — Number of grid points in the characteristic function variable
`4096` (default) | numeric

Number of grid points in the characteristic function variable and in each column of the log-strike grid, specified as the comma-separated pair consisting of `'NumFFT'` and a scalar numeric value.

Data Types: `double`

`CharacteristicFcnStep` — Characteristic function variable grid spacing
`0.01` (default) | numeric

Characteristic function variable grid spacing, specified as the comma-separated pair consisting of `'CharacteristicFcnStep'` and a scalar numeric value.

Data Types: `double`

`LogStrikeStep` — Log-strike grid spacing
`2*pi/NumFFT/CharacteristicFcnStep` (default) | numeric

Log-strike grid spacing, specified as the comma-separated pair consisting of `'LogStrikeStep'` and a scalar numeric value.

Note If (`LogStrikeStep*CharacteristicFcnStep`) is `2*pi/NumFFT`, FFT is used. Otherwise, FRFT is used.

Data Types: double

DampingFactor — Damping factor for Carr-Madan formulation
1.5 (default) | numeric

Damping factor for Carr-Madan formulation, specified as the comma-separated pair consisting of 'DampingFactor' and a scalar numeric value.

Data Types: double

Quadrature — Type of quadrature
"simpson" (default) | character vector with values: 'simpson' or 'trapezoidal' | string array with values: "simpson" or "trapezoidal"

Type of quadrature, specified as the comma-separated pair consisting of 'Quadrature' and a single character vector or string array with a value of 'simpson' or 'trapezoidal'.

Data Types: char | string

ExpandOutput — Flag to expand the outputs
false (outputs are NINST-by-1 vectors) (default) | logical with value of true or false

Flag to expand the outputs, specified as the comma-separated pair consisting of 'ExpandOutput' and a logical:

- true — If true, the outputs are NRows-by- NColumns matrices. NRows is the number of strikes for each column and it is determined by the Strike input. For example, Strike can be a NRows-by-1 vector, or a NRows-by-NColumns matrix. If Strike is empty, NRows is equal to NumFFT. NColumns is determined by the sizes of AssetPrice, Settle, Maturity, and OptSpec, which must all be either scalar or NColumns-by-1 vectors.
- false — If false, the outputs are NINST-by-1 vectors. Also, the inputs Strike, AssetPrice, Settle, Maturity, and OptSpec must all be either scalar or NINST-by-1 vectors.

Data Types: logical

Output Arguments

PriceSens — Option prices or sensitivities
numeric

Option prices or sensitivities, returned as a NINST-by-1, or NRows-by-NColumns, depending on ExpandOutput. The name-value pair argument OutSpec determines the types and order of the outputs.

StrikeOut — Strikes corresponding to Price
numeric

Strikes corresponding to Price, returned as a NINST-by-1, or NRows-by-NColumns, depending on ExpandOutput.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Merton Jump Diffusion Model

The Merton jump diffusion model (Merton (1976)) is a different extension of the Black-Scholes model, where sudden asset price movements (both up and down) are modeled by adding the jump diffusion parameters with the Poisson process.

The stochastic differential equation is:

$$dS_t = (r - q - \lambda_p \mu_j)S_t dt + \sigma S_t dW_t + JS_t dP_t$$
$$\text{prob}(dP_t = 1) = \lambda_p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

W_t is the Weiner process.

J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1 + J)\delta\sqrt{2\pi}} \exp\left\{ \frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2} \right\}$$

μ_J is the mean of J for ($\mu_J > -1$).

δ is the standard deviation of $\ln(1+J)$ for ($\delta \geq 0$).

λ_p is the annual frequency (intensity) of Poisson process $P_t$for ($\lambda_p \geq 0$).

σ is the volatility of the asset price for ($\sigma > 0$).

The characteristic function $f_{Merton76_j}(\phi)$ for $j = 1$ (asset prices measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Merton76_j} = f_{BS_j}\exp\left(\lambda_p\tau\left(1+\mu_j\right)^{m_j} + \frac{1}{2}\left[\left(1+\mu_j\right)^{i\phi}e^{\delta^2\left(m_j i\phi + \frac{(i\phi)^2}{2}\right)} - 1\right] - \lambda_p\tau\mu_j i\phi\right)$$

where for $j = 1, 2$:

$$f_{BS_1}(\phi) = \frac{f_{BS_2}(\phi - i)}{f_{BS_2}(-i)}$$

$$f_{BS2}(\phi) = \exp\left(i\phi\left[\ln S_t + \left(r - q - \frac{\sigma^2}{2}\right)\tau\right] - \frac{\phi^2\sigma^2}{2}\tau\right)$$

$$m_1 = \frac{1}{2}, m_2 = -\frac{1}{2}$$

where

ϕ is the characteristic function variable.

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

Carr-Madan Formulation

The Carr and Madan (1999) formulation is a popular modified implementation of Heston (1993) framework.

Rather than computing the probabilities P_1 and P_2 as intermediate steps, Carr and Madan developed an alternative expression so that taking its inverse Fourier transform gives the option price itself directly.

$$Call(k) = \frac{e^{-\alpha k}}{\pi}\int_0^\infty \mathrm{Re}\left[e^{-iuk}\psi(u)\right]du$$

$$\psi(u) = \frac{e^{-r\tau}f_2(\phi = (u - (\alpha+1)i))}{\alpha^2 + \alpha - u^2 + iu(2\alpha+1)}$$

$$Put(K) = Call(K) + Ke^{-r\tau} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

τ is time to maturity ($\tau = T - t$).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number ($i^2 = -1$).

ϕ is the characteristic function variable.

α is the damping factor.

u is the characteristic function variable for integration, where $\phi = (u - (\alpha+1)i)$.

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

To apply FFT or FRFT to this formulation, the characteristic function variable for integration, u, is discretized into `NumFFT`(N) point with the step size `CharacteristicFcnStep` (Δu), and the log-strike k is discretized into N points with the step size `LogStrikeStep`(Δk).

The discretized characteristic function variable for integration, u_j(for j = 1,2,3,...,N), has a minimum value of 0 and a maximum value of (N-1) (Δu), and it approximates the continuous integration range from 0 to infinity.

The discretized log-strike grid, k_n(for n = 1, 2, 3, N) is approximately centered around `ln`(S_t), with a minimum value of

$$\ln(S_t) - \frac{N}{2}\Delta k$$

and a maximum value of

$$\ln(S_t) + \left(\frac{N}{2} - 1\right)\Delta k$$

Where the minimum allowable strike is

$$S_t \exp\left(-\frac{N}{2}\Delta k\right)$$

and the maximum allowable strike is

$$S_t \exp\left[\left(\frac{N}{2} - 1\right)\Delta k\right]$$

As a result of the discretization, the expression for the call option becomes

$$Call(k_n) = \Delta u \frac{e^{-\alpha k_n}}{\pi} \sum_{j=1}^{N} \mathrm{Re}\left[e^{-i\Delta k \Delta u (j-1)(n-1)} e^{iuj\left[\frac{N\Delta k}{2} - \ln(S_t)\right]} \psi(u_j)\right] w_j$$

where

Δu is the step size of discretized characteristic function variable for integration.

Δk is the step size of discretized log-strike.

N is the number of FFT or FRFT points.

w_j is the weights for quadrature used for approximating the integral.

FFT is used to evaluate the above expression if Δk and Δu are subject to the following constraint:

$$\Delta k \Delta u = \left(\frac{2\pi}{N}\right)$$

otherwise, the functions use the FRFT method described in Chourdakis (2005).

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optSensByMertonFFT` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bates, D. S. “Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options.” *The Review of Financial Studies*. Vol 9. No. 1. 1996.

[2] Carr, P., and D.B. Madan. “Option Valuation Using the Fast Fourier Transform.” *Journal of Computational Finance.* Vol 2. No. 4. 1999.

[3] Cont, R. and P. Tankov. *Financial Modeling with Jump Processes*. Chapman & Hall/CRC Press, 2004.

[4] Chourdakis, K. “Option Pricing Using Fractional FFT.” *Journal of Computational Finance.* 2005.

[5] Merton, R. “Option Pricing When Underlying Stock Returns are Discontinuous.” *Journal of Financial Economics.* Vol 3. 1976.

See Also

`optByHestonFFT` | `optSensByHestonFFT` | `optByHestonNI` | `optSensByHestonNI` | `optByBatesFFT` | `optSensByBatesFFT` | `optByBatesNI` | `optSensByBatesNI` | `optByMertonFFT` | `optByMertonNI` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19

“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optByMertonNI

Option price by Merton76 model using numerical integration

Syntax

```
Price = optByMertonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,Sigma,
MeanJ,JumpVol,JumpFreq)
Price = optByMertonNI( ___ ,Name,Value)
```

Description

`Price = optByMertonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,Sigma, MeanJ,JumpVol,JumpFreq)` computes vanilla European option price by the Merton76 model, using numerical integration.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = optByMertonNI( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Price Surface Using the Merton76 Model

`optByMertonNI` uses numerical integration to compute option prices and then plot an option price surface.

Define Option Variables and Merton76 Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

Sigma = 0.16;
MeanJ = 0.02;
JumpVol = 0.08;
JumpFreq = 2;
```

Compute the Option Price for a Single Strike

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;

Call = optByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield)
```

```
Call = 4.5600
```

Compute the Option Prices for a Vector of Strikes

The `Strike` input can be a vector.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Call = optByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield)
```

```
Call = 5×1

    6.7410
    5.5762
    4.5600
    3.6891
    2.9551
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Call = optByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield)
```

```
Call = 5×1

    8.5589
    8.9439
    9.2316
    9.4653
    9.6565


    % Five values in vector output
```

Expand the Output for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the output into a `NStrikes`-by-`NMaturities` matrix. In this case, it is a square matrix.

```
Call = optByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'ExpandOutput', true) % (5 x 5) matrix output
```

```
Call = 5×5

    8.5589    9.9675   11.1343   12.1492   13.0464
```

```
    7.4844    8.9439   10.1481   11.1939   12.1181
    6.5125    8.0023    9.2316   10.2999   11.2449
    5.6401    7.1402    8.3827    9.4653   10.4249
    4.8630    6.3545    7.5990    8.6881    9.6565
```

Compute the Option Prices for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities`. That is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Call = optByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'ExpandOutput', true) % (5 x 6) matrix output
```

```
Call = 5×6

    6.7410    8.5589    9.9675   11.1343   12.1492   13.0464
    5.5762    7.4844    8.9439   10.1481   11.1939   12.1181
    4.5600    6.5125    8.0023    9.2316   10.2999   11.2449
    3.6891    5.6401    7.1402    8.3827    9.4653   10.4249
    2.9551    4.8630    6.3545    7.5990    8.6881    9.6565
```

Compute the Option Prices for a Vector of Strikes and a Vector of Asset Prices

When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Call = optByMertonNI(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'ExpandOutput', true) % (5 x 4) matrix output
```

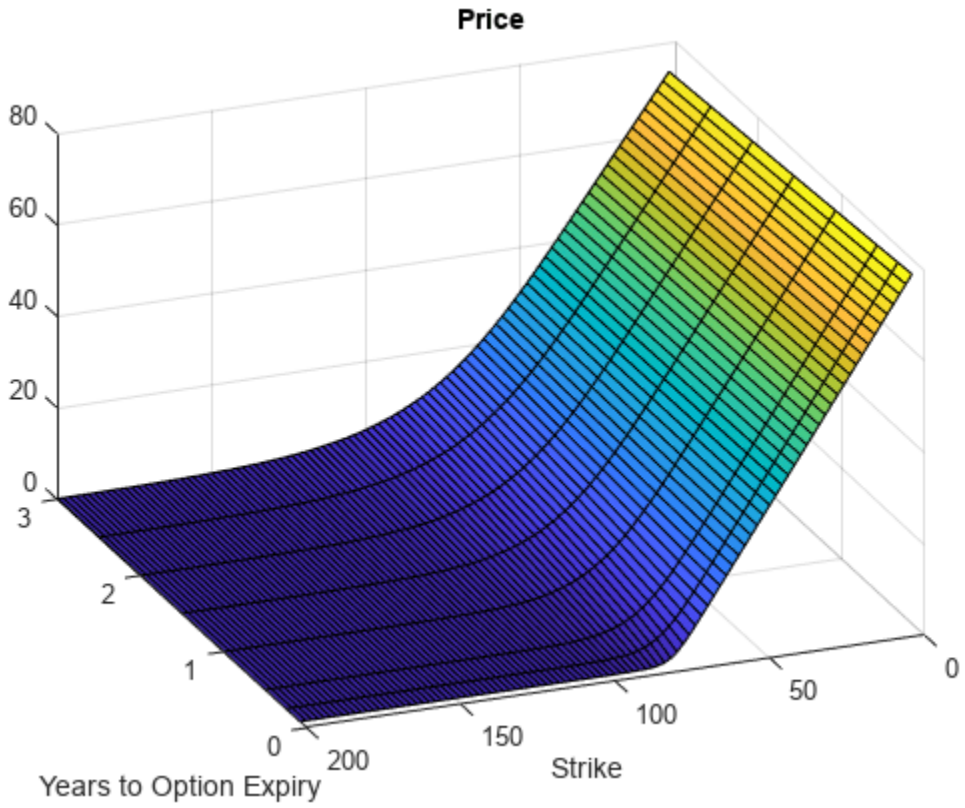
```
Call = 5×4

    3.4186    5.6579    8.5589   12.0417
    2.8538    4.8401    7.4844   10.7343
    2.3718    4.1205    6.5125    9.5230
    1.9635    3.4922    5.6401    8.4090
    1.6198    2.9476    4.8630    7.3921
```

Plot an Option Price Surface

The `Strike` and `Maturity` inputs can be vectors. Set `ExpandOutput` to `"true"` to output the surface as a `NStrikes`-by-`NMaturities` matrix.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
```

```
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

Call = optByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'ExpandOutput', true);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Call);
title('Price');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
zlim([0 80]);
```



Input Arguments

Rate — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

AssetPrice — Current underlying asset price
numeric

Current underlying asset price, specified as numeric value using a scalar or a NINST-by-1 or NColumns-by-1 vector.

For more information on the proper dimensions for AssetPrice, see the name-value pair argument ExpandOutput.

Data Types: double

Settle — Option settlement date
datetime array | string array | date character vector

Option settlement date, specified as a NINST-by-1 or NColumns-by-1 vector using a datetime array, string array, or date character vectors. The Settle date must be before the Maturity date.

To support existing code, optByMertonNI also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for Settle, see the name-value pair argument ExpandOutput.

Maturity — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a NINST-by-1 or NColumns-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optByMertonNI also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for Maturity, see the name-value pair argument ExpandOutput.

OptSpec — Definition of option
cell array of character vector with values 'call' or 'put' | string array with values "call" or "put"

Definition of the option, specified as a NINST-by-1 or NColumns-by-1 vector using a cell array of character vectors or string arrays with values 'call' or 'put'.

For more information on the proper dimensions for OptSpec, see the name-value pair argument ExpandOutput.

Data Types: cell | string

Strike — Option strike price value
numeric

Option strike price value, specified as a NINST-by-1, NRows-by-1, NRows-by-NColumns vector of strike prices.

For more information on the proper dimensions for Strike, see the name-value pair argument ExpandOutput.

Data Types: double

`Sigma` — Volatility of underlying asset

numeric

Volatility of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size

decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal value where `log`(1+*J*) is normally distributed with mean (`log`(1+`MeanJ`)-0.5*`JumpVol^2`) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log(1+`*J*`)`

decimal

Standard deviation of `log`(1+*J*) where `J` is the random percentage jump size, specified as a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process

numeric

Annual frequency of Poisson jump process, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optByMertonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,Sigma,MeanJ,JumpVol,JumpFreq,'Basis',7)`

`Basis` — Day-count basis of instrument

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)

- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`DividendYield` — Continuously compounded underlying asset yield
`0` (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of `'DividendYield'` and a scalar numeric value.

Data Types: `double`

`AbsTol` — Absolute error tolerance for numerical integration
`1e-10` (default) | numeric

Absolute error tolerance for numerical integration, specified as the comma-separated pair consisting of `'AbsTol'` and a scalar numeric value.

Data Types: `double`

`RelTol` — Relative error tolerance for numerical integration
`1e-6` (default) | numeric

Relative error tolerance for numerical integration, specified as the comma-separated pair consisting of `'RelTol'` and a scalar numeric value.

Data Types: `double`

`IntegrationRange` — Numerical integration range used to approximate continuous integral over `[0 Inf]`
`[1e-9 Inf]` (default) | vector

Numerical integration range used to approximate the continuous integral over `[0 Inf]`, specified as the comma-separated pair consisting of `'IntegrationRange'` and a 1-by-2 vector representing `[LowerLimit UpperLimit]`.

Data Types: `double`

`Framework` — Framework for computing option prices and sensitivities using numerical integration of models
`"heston1993"` (default) | string with values `"heston1993"` or `"lewis2001"` | character vector with values `'heston1993'` or `'lewis2001'`

Framework for computing option prices and sensitivities using numerical integration of models, specified as the comma-separated pair consisting of `'Framework'` and a scalar string or character vector with the following values:

- `"heston1993"` or `'heston1993'` — Method used in Heston (1993)
- `"lewis2001"` or `'lewis2001'` — Method used in Lewis (2001)

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-1 vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-1 vector, or a `NRows`-by-`NColumns` matrix. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-1 vectors.
- `false` — If `false`, the outputs are `NINST`-by-1 vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-1 vectors.

Data Types: `logical`

Output Arguments

`Price` — Option prices
numeric

Option prices, returned as a `NINST`-by-1, or `NRows`-by-`NColumns`, depending on `ExpandOutput`.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Merton Jump Diffusion Model

The Merton jump diffusion model (Merton (1976)) is a different extension of the Black-Scholes model, where sudden asset price movements (both up and down) are modeled by adding the jump diffusion parameters with the Poisson process.

The stochastic differential equation is:

$$dS_t = (r - q - \lambda_p \mu_j) S_t dt + \sigma S_t dW_t + J S_t dP_t$$
$$\text{prob}(dP_t = 1) = \lambda_p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

W_t is the Weiner process.

J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1 + J)\delta\sqrt{2\pi}} \exp\left\{ \frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2} \right\}$$

μ_J is the mean of J for (μ_J > -1).

δ is the standard deviation of $\ln(1+J)$ for ($\delta \geq 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \geq 0$).

σ is the volatility of the asset price for ($\sigma > 0$).

The characteristic function $f_{Merton76_j}(\phi)$ for $j = 1$ (asset prices measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Merton76_j} = f_{BS_j} \exp\left(\lambda_p \tau \left(1 + \mu_j\right)^{m_j + \frac{1}{2}} \left[\left(1 + \mu_j\right)^{i\phi} e^{\delta^2 \left(m_j i\phi + \frac{(i\phi)^2}{2} \right)} - 1 \right] - \lambda_p \tau \mu_j i\phi \right)$$

where for $j = 1, 2$:

$$f_{BS_1}(\phi) = \frac{f_{BS_2}(\phi - i)}{f_{BS_2}(-i)}$$

$$f_{BS2}(\phi) = \exp\left(i\phi \left[\ln S_t + \left(r - q - \frac{\sigma^2}{2} \right) \tau \right] - \frac{\phi^2 \sigma^2}{2} \tau \right)$$

$$m_1 = \frac{1}{2}, m_2 = -\frac{1}{2}$$

where

ϕ is the characteristic function variable

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

Numerical Integration Method Under Heston (1993) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Heston (1993) framework is based on the following expressions:

$$Call(K) = S_t e^{-q\tau} P_1 - K e^{-r\tau} P_2$$

$$Put(K) = Call(K) + K e^{-r\tau} - S_t e^{-q\tau}$$

$$P_j = \frac{1}{2} + \frac{1}{\pi} \int_0^{\infty} \mathrm{Re}\left[\frac{e^{-i\phi \ln(K)} f_j(\phi)}{i\phi} \right] d\phi$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity ($\tau = T - t$).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K

i is a unit imaginary number ($i^2 = -1$)

ϕ is the characteristic function variable.

$f_j(\phi)$ is the characteristic function for $P_j (j = 1,2)$.

P_1 is the probability of $S_t > K$ under the asset price measure for the model.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

Where $j = 1,2$ so that $f_1(\phi)$ and $f_2(\phi)$ are the characteristic functions for probabilities P_1 and P_2, respectively.

This framework is chosen with the default value “`Heston1993`” for the `Framework` name-value pair argument.

Numerical Integration Method Under Lewis (2001) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Lewis (2001) framework is based on the following expressions:

$$Call(k) = S_t e^{-q\tau} - \frac{\sqrt{K}e^{-\tau t}}{\pi}\int_0^{\infty} \mathrm{Re}\left[K^{-iu} f_2\left(\phi = \left(u - \frac{i}{2}\right)\right)\frac{1}{u^2 + \frac{1}{4}}\right]du$$

$$Put(K) = Call(K) = Ke^{-\tau t} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K

i is a unit imaginary number (i^2= -1)

ϕ is the characteristic function variable.

u is the characteristic function variable for integration, where $\phi = \left(u - \frac{i}{2}\right)$.

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

This framework is chosen with the value “`Lewis2001`” for the `Framework` name-value pair argument.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optByMertonNI` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bates, D. S. “Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options.” *The Review of Financial Studies.* Vol 9. No. 1. 1996.

[2] Cont, R. and P. Tankov. *Financial Modeling with Jump Processes*. Chapman & Hall/CRC Press, 2004.

[3] Heston, S. L. “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.” *The Review of Financial Studies.* Vol 6. No. 2. 1993.

[4] Lewis, A. L. “A Simple Option Formula for General Jump-Diffusion and Other Exponential Levy Processes.” Envision Financial Systems and OptionCity.net, 2001.

[5] Merton, R. “Option Pricing When Underlying Stock Returns are Discontinuous.” *Journal of Financial Economics.* Vol 3. 1976.

See Also

`optByHestonFFT` | `optSensByHestonFFT` | `optByHestonNI` | `optSensByHestonNI` | `optByBatesFFT` | `optSensByBatesFFT` | `optByBatesNI` | `optSensByBatesNI` | `optByMertonFFT` | `optSensByMertonFFT` | `optSensByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optSensByMertonNI

Option price and sensitivities by Merton76 model using numerical integration

Syntax

```
PriceSens = optSensByMertonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,
Sigma,MeanJ,JumpVol,JumpFreq)
PriceSens = optSensByMertonNI( ___ ,Name,Value)
```

Description

`PriceSens = optSensByMertonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike, Sigma,MeanJ,JumpVol,JumpFreq)` computes vanilla European option price and sensitivities by the Merton76 model, using numerical integration.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = optSensByMertonNI( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Workflow for Plotting an Option Sensitivity Surface Using the Merton76 Model

`optSensByMertonNI` uses numerical integration to compute option sensitivities and then plot option sensitivity surfaces.

Define Option Variables and Merton76 Model Parameters

```
AssetPrice = 80;
Rate = 0.03;
DividendYield = 0.02;
OptSpec = 'call';

Sigma = 0.16;
MeanJ = 0.02;
JumpVol = 0.08;
JumpFreq = 2;
```

Compute the Option Sensitivity for a Single Strike

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = 80;

Delta = optSensByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
```

```
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'OutSpec', "delta")

Delta = 0.5283
```

Compute the Option Sensitivities for a Vector of Strikes

The `Strike` input can be a vector.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 6);
Strike = (76:2:84)';

Delta = optSensByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'OutSpec', "delta")

Delta = 5×1

    0.6727
    0.6013
    0.5283
    0.4565
    0.3883
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of the Same Lengths

Use the `Strike` input to specify the strikes. Also, the `Maturity` input can be a vector, but it must match the length of the `Strike` vector if the `ExpandOutput` name-value pair argument is not set to `"true"`.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, [12 18 24 30 36]); % Five maturities
Strike = [76 78 80 82 84]'; % Five strikes

Delta = optSensByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'OutSpec', "delta")

Delta = 5×1

    0.6419
    0.5907
    0.5565
    0.5311
    0.5110


    % Five values in vector output
```

Expand the Output for a Surface

Set the `ExpandOutput` name-value pair argument to `"true"` to expand the output into a `NStrikes`-by-`NMaturities` matrix. In this case, it is a square matrix.

```
Delta = optSensByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'ExpandOutput', true) % (5 x 5) matrix output

Delta = 5×5

    0.6419    0.6305    0.6245    0.6204    0.6173
    0.5922    0.5907    0.5905    0.5905    0.5905
    0.5422    0.5507    0.5565    0.5607    0.5637
    0.4927    0.5112    0.5229    0.5311    0.5372
    0.4447    0.4725    0.4898    0.5020    0.5110
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Dates of Different Lengths

When `ExpandOutput` is `"true"`, `NStrikes` do not have to match `NMaturities`. That is, the output `NStrikes`-by-`NMaturities` matrix can be rectangular.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*(0.5:0.5:3)'); % Six maturities
Strike = (76:2:84)'; % Five strikes

Delta = optSensByMertonNI(Rate, AssetPrice, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'ExpandOutput', true)  % (5 x 6) matrix output

Delta = 5×6

    0.6727    0.6419    0.6305    0.6245    0.6204    0.6173
    0.6013    0.5922    0.5907    0.5905    0.5905    0.5905
    0.5283    0.5422    0.5507    0.5565    0.5607    0.5637
    0.4565    0.4927    0.5112    0.5229    0.5311    0.5372
    0.3883    0.4447    0.4725    0.4898    0.5020    0.5110
```

Compute the Option Sensitivities for a Vector of Strikes and a Vector of Asset Prices

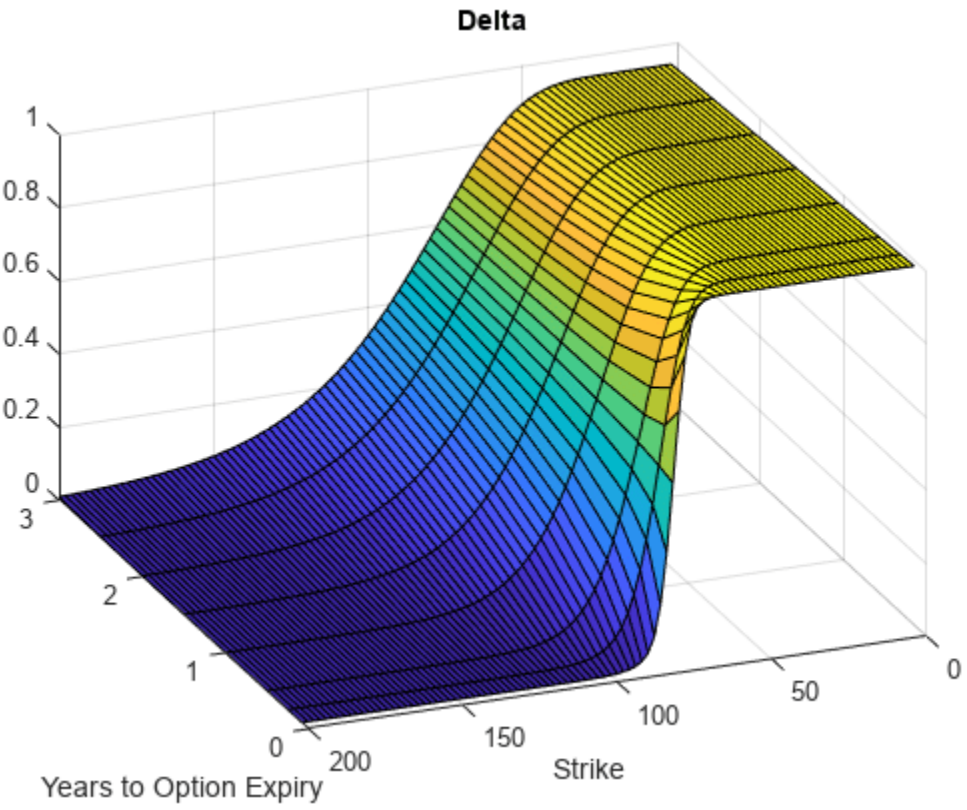
When `ExpandOutput` is `"true"`, the output can also be a `NStrikes`-by-`NAssetPrices` rectangular matrix by accepting a vector of asset prices.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12); % Single maturity
ManyAssetPrices = [70 75 80 85]; % Four asset prices
Strike = (76:2:84)'; % Five strikes

Delta = optSensByMertonNI(Rate, ManyAssetPrices, Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'OutSpec', "delta", 'ExpandOutput', true) % (5 x 4) matrix output

Delta = 5×4

    0.3796    0.5157    0.6419    0.7472
    0.3315    0.4637    0.5922    0.7043
    0.2874    0.4137    0.5422    0.6592
    0.2474    0.3664    0.4927    0.6128
    0.2117    0.3224    0.4447    0.5657
```

Plot Option Sensitivity Surfaces

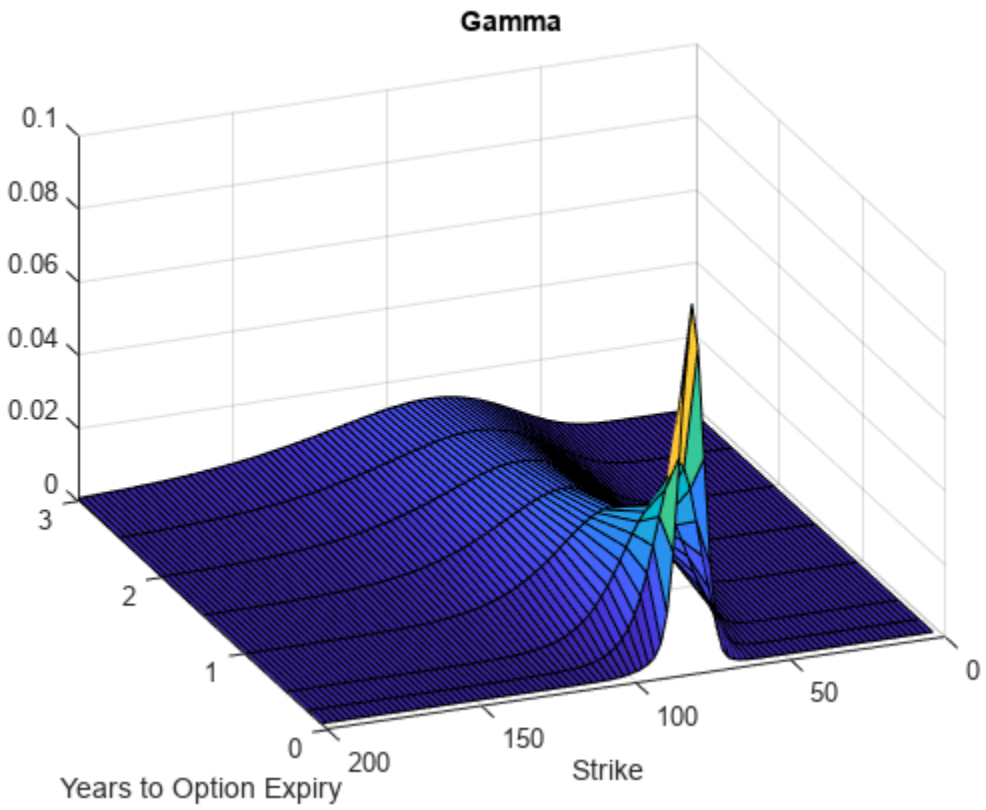
The `Strike` and `Maturity` inputs can be vectors. Set `ExpandOutput` to `"true"` to output the surfaces as `NStrikes`-by-`NMaturities` matrices.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle, 12*[1/12 0.25 (0.5:0.5:3)]');
Times = yearfrac(Settle, Maturity);
Strike = (2:2:200)';

[Delta, Gamma, Rho, Theta, Vega] = optSensByMertonNI(...
    Rate, AssetPrice,Settle, Maturity, OptSpec, Strike, ...
    Sigma, MeanJ, JumpVol, JumpFreq, 'DividendYield', DividendYield, ...
    'OutSpec', ["delta", "gamma", "rho", "theta", "vega"], ...
    'ExpandOutput', true);

[X,Y] = meshgrid(Times,Strike);

figure;
surf(X,Y,Delta);
title('Delta');
xlabel('Years to Option Expiry');
ylabel('Strike');
view(-112,34);
xlim([0 Times(end)]);
```

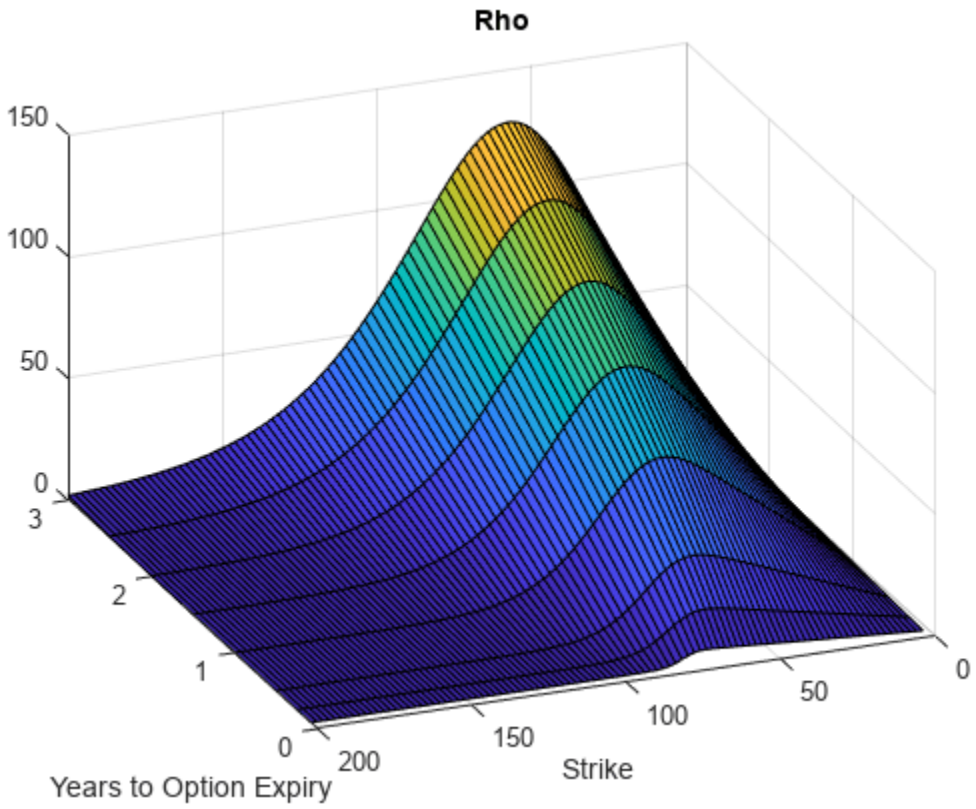



```
figure;
```

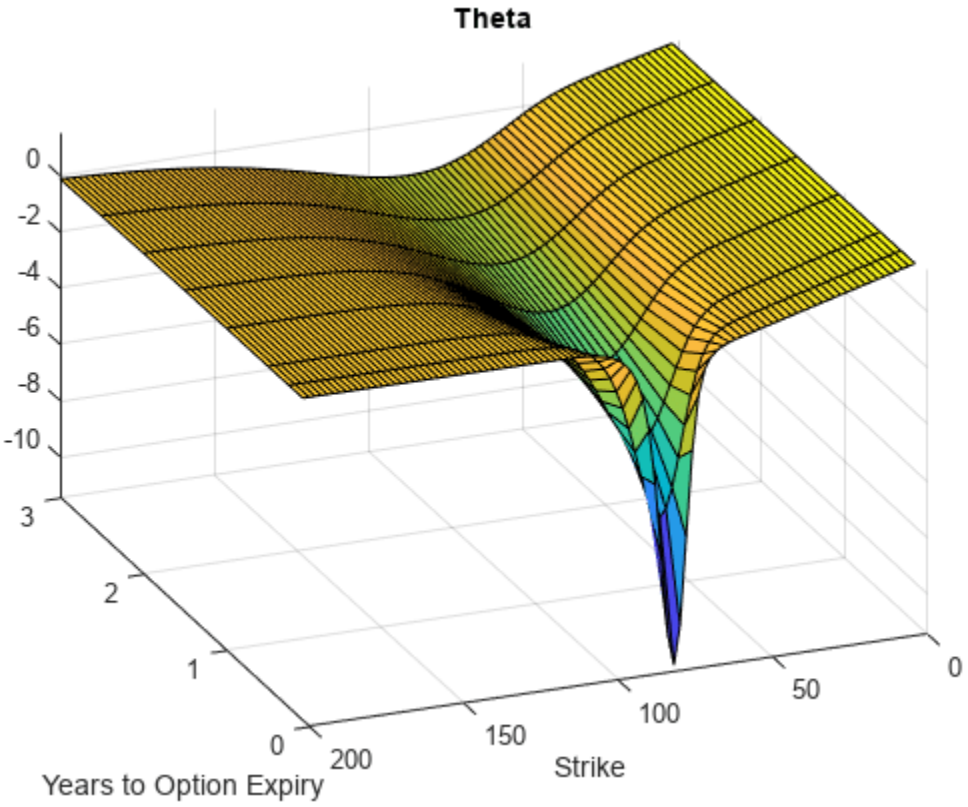
```
surf(X,Y,Gamma)
title('Gamma')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



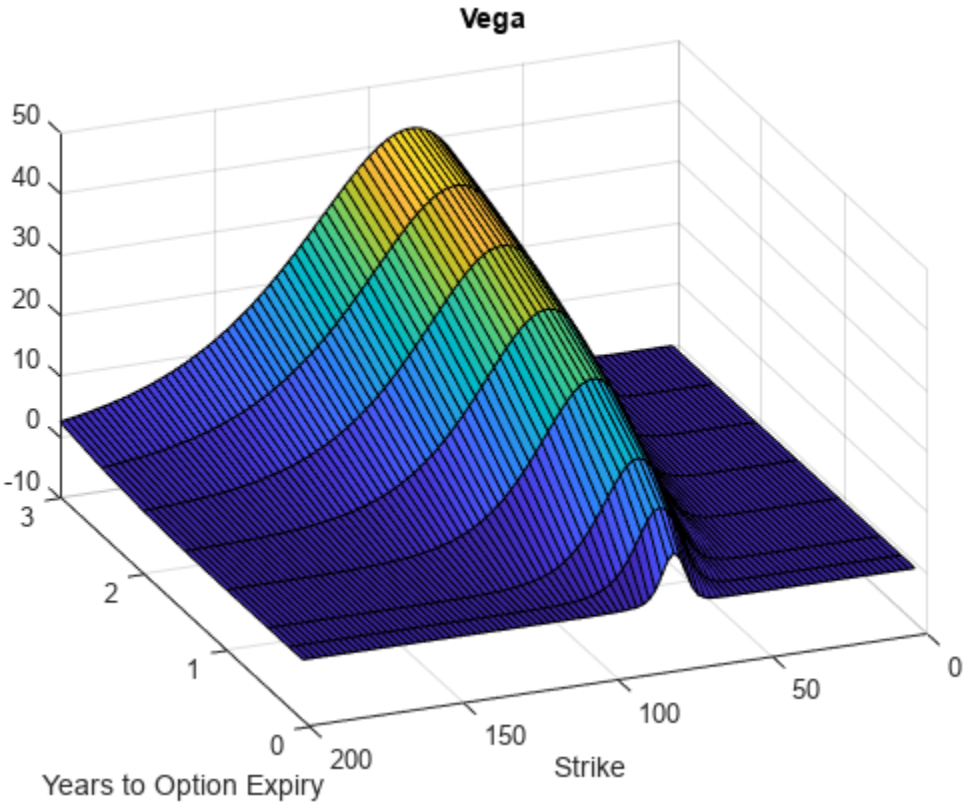

```
figure;
surf(X,Y,Rho)
title('Rho')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,Theta)
title('Theta')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```




```
figure;
surf(X,Y,Vega)
title('Vega')
xlabel('Years to Option Expiry')
ylabel('Strike')
view(-112,34);
xlim([0 Times(end)]);
```



Input Arguments

`Rate` — Continuously compounded risk-free interest rate

decimal

Continuously compounded risk-free interest rate, specified as a scalar decimal value.

Data Types: `double`

`AssetPrice` — Current underlying asset price

numeric

Current underlying asset price, specified as numeric value using a scalar or a `NINST`-by-`1` or `NColumns`-by-`1` vector.

For more information on the proper dimensions for `AssetPrice`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Settle` — Option settlement date

datetime array | string array | date character vector

Option settlement date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `optSensByMertonNI` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Settle`, see the name-value pair argument `ExpandOutput`.

`Maturity` — Option maturity date
datetime array | string array | date character vector

Option maturity date, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optSensByMertonNI` also accepts serial date numbers as inputs, but they are not recommended.

For more information on the proper dimensions for `Maturity`, see the name-value pair argument `ExpandOutput`.

`OptSpec` — Definition of option
cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option, specified as a `NINST`-by-`1` or `NColumns`-by-`1` vector using a cell array of character vectors or string arrays with values `'call'` or `'put'`.

For more information on the proper dimensions for `OptSpec`, see the name-value pair argument `ExpandOutput`.

Data Types: `cell` | `string`

`Strike` — Option strike price value
numeric

Option strike price value, specified as a `NINST`-by-`1`, `NRows`-by-`1`, `NRows`-by-`NColumns` vector of strike prices.

For more information on the proper dimensions for `Strike`, see the name-value pair argument `ExpandOutput`.

Data Types: `double`

`Sigma` — Volatility of underlying asset
numeric

Volatility of the underling asset, specified as a scalar numeric value.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
decimal

Mean of the random percentage jump size (*J*), specified as a scalar decimal value where `log`(1+*J*) is normally distributed with mean (`log`(1+`MeanJ`)-0.5*`JumpVol`^2) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log(1+`*`J`*`)`
decimal

Standard deviation of `log`(1+*J*) where `J` is the random percentage jump size, specified as a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
numeric

Annual frequency of Poisson jump process, specified as a scalar numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optSensByMertonNI(Rate,AssetPrice,Settle,Maturity,OptSpec,Strike,Sigma,MeanJ, JumpVol,JumpFreq,'Basis',7)`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a scalar using a supported value:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

DividendYield — Continuously compounded underlying asset yield
0 (default) | numeric

Continuously compounded underlying asset yield, specified as the comma-separated pair consisting of 'DividendYield' and a scalar numeric value.

Data Types: double

OutSpec — Define outputs
["price"] (default) | string array with values "price", "delta", "gamma", "vega", "rho", and "theta" | cell array of character vectors with values 'price', 'delta', 'gamma', 'vega', 'rho', and 'theta'

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and a NOUT- by-1 or a 1-by-NOUT string array or cell array of character vectors with supported values.

Note "vega" is the sensitivity with respect the initial volatility sqrt(V0).

Example: OutSpec = ["price","delta","gamma","vega","rho","theta"]

Data Types: string | cell

AbsTol — Absolute error tolerance for numerical integration
1e-10 (default) | numeric

Absolute error tolerance for numerical integration, specified as the comma-separated pair consisting of 'AbsTol' and a scalar numeric value.

Data Types: double

RelTol — Relative error tolerance for numerical integration
1e-6 (default) | numeric

Relative error tolerance for numerical integration, specified as the comma-separated pair consisting of 'RelTol' and a scalar numeric value.

Data Types: double

IntegrationRange — Numerical integration range used to approximate continuous integral over [0 Inf]
[1e-9 Inf] (default) | vector

Numerical integration range used to approximate the continuous integral over [0 Inf], specified as the comma-separated pair consisting of 'IntegrationRange' and a 1-by-2 vector representing [LowerLimit UpperLimit].

Data Types: double

Framework — Framework for computing option prices and sensitivities using numerical integration of models
"heston1993" (default) | string with values "heston1993" or "lewis2001" | character vector with values 'heston1993' or 'lewis2001'

Framework for computing option prices and sensitivities using numerical integration of models, specified as the comma-separated pair consisting of 'Framework' and a scalar string or character vector with the following values:

- `"heston1993"` or `'heston1993'` — Method used in Heston (1993)
- `"lewis2001"` or `'lewis2001'` — Method used in Lewis (2001)

Data Types: `char` | `string`

`ExpandOutput` — Flag to expand the outputs
`false` (outputs are `NINST`-by-1 vectors) (default) | logical with value of `true` or `false`

Flag to expand the outputs, specified as the comma-separated pair consisting of `'ExpandOutput'` and a logical:

- `true` — If `true`, the outputs are `NRows`-by- `NColumns` matrices. `NRows` is the number of strikes for each column and it is determined by the `Strike` input. For example, `Strike` can be a `NRows`-by-1 vector, or a `NRows`-by-`NColumns` matrix. `NColumns` is determined by the sizes of `AssetPrice`, `Settle`, `Maturity`, and `OptSpec`, which must all be either scalar or `NColumns`-by-1 vectors.
- `false` — If `false`, the outputs are `NINST`-by-1 vectors. Also, the inputs `Strike`, `AssetPrice`, `Settle`, `Maturity`, and `OptSpec` must all be either scalar or `NINST`-by-1 vectors.

Data Types: `logical`

Output Arguments

`PriceSens` — Option prices or sensitivities
numeric

Option prices or sensitivities, returned as a `NINST`-by-1, or `NRows`-by-`NColumns`, depending on `ExpandOutput`. The name-value pair argument `OutSpec` determines the types and order of the outputs.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Merton Jump Diffusion Model

The Merton jump diffusion model (Merton (1976)) is a different extension of the Black-Scholes model, where sudden asset price movements (both up and down) are modeled by adding the jump diffusion parameters with the Poisson process.

The stochastic differential equation is:

$$dS_t = (r - q - \lambda_p \mu_j) S_t dt + \sigma S_t dW_t + J S_t dP_t$$
$$\text{prob}(dP_t = 1) = \lambda_p dt$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

W_t is the Weiner process.

J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1 + J)\delta\sqrt{2\pi}} \exp\left\{ \frac{-\left[\ln(1 + J) - \left(\ln(1 + \mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2} \right\}$$

μ_J is the mean of J for (μ_J > -1).

δ is the standard deviation of $\ln(1+J)$ for ($\delta \geq 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \geq 0$).

σ is the volatility of the asset price for ($\sigma > 0$).

The characteristic function $f_{Merton76_j}(\phi)$ for $j = 1$ (asset prices measure) and $j = 2$ (risk-neutral measure) is:

$$f_{Merton76_j} = f_{BS_j} \exp\left(\lambda_p \tau \left(1 + \mu_j\right)^{m_j + \frac{1}{2}} \left[\left(1 + \mu_j\right)^{i\phi} e^{\delta^2 \left(m_j i\phi + \frac{(i\phi)^2}{2} \right)} - 1 \right] - \lambda_p \tau \mu_j i\phi \right)$$

where for $j = 1, 2$:

$$f_{BS_1}(\phi) = \frac{f_{BS_2}(\phi - i)}{f_{BS_2}(-i)}$$

$$f_{BS2}(\phi) = \exp\left(i\phi \left[\ln S_t + \left(r - q - \frac{\sigma^2}{2} \right) \tau \right] - \frac{\phi^2 \sigma^2}{2} \tau \right)$$

$$m_1 = \frac{1}{2}, m_2 = -\frac{1}{2}$$

where

ϕ is the characteristic function variable.

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

Numerical Integration Method Under Heston (1993) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Heston (1993) framework is based on the following expressions:

$$Call(K) = S_t e^{-q\tau} P_1 - K e^{-r\tau} P_2$$

$$Put(K) = Call(K) + K e^{-r\tau} - S_t e^{-q\tau}$$

$$P_j = \frac{1}{2} + \frac{1}{\pi} \int_0^{\infty} \mathrm{Re}\left[\frac{e^{-i\phi \ln(K)} f_j(\phi)}{i\phi} \right] d\phi$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity ($\tau = T - t$).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number ($i^2 = -1$).

ϕ is the characteristic function variable.

$f_j(\phi)$ is the characteristic function for $P_j (j = 1,2)$.

P_1 is the probability of $S_t > K$ under the asset price measure for the model.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

Where $j = 1,2$ so that $f_1(\phi)$ and $f_2(\phi)$ are the characteristic functions for probabilities P_1 and P_2, respectively.

This framework is chosen with the default value “`Heston1993`” for the `Framework` name-value pair argument.

Numerical Integration Method Under Lewis (2001) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Lewis (2001) framework is based on the following expressions:

$$Call(k) = S_t e^{-q\tau} - \frac{\sqrt{K}e^{-\tau t}}{\pi}\int_0^{\infty} \mathrm{Re}\left[K^{-iu} f_2\left(\phi = \left(u - \frac{i}{2}\right)\right)\frac{1}{u^2 + \frac{1}{4}}\right]du$$

$$Put(K) = Call(K) = Ke^{-\tau t} - S_t e^{-q\tau}$$

where

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number (i^2= -1).

ϕ is the characteristic function variable.

u is the characteristic function variable for integration, where $\phi = \left(u - \frac{i}{2}\right)$.

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

This framework is chosen with the value “`Lewis2001`” for the `Framework` name-value pair argument.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optSensByMertonNI` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bates, D. S. “Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options.” *The Review of Financial Studies.* Vol 9. No. 1. 1996.

[2] Cont, R. and P. Tankov. *Financial Modeling with Jump Processes*. Chapman & Hall/CRC Press, 2004.

[3] Heston, S. L. “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.” *The Review of Financial Studies.* Vol 6. No. 2. 1993.

[4] Lewis, A. L. “A Simple Option Formula for General Jump-Diffusion and Other Exponential Levy Processes.” Envision Financial Systems and OptionCity.net, 2001.

[5] Merton, R. “Option Pricing When Underlying Stock Returns are Discontinuous.” *Journal of Financial Economics.* Vol 3. 1976.

See Also

`optByHestonFFT` | `optSensByHestonFFT` | `optByHestonNI` | `optSensByHestonNI` | `optByBatesFFT` | `optSensByBatesFFT` | `optByBatesNI` | `optSensByBatesNI` | `optByMertonFFT` | `optSensByMertonFFT` | `optByMertonNI` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optembndbybdt

Price bonds with embedded options by Black-Derman-Toy interest-rate tree

Syntax

```
[Price,PriceTree] = optembndbybdt(BDTTree,CouponRate,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[Price,PriceTree] = optembndbybdt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optembndbybdt(BDTTree,CouponRate,Settle,Maturity,OptSpec, Strike,ExerciseDates)` calculates price for bonds with embedded options from a Black-Derman-Toy interest-rate tree and returns exercise probabilities in `PriceTree`.

`optembndbybdt` computes prices of vanilla bonds with embedded options, stepped coupon bonds with embedded options, amortizing bonds with embedded options, and sinking fund bonds with call embedded option. For more information, see “More About” on page 11-1450.

Note Alternatively, you can use the `OptionEmbeddedFixedBond` object to price embedded fixed-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optembndbybdt( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Callable Bond Using a BDT Interest-Rate Tree Model

Create a `BDTTree` with the following data:

```
ZeroRates = [ 0.035;0.04;0.045];
Compounding = 1;
StartDates = [datetime(2007,1,1) ; datetime(2008,1,1) ; datetime(2009,1,1)];
EndDates   = [datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1)];
ValuationDate = 'jan-1-2007';
```

Create a `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [3x1 double]
            Rates: [3x1 double]
```

```
         EndTimes: [3x1 double]
       StartTimes: [3x1 double]
         EndDates: [3x1 double]
       StartDates: 733043
    ValuationDate: 733043
            Basis: 0
     EndMonthRule: 1
```

Create a `VolSpec`.

```
Volatility = 0.10 * ones (3,1);
VolSpec = bdtvolspec(ValuationDate, EndDates, Volatility)
```

```
VolSpec = struct with fields:
             FinObj: 'BDTVolSpec'
      ValuationDate: 733043
           VolDates: [3x1 double]
           VolCurve: [3x1 double]
    VolInterpMethod: 'linear'
```

Create a `TimeSpec`.

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

Build the `BDTTree`.

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

```
BDTTree = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2]
        dObs: [733043 733408 733774]
        TFwd: {[3x1 double]  [2x1 double]  [2]}
      CFlowT: {[3x1 double]  [2x1 double]  [3]}
     FwdTree: {[1.0350]  [1.0406 1.0495]  [1.0447 1.0546 1.0667]}
```

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 = datetime(2007,1,1);
BondMaturity   = datetime(2010,1,1);
CouponRate = 0.0525;
Period = 1;
OptSpec = 'call';
Strike = [100];
ExerciseDates = [datetime(2008,1,1) datetime(2010,1,1)];
AmericanOpt = 1;

PriceCallBond = optembndbybdt(BDTTree, CouponRate, BondSettlement, BondMaturity,...
OptSpec, Strike, ExerciseDates, 'Period', 1,'AmericanOp', 1)
```

```
PriceCallBond = 101.4750
```

Obtain Callable Bond Exercise Information Using a BDT Interest-Rate Tree Model

Create a `BDTTree` with the following data:

```
ZeroRates = [ 0.025;0.03;0.045];
Compounding = 1;
StartDates = [datetime(2007,1,1) ; datetime(2008,1,1) ; datetime(2009,1,1)];
EndDates   = [datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1)];
ValuationDate = datetime(2007,1,1);
```

Create a `RateSpec`.

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

Build the BDT tree with the following data.

```
Volatility = 0.10 * ones (3,1);
VolSpec = bdtvolspec(ValuationDate, EndDates, Volatility)
```

```
VolSpec = struct with fields:
             FinObj: 'BDTVolSpec'
      ValuationDate: 733043
           VolDates: [3x1 double]
           VolCurve: [3x1 double]
    VolInterpMethod: 'linear'
```

```
TimeSpec = bdttimespec(ValuationDate, EndDates, Compounding);
BDTTree = bdttree(VolSpec, RateSpec, TimeSpec)
```

```
BDTTree = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2]
        dObs: [733043 733408 733774]
        TFwd: {[3x1 double]  [2x1 double]  [2]}
      CFlowT: {[3x1 double]  [2x1 double]  [3]}
     FwdTree: {[1.0250]  [1.0315 1.0385]  [1.0614 1.0750 1.0917]}
```

Define the callable bond instruments.

```
Settle = datetime(2007,1,1);
Maturity = [datetime(2008,1,1) datetime(2010,1,1)];
CouponRate = {{datetime(2008,1,1) .0425;datetime(2010,1,1) .0450}};
OptSpec='call';
Strike= [86;90];
ExerciseDates= [datetime(2008,1,1) ; datetime(2010,1,1)];
```

Price the instruments.

```
[Price, PriceTree]= optembndbybdt(BDTTree, CouponRate,  Settle, Maturity, OptSpec, Strike,...
ExerciseDates, 'Period', 1,'AmericanOp', 1)
```

```
Price = 2×1

    86
    90


PriceTree = struct with fields:
                FinObj: 'BDTPriceTree'
                  tObs: [0 1 2 3]
                 PTree: {1x4 cell}
              ProbTree: {[1]  [0.5000 0.5000]  [0.2500 0.5000 0.2500]  [1x3 double]}
                 ExTree: {1x4 cell}
            ExProbTree: {1x4 cell}
    ExProbsByTreeLevel: [2x4 double]
```

Examine the output `PriceTree.ExTree`. `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it is not.

```
PriceTree.ExTree{4}
```

```
ans = 2×3

     0     0     0
     1     1     1
```

There is no exercise for instrument 1 and instrument 2 is exercised at all nodes.

```
PriceTree.ExTree{3}
```

```
ans = 2×3

     0     0     0
     0     0     0
```

There is no exercise for instrument 1 or instrument 2.

```
PriceTree.ExTree{2}
```

```
ans = 2×2

     1     1
     1     0
```

There is exercise for instrument 1 at all nodes and instrument 2 is exercised at some nodes.

Next view the probability of reaching each node from the root node using `PriceTree.ProbTree`.

```
PriceTree.ProbTree{2}
```

```
ans = 1×2

    0.5000    0.5000
```

```
PriceTree.ProbTree{3}
```

```
ans = 1×3

    0.2500    0.5000    0.2500
```

```
PriceTree.ProbTree{4}
```

```
ans = 1×3

    0.2500    0.5000    0.2500
```

Then view the exercise probabilities using `PriceTree.ExProbTree`. `PriceTree.ExProbTree` contains the exercise probabilities. Each element in the cell array is an array containing 0's where there is no exercise, or the probability of reaching that node where exercise happens.

```
PriceTree.ExProbTree{4}
```

```
ans = 2×3

         0         0         0
    0.2500    0.5000    0.2500
```

```
PriceTree.ExProbTree{3}
```

```
ans = 2×3

     0     0     0
     0     0     0
```

```
PriceTree.ExProbTree{2}
```

```
ans = 2×2

    0.5000    0.5000
    0.5000         0
```

View the exercise probabilities at each tree level using `PriceTree.ExProbsByTreeLevel`. `PriceTree.ExProbsByTreeLevel` is an array with each row holding the exercise probability for a given option at each tree observation time.

```
PriceTree.ExProbsByTreeLevel
```

```
ans = 2×4

    1.0000    1.0000         0         0
    1.0000    0.5000         0    1.0000
```

Price Single Stepped Callable Bonds Using a BDT Interest-Rate Tree Model

Price the following single stepped callable bonds using the following data: The data for the interest rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2010,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
Compounding = 1;

% Create RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);

% Instrument
Settle = datetime(2010,1,1);
Maturity =[datetime(2013,1,1) ; datetime(2014,1,1)];
CouponRate = {{'01-Jan-2012' .0425;'01-Jan-2014' .0750}};
OptSpec='call';
Strike=100;
ExerciseDates= datetime(2012,1,1);  %Callable in two years

% Build the tree
% Assume the volatility to be 10%
Sigma = 0.1;
BDTTimeSpec = bdttimespec(ValuationDate, EndDates, Compounding);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVolSpec, RS, BDTTimeSpec);

% The first row corresponds to the price of the callable bond with maturity
% of three years. The second row corresponds to the price of the callable bond
% with maturity of four years.
PBDT=  optembndbybdt(BDTT, CouponRate, Settle, Maturity ,OptSpec, Strike,...
ExerciseDates, 'Period', 1)
```

```
PBDT = 2×1

  100.0945
  100.0297
```

Price a Sinking Fund Bond Using a BDT Interest-Rate Tree Model

A corporation issues a three year bond with a sinking fund obligation requiring the company to sink 1/3 of face value after the first year and 1/3 after the second year. The company has the option to buy the bonds in the market or call them at $98. The following data describes the details needed for pricing the sinking fund bond:

```
Rates = [0.1;0.1;0.1;0.1];
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
Compounding = 1;

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

% Build the BDT tree
```

```
% Assume the volatility to be 10%
Sigma = 0.1;
BDTTimeSpec = bdttimespec(ValuationDate, EndDates);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);

% Instrument
% The bond has a coupon rate of 9%, a period of one year and matures in
% 1-Jan-2014. Face decreases 1/3 after the first year and 1/3 after the
% second year.
CouponRate = 0.09;
Settle = datetime(2011,1,1);
Maturity = datetime(2014,1,1);
Period = 1;
Face = { ...
            {datetime(2012,1,1) 100; ...
             datetime(2013,1,1) 66.6666; ...
             datetime(2014,1,1) 33.3333};
        };

% Option provision
OptSpec = 'call';
Strike = [98 98];
ExerciseDates = [datetime(2012,1,1) , datetime(2013,1,1)];

% Price of non-sinking fund bond.
PNSF = bondbybdt(BDTT, CouponRate, Settle, Maturity, Period)
```

```
PNSF = 97.5131
```

Price of the bond with the option sinking provision.

```
PriceSF = optembndbybdt(BDTT, CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates,'Period', Period, 'Face', Face)
```

```
PriceSF = 96.8364
```

Price an Amortizing Callable Bond Using a BDT Interest-Rate Tree Model

This example shows how to price an amortizing callable bond and a vanilla callable bond using a BDT lattice model.

Create a `RateSpec`.

```
Rates = [0.025;0.028;0.030;0.031];
ValuationDate = datetime(2018,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2019,1,1) ; datetime(2020,1,1) ; datetime(2021,1,1) ; datetime(2022,1,1)];
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates',...
StartDates, 'EndDates', EndDates, 'Rates', Rates, 'Compounding', Compounding);
```

Build a BDT tree and assume a volatility of 5%.

```
Sigma = 0.05;
BDTTimeSpec = bdttimespec(ValuationDate, EndDates);
```

```
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Define the callable bond.

```
CouponRate = 0.05;
Settle = datetime(2018,1,1);
Maturity = datetime(2021,1,1);
Period = 1;

    Face = {
                {datetime(2019,1,1) 100;
                 datetime(2020,1,1) 70; ...
                 datetime(2021,1,1) 50};
                };

OptSpec = 'call';
Strike = [97 95 93];
ExerciseDates = [datetime(2019,1,1) datetime(2020,1,1) datetime(2021,1,1)];
```

Price a callable amortizing bond using the BDT tree.

```
BondType = 'amortizing';
Pcallbonds = optembndbybdt(BDTT, CouponRate, Settle, Maturity, OptSpec, Strike, ExerciseDates, '
```

```
Pcallbonds = 99.5122
```

Input Arguments

`BDTTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bdttree`.

Data Types: `struct`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optembndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char`

`Strike` — Option strike price values
nonnegative integer

Option strike price value, specified as a `NINST`-by-`1` or `NINST`-by-`NSTRIKES` depending on the type of option:

- 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.

Data Types: `double`

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1`, `NINST`-by-`2`, or `NINST`-by-`NSTRIKES` vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a `NINST`-by-`1` vector of dates. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a `NINST`-by-`NSTRIKES` vector of dates.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-`1` vector, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `optembndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optembndbybdt(BDTTree,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates,'Period',1,'AmericanOp',1)`

`AmericanOpt` — Option type

`0` European/Bermuda (default) | integer with values `0` or `1`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`Period` — Coupons per year

`2` per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis

`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbybdt` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | `NINST`-by-`1` vector | `NINST`-by-`1` cell array

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector or a `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated face value. The date indicates the last day that the face value is valid.

Note Instruments without a `Face` schedule are treated as either vanilla bonds or stepped coupon bonds with embedded options.

Data Types: `double`

`BondType` — Type of underlying bond
`'vanilla'` for scalar `Face` values, `'callablesinking'` for scheduled `Face` values (default) | cell array of character vectors with values `'vanilla'`,`'amortizing'`, or `'callablesinking'` | string array with values `"vanilla"`, `"amortizing"`, or `"callablesinking"`

Type of underlying bond, specified as the comma-separated pair consisting of `'BondType'` and a `NINST`-by-`1` cell array of character vectors or string array specifying if the underlying is a vanilla bond, an amortizing bond, or a callable sinking fund bond. The supported types are:

- `'vanilla'` is a standard callable or puttable bond with a scalar `Face` value and a single coupon or stepped coupons.
- `'callablesinking'` is a bond with a schedule of `Face` values and a sinking fund call provision with a single or stepped coupons.
- `'amortizing'` is an amortizing callable or puttable bond with a schedule of `Face` values with single or stepped coupons.

Data Types: `char` | `string`

`Options` — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of embedded option at time `0`

matrix

Expected price of the embedded option at time `0`, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure containing trees of vectors of instrument prices and exercise probabilities for each node

structure

Structure containing trees of vectors of instrument prices, a vector of observation times for each node, and exercise probabilities. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it isn't.
- `PriceTree.ProbTree` contains the probability of reaching each node from root node.
- `PriceTree.ExProbTree` contains the exercise probabilities. Each element in the cell array is an array containing `0`'s where there is no exercise, or the probability of reaching that node where exercise happens.
- `PriceTree.ExProbsByTreeLevel` is an array with each row holding the exercise probability for a given option at each tree observation time.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2008a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optembndbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdtprice` | `bdttree` | `cfamounts` | `instoptembnd` | `OptionEmbeddedFixedBond`

Topics

“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond with Embedded Options” on page 2-7
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optembndbybk

Price bonds with embedded options by Black-Karasinski interest-rate tree

Syntax

```
[Price,PriceTree] = optembndbybk(BKTree,CouponRate,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[Price,PriceTree] = optembndbybk( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optembndbybk(BKTree,CouponRate,Settle,Maturity,OptSpec, Strike,ExerciseDates)` calculates price for bonds with embedded options from a Black-Karasinski interest-rate tree and returns exercise probabilities in `PriceTree`.

`optembndbybk` computes prices of vanilla bonds with embedded options, stepped coupon bonds with embedded options, amortizing bonds with embedded options, and sinking fund bonds with call embedded option. For more information, see “More About” on page 11-1463.

Note Alternatively, you can use the `OptionEmbeddedFixedBond` object to price embedded fixed-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optembndbybk( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Callable Bond Using a BK Interest-Rate Tree Model

Create a `BKTree` with the following data:

```
ZeroRates = [ 0.035;0.04;0.045];
Compounding = 1;
StartDates = [datetime(2007,1,1) ; datetime(2008,1,1) ; datetime(2009,1,1)];
EndDates   = [datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1)];
ValuationDate = datetime(2007,1,1);
```

Create a `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [3x1 double]
            Rates: [3x1 double]
```

```
         EndTimes: [3x1 double]
       StartTimes: [3x1 double]
         EndDates: [3x1 double]
       StartDates: 733043
    ValuationDate: 733043
            Basis: 0
     EndMonthRule: 1
```

Create a `VolSpec`.

```
VolDates = [datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2010,1,1);
AlphaCurve = 0.1;
BKVolSpec = bkvolspec(ValuationDate, VolDates, VolCurve, AlphaDates, AlphaCurve)
```

```
BKVolSpec = struct with fields:
             FinObj: 'BKVolSpec'
      ValuationDate: 733043
           VolDates: [3x1 double]
           VolCurve: [3x1 double]
         AlphaCurve: 0.1000
         AlphaDates: 734139
    VolInterpMethod: 'linear'
```

Create a `TimeSpec`.

```
BKTimeSpec = bktimespec(ValuationDate, EndDates, Compounding);
```

Build the `BKTree`.

```
BKTree = bktree(BKVolSpec, RateSpec, BKTimeSpec)
```

```
BKTree = struct with fields:
      FinObj: 'BKFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2]
        dObs: [733043 733408 733774]
      CFlowT: {[3x1 double]  [2x1 double]  [3]}
       Probs: {[3x1 double]  [3x3 double]}
     Connect: {[2]  [2 3 4]}
     FwdTree: {[1.0350]  [1.0458 1.0450 1.0442]  [1.0571 1.0561 1.0551 ... ]}
```

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 = datetime(2007,1,1);
BondMaturity   = datetime(2010,1,1);
CouponRate = 0.0525;
Period = 1;
OptSpec = 'put';
Strike = [100];
ExerciseDates = [datetime(2008,1,1) datetime(2010,1,1)];
AmericanOpt = 1;
```

```
PricePutBondBK = optembndbybk(BKTree, CouponRate, BondSettlement, BondMaturity,...
OptSpec, Strike, ExerciseDates,'Period', 1, 'AmericanOpt', 1)

PricePutBondBK = 102.3820
```

Obtain Callable Bond Exercise Information Using a BK Interest-Rate Tree Model

Create a `BKTree` with the following data:

```
ZeroRates = [ 0.025;0.03;0.045];
Compounding = 1;
StartDates = [datetime(2007,1,1); datetime(2008,1,1) ; datetime(2009,1,1)];
EndDates   = [datetime(2008,1,1); datetime(2009,1,1) ; datetime(2010,1,1)];
ValuationDate = datetime(2007,1,1);
```

Create a `RateSpec`.

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

Build the BK tree with the following data.

```
VolDates = [datetime(2008,1,1); datetime(2009,1,1) ; datetime(2010,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2010,1,1);
AlphaCurve = 0.1;
BKVolSpec = bkvolspec(ValuationDate, VolDates, VolCurve, AlphaDates, AlphaCurve);

BKTimeSpec = bktimespec(ValuationDate, EndDates, Compounding);
BKTree = bktree(BKVolSpec, RateSpec, BKTimeSpec);
```

Define the callable bond instruments.

```
Settle = datetime(2007,1,1);
Maturity = [datetime(2008,1,1) ; datetime(2010,1,1)];
CouponRate = {{datetime(2008,1,1) .0315;datetime(2010,1,1) .0350}};
OptSpec='call';
Strike= [86;90];
ExerciseDates= [datetime(2008,1,1) ; datetime(2010,1,1)];
```

Price the instruments.

```
[Price, PriceTree]= optembndbybk(BKTree, CouponRate,  Settle, Maturity, OptSpec, Strike,...
ExerciseDates, 'Period', 1,'AmericanOp', 1)

Price = 2×1

   86.0000
   88.3060


PriceTree = struct with fields:
                FinObj: 'BKPriceTree'
                  tObs: [0 1 2 3]
                 PTree: {1x4 cell}
```

```
            ProbTree: {1x4 cell}
              ExTree: {1x4 cell}
           ExProbTree: {1x4 cell}
    ExProbsByTreeLevel: [2x4 double]
              Connect: {[2]  [2 3 4]}
```

Examine the output `PriceTree.ExTree`. `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it is not.

```
PriceTree.ExTree{4}
```

```
ans = 2×5

     0     0     0     0     0
     1     1     1     1     1
```

There is no exercise for instrument 1 and instrument 2 is exercised at all nodes.

```
PriceTree.ExTree{3}
```

```
ans = 2×5

     0     0     0     0     0
     0     0     0     0     0
```

There is no exercise for instrument 1 or instrument 2.

```
PriceTree.ExTree{2}
```

```
ans = 2×3

     1     1     1
     0     0     0
```

There is exercise for instrument 1 at all nodes and instrument 2 is not exercised.

Next view the probability of reaching each node from the root node using `PriceTree.ProbTree`.

```
PriceTree.ProbTree{2}
```

```
ans = 1×3

    0.1667    0.6667    0.1667
```

```
PriceTree.ProbTree{3}
```

```
ans = 1×5

    0.0203    0.2206    0.5183    0.2206    0.0203
```

```
PriceTree.ProbTree{4}
```

```
ans = 1×5
```

```
    0.0203    0.2206    0.5183    0.2206    0.0203
```

Then view the exercise probabilities using `PriceTree.ExProbTree`. `PriceTree.ExProbTree` contains the exercise probabilities. Each element in the cell array is an array containing 0's where there is no exercise, or the probability of reaching that node where exercise happens.

```
PriceTree.ExProbTree{4}
```

```
ans = 2×5

         0         0         0         0         0
    0.0203    0.2206    0.5183    0.2206    0.0203
```

```
PriceTree.ExProbTree{3}
```

```
ans = 2×5

     0     0     0     0     0
     0     0     0     0     0
```

```
PriceTree.ExProbTree{2}
```

```
ans = 2×3

    0.1667    0.6667    0.1667
         0         0         0
```

View the exercise probabilities at each tree level using `PriceTree.ExProbsByTreeLevel`. `PriceTree.ExProbsByTreeLevel` is an array with each row holding the exercise probability for a given option at each tree observation time.

```
PriceTree.ExProbsByTreeLevel
```

```
ans = 2×4

    1.0000    1.0000         0         0
         0         0         0    1.0000
```

Price Single Stepped Callable Bonds Using a BK Interest-Rate Tree Model

Price the following single stepped callable bonds using the following data: The data for the interest rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2010,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
Compounding = 1;

% Create RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
```

```
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);

% Instrument
Settle = datetime(2010,1,1);
Maturity = [datetime(2013,1,1) ; datetime(2014,1,1)];
CouponRate = {{datetime(2012,1,1) .0425;datetime(2014,1,1) .0750}};
OptSpec='call';
Strike=100;
ExerciseDates= datetime(2012,1,1);  %Callable in two years

% Build the tree with the following data
VolDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ;datetime(2014,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2014,1,1);
AlphaCurve = 0.1;


BKVolSpec = bkvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RS.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RS, BKTimeSpec);

% The first row corresponds to the price of the callable bond with maturity
% of three years. The second row corresponds to the price of the callable bond
% with maturity of four years.
PBK= optembndbybk(BKT, CouponRate,  Settle, Maturity ,OptSpec, Strike,...
ExerciseDates, 'Period', 1)

PBK = 2×1

  100.0945
  100.0945
```

Price an Amortizing Callable Bond Using a BK Interest-Rate Tree Model

This example shows how to price an amortizing callable bond and a vanilla callable bond using a BK lattice model.

Create a RateSpec.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);
```

Build a BK tree.

```
VolDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2016,1,1);
AlphaCurve = 0.1;
```

```
BKVolSpec = bkvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RS.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RS, BKTimeSpec);
```

Define the callable bond.

```
CouponRate = 0.05;
Settle = datetime(2012,1,1);
Maturity =  datetime(2016,1,1);
Period = 1;

    Face = {
                {datetime(2014,1,1) 100;
                 datetime(2015,1,1) 70;
                 datetime(2016,1,1) 50};
                };

OptSpec = 'call';
Strike = [97 95 93];
ExerciseDates = [datetime(2014,1,1) datetime(2015,1,1) datetime(2016,1,1)];
```

Price a callable amortizing bond using the BDT tree.

```
BondType = 'amortizing';
Pcallbonds = optembndbybk(BKT, CouponRate,  Settle, Maturity ,OptSpec, Strike,...
ExerciseDates, 'Period', 1,'Face',Face,'BondType', BondType)

Pcallbonds = 98.7475
```

Input Arguments

BKTree — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bktree`.

Data Types: `struct`

CouponRate — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

Settle — Settlement date

datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optembndbybk` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbybk` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char`

`Strike` — Option strike price values
nonnegative integer

Option strike price value, specified as a `NINST`-by-`1` or `NINST`-by-`NSTRIKES` depending on the type of option:

- 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.

Data Types: `double`

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1`, `NINST`-by-`2`, or `NINST`-by-`NSTRIKES` vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a `NINST`-by-`1` vector of dates. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a `NINST`-by-`NSTRIKES` vector of dates.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-`1` vector, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

To support existing code, `optembndbybk` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optembndbybk(BKTree,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates,'Period',1,'AmericanOp',1)`

`AmericanOpt` — Option type
`0` European/Bermuda (default) | integer with values `0` or `1`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`Period` — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbybk` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbybk` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbybk` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

StartDate — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of 'StartDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optembndbybk also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify StartDate, the effective start date is the Settle date.

Face — Face value
100 (default) | NINST-by-1 vector | NINST-by-1 cell array

Face or par value, specified as the comma-separated pair consisting of 'Face' and a NINST-by-1 vector or a NINST-by-1 cell array where each element is a NumDates-by-2 cell array where the first column is dates and the second column is associated face value. The date indicates the last day that the face value is valid.

Note Instruments without a Face schedule are treated as either vanilla bonds or stepped coupon bonds with embedded options.

Data Types: double

BondType — Type of underlying bond
'vanilla' for scalar Face values, 'callablesinking' for scheduled Face values (default) | cell array of character vectors with values 'vanilla','amortizing', or 'callablesinking' | string array with values "vanilla", "amortizing", or "callablesinking"

Type of underlying bond, specified as the comma-separated pair consisting of 'BondType' and a NINST-by-1 cell array of character vectors or string array specifying if the underlying is a vanilla bond, an amortizing bond, or a callable sinking fund bond. The supported types are:

- 'vanilla' is a standard callable or puttable bond with a scalar Face value and a single coupon or stepped coupons.
- 'callablesinking' is a bond with a schedule of Face values and a sinking fund call provision with a single or stepped coupons.
- 'amortizing' is an amortizing callable or puttable bond with a schedule of Face values with single or stepped coupons.

Data Types: char | string

Options — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of 'Options' and a structure that is created with derivset.

Data Types: struct

Output Arguments

`Price` — Expected prices of embedded option at time `0`

matrix

Expected price of the embedded option at time `0`, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure containing trees of vectors of instrument prices and exercise probabilities for each node

structure

Structure containing trees of vectors of instrument prices, a vector of observation times for each node, and exercise probabilities. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it isn't.
- `PriceTree.ProbTree` contains the probability of reaching each node from root node.
- `PriceTree.ExProbTree` contains the exercise probabilities. Each element in the cell array is an array containing `0`'s where there is no exercise, or the probability of reaching that node where exercise happens.
- `PriceTree.ExProbsByTreeLevel` is an array with each row holding the exercise probability for a given option at each tree observation time.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2008a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optembndbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bkprice` | `cfamounts` | `bktree` | `instoptembnd` | `OptionEmbeddedFixedBond`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond with Embedded Options” on page 2-7
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optembndbycir

Price bonds with embedded options by Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = optembndbycir(CIRTree,CouponRate,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[Price,PriceTree] = optembndbycir( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optembndbycir(CIRTree,CouponRate,Settle,Maturity,OptSpec, Strike,ExerciseDates)` calculates price for bonds with embedded options from a Cox-Ingersoll-Ross (CIR) interest-rate tree and returns exercise probabilities in `PriceTree`.

`optembndbycir` computes prices of vanilla bonds with embedded options, stepped coupon bonds with embedded options, amortizing bonds with embedded options, and sinking fund bonds with call embedded option using a CIR++ model with the Nawalka-Beliaeva (NB) approach. For more information, see “More About” on page 11-1474.

`[Price,PriceTree] = optembndbycir( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Callable Bond Using a CIR Interest-Rate Tree

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.025; 0.032; 0.037; 0.042];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = datetime(2017,1,1);
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',EndD
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Maturity = '01-Jan-2018';
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);
CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
```

```
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 0.2500 0.5000 0.7500]
        dObs: [736696 736787 736878 736969]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Define the bond with embedded option instrument.

```
BondSettlement = datetime(2017,1,1);
BondMaturity   = datetime(2020,1,1);
CouponRate = 0.035;
Period = 1;
OptSpec = 'put';
Strike = 100;
ExerciseDates = [datetime(2018,1,1) datetime(2019,1,1)];
```

Price the bond.

```
[Price,PriceTree] = optembndbycir(CIRT,CouponRate,BondSettlement,BondMaturity,OptSpec,...
Strike,ExerciseDates,'AmericanOpt',1,'Period',1)
```

```
Price = 103.3099
```

```
PriceTree = struct with fields:
                FinObj: 'CIRPriceTree'
                  tObs: [0 0.2500 0.5000 0.7500 1]
                 PTree: {1x5 cell}
               ProbTree: {1x5 cell}
                 ExTree: {[0]  [0 0 0]  [0 0 0 0 0]  [0 0 0 0 0 0 0]  [1 1 1 0 0 0 0]}
            ExProbTree: {[0]  [0 0 0]  [0 0 0 0 0]  [0 0 0 0 0 0 0]  [0.0222 ... ]}
    ExProbsByTreeLevel: [0 0 0 0 0.3089]
               Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Obtain Callable Bond Exercise Information Using a CIR Interest-Rate Tree Model

Create a CIR`Tree` with the following data:

```
Rates = [0.025; 0.027; 0.028; 0.03];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = datetime(2017,1,1);
EndDates = Dates(2:end)';
Compounding = 1;
```

Create a `RateSpec`.

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

Build the CIR tree with the following data.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
```

```
Theta = 0.02;
Sigma = 0.1;
Maturity = datetime(2018,1,1);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);
CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec);
```

Define the callable bond instruments.

```
Settle = datetime(2017,1,1);
Maturity = [datetime(2018,1,1) ; datetime(2020,1,1)];
CouponRate = {{datetime(2019,1,1) .0325;datetime(2020,1,1) .0375}};
OptSpec='call';
Strike= [100;110];
ExerciseDates= [datetime(2018,1,1) ; datetime(2020,1,1)];
```

Price the instruments.

```
[Price, PriceTree]= optembndbycir(CIRT, CouponRate,  Settle, Maturity, OptSpec, Strike,...
ExerciseDates, 'Period', 1,'AmericanOpt',1)
```

```
Price = 2×1

   98.1718
  102.6458


PriceTree = struct with fields:
                FinObj: 'CIRPriceTree'
                  tObs: [0 0.2500 0.5000 0.7500 1]
                 PTree: {1x5 cell}
              ProbTree: {1x5 cell}
                 ExTree: {1x5 cell}
            ExProbTree: {1x5 cell}
    ExProbsByTreeLevel: [2x5 double]
               Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Examine the output `PriceTree.ExTree`. `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it is not.

```
PriceTree.ExTree{4}
```

```
ans = 2×7

     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
```

There is no exercise for instrument 1 or instrument 2.

```
PriceTree.ExTree{3}
```

```
ans = 2×5

     0     0     0     0     0
     0     0     0     0     0
```

There is no exercise for instrument 1 or instrument 2.

```
PriceTree.ExTree{2}
```

```
ans = 2×3

     0     0     0
     0     0     0
```

There is exercise for instrument 1 or instrument 2.

Next view the probability of reaching each node from the root node using `PriceTree.ProbTree`.

```
PriceTree.ProbTree{2}
```

```
ans = 1×3

    0.2794    0.3750    0.3456
```

```
PriceTree.ProbTree{3}
```

```
ans = 1×5

    0.0786    0.2095    0.3318    0.2592    0.1209
```

```
PriceTree.ProbTree{4}
```

```
ans = 1×7

    0.0222    0.0885    0.1982    0.2678    0.2442    0.1360    0.0432
```

Then view the exercise probabilities using `PriceTree.ExProbTree`. `PriceTree.ExProbTree` contains the exercise probabilities. Each element in the cell array is an array containing `0`'s where there is no exercise, or the probability of reaching that node where exercise happens.

```
PriceTree.ExProbTree{4}
```

```
ans = 2×7

     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
```

```
PriceTree.ExProbTree{3}
```

```
ans = 2×5

     0     0     0     0     0
     0     0     0     0     0
```

```
PriceTree.ExProbTree{2}
```

```
ans = 2×3

     0     0     0
```

```
     0     0     0
```

View the exercise probabilities at each tree level using `PriceTree.ExProbsByTreeLevel`. `PriceTree.ExProbsByTreeLevel` is an array with each row holding the exercise probability for a given option at each tree observation time.

```
PriceTree.ExProbsByTreeLevel
```

```
ans = 2×5

     0     0     0     0     0
     0     0     0     0     0
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `cirtree`.

Data Types: `struct`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optembndbycir` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbycir` also accepts serial date numbers as inputs, but they are not recommended.

OptSpec — Definition of option
character vector with value 'call' or 'put' | cell array of character vectors with values 'call' or 'put' | string array with value "call" or "put"

Definition of option, specified as a NINST-by-1 cell array of character vectors or string arrays with a value of 'call' or 'put'.

Data Types: char | cell | string

Strike — Option strike price values
nonnegative integer

Option strike price value, specified as a NINST-by-1 or NINST-by-NSTRIKES depending on the type of option:

- European option — NINST-by-1 vector of strike price values.
- Bermuda option — NINST by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- American option — NINST-by-1 vector of strike price values for each option.

Data Types: double

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a NINST-by-1, NINST-by-2, or NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a NINST-by-1 vector, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

To support existing code, optembndbycir also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree] = optembndbycir(BDTTree,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates,'Period',1,'AmericanOpt',1)

AmericanOpt — Option type
0 European/Bermuda (default) | integer with values 0 or 1

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and an `NINST`-by-`1` positive integer flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`Period` — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.

- 1 = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: double

IssueDate — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of 'IssueDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optembndbycir also accepts serial date numbers as inputs, but they are not recommended.

FirstCouponDate — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optembndbycir also accepts serial date numbers as inputs, but they are not recommended.

When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

LastCouponDate — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optembndbycir also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

StartDate — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of 'StartDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optembndbycir also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify StartDate, the effective start date is the Settle date.

Face — Face value
100 (default) | NINST-by-1 vector | NINST-by-1 cell array

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector or a `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated face value. The date indicates the last day that the face value is valid.

Note Instruments without a `Face` schedule are treated as either vanilla bonds or stepped coupon bonds with embedded options.

Data Types: `double`

BondType — Type of underlying bond
`'vanilla'` for scalar `Face` values, `'callablesinking'` for scheduled `Face` values (default) | cell array of character vectors with values `'vanilla'`,`'amortizing'`, or `'callablesinking'` | string array with values `"vanilla"`, `"amortizing"`, or `"callablesinking"`

Type of underlying bond, specified as the comma-separated pair consisting of `'BondType'` and a `NINST`-by-`1` cell array of character vectors or string array specifying if the underlying is a vanilla bond, an amortizing bond, or a callable sinking fund bond. The supported types are:

- `'vanilla'` is a standard callable or puttable bond with a scalar `Face` value and a single coupon or stepped coupons.
- `'callablesinking'` is a bond with a schedule of `Face` values and a sinking fund call provision with a single or stepped coupons.
- `'amortizing'` is an amortizing callable or puttable bond with a schedule of `Face` values with single or stepped coupons.

Data Types: `char` | `string`

Output Arguments

`Price` — Expected prices of embedded option at time `0`
matrix

Expected price of the embedded option at time `0`, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure containing trees of vectors of instrument prices and exercise probabilities for each node
structure

Structure containing trees of vectors of instrument prices, a vector of observation times for each node, and exercise probabilities. Values are:

- `PriceTree.tObs` contains the observation times.
- `PriceTree.PTree` contains the clean prices.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it isn't.

- `PriceTree.ProbTree` contains the probability of reaching each node from root node.
- `PriceTree.ExProbTree` contains the exercise probabilities. Each element in the cell array is an array containing `0`'s where there is no exercise, or the probability of reaching that node where exercise happens.
- `PriceTree.ExProbsByTreeLevel` is an array with each row holding the exercise probability for a given option at each tree observation time.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (calls and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optembndbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J., and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir` | `instoptembnd`

Topics

“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Bond with Embedded Options” on page 2-7
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

optembndbyhjm

Price bonds with embedded options by Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree] = optembndbyhjm(HJMTree,CouponRate,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[Price,PriceTree] = optembndbyhjm( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optembndbyhjm(HJMTree,CouponRate,Settle,Maturity,OptSpec, Strike,ExerciseDates)` calculates price for bonds with embedded options from a Heath-Jarrow-Morton interest-rate tree and returns exercise probabilities in `PriceTree`.

`optembndbyhjm` computes prices of vanilla bonds with embedded options, stepped coupon bonds with embedded options, amortizing bonds with embedded options, and sinking fund bonds with call embedded option. For more information, see “More About” on page 11-1490.

`[Price,PriceTree] = optembndbyhjm( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Callable Bond Using an HJM Interest-Rate Tree Model

Create a `HJMTree` with the following data:

```
Rates = [0.05;0.06;0.07];
Compounding = 1;
StartDates = [datetime(2007,1,1) ; datetime(2008,1,1) ; datetime(2009,1,1)];
EndDates   = [datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1)];
ValuationDate = datetime(2007,1,1);
```

Create a `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [3x1 double]
            Rates: [3x1 double]
         EndTimes: [3x1 double]
       StartTimes: [3x1 double]
         EndDates: [3x1 double]
       StartDates: 733043
    ValuationDate: 733043
            Basis: 0
```

```
     EndMonthRule: 1
```

Create a `VolSpec`.

```
VolSpec = hjmvolspec('Constant', 0.01)
```

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

Create a `TimeSpec`.

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

```
TimeSpec = struct with fields:
           FinObj: 'HJMTimeSpec'
    ValuationDate: 733043
         Maturity: [3x1 double]
      Compounding: 1
            Basis: 0
     EndMonthRule: 1
```

Build the `HJMTree`.

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

```
HJMTree = struct with fields:
      FinObj: 'HJMFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2]
        dObs: [733043 733408 733774]
        TFwd: {[3x1 double]  [2x1 double]  [2]}
      CFlowT: {[3x1 double]  [2x1 double]  [3]}
     FwdTree: {[3x1 double]  [2x1x2 double]  [1x2x2 double]}
```

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 = datetime(2007,1,1);
BondMaturity   = datetime(2010,1,1);
CouponRate = 0.06;
Period = 1;
OptSpec = 'call';
Strike = [98];
ExerciseDates = datetime(2010,1,1);
AmericanOpt = 1;
```

```
[PriceCallBond,PT] = optembndbyhjm(HJMTree, CouponRate, BondSettlement, BondMaturity,...
OptSpec, Strike, ExerciseDates, 'Period', 1,'AmericanOp',1)

PriceCallBond = 95.9492

PT = struct with fields:
                FinObj: 'HJMPriceTree'
                  tObs: [0 1 2 3]
                 PBush: {[95.9492]  [1x1x2 double]  [1x2x2 double]  [98 ... ]}
              ProbBush: {[1]  [1x1x2 double]  [1x2x2 double]  [0.2500 ... ]}
                ExBush: {[0]  [1x1x2 double]  [1x2x2 double]  [1 1 1 1]}
            ExProbBush: {[0]  [1x1x2 double]  [1x2x2 double]  [0.2500 ... ]}
    ExProbsByBushLevel: [0 0 0 1]
```

Obtain Callable Bond Exercise Information Using a HJM Interest-Rate Tree Model

Create a HJMTree with the following data:

```
Rates = [0.05;0.06;0.07];
Compounding = 1;
StartDates = [datetime(2007,1,1) ; datetime(2008,1,1) ; datetime(2009,1,1)];
EndDates   = [datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1)];
ValuationDate = 'jan-1-2007';
```

Create a RateSpec.

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

Build the HJM tree with the following data.

```
VolSpec = hjmvolspec('Constant', 0.01);
TimeSpec = hjmtimespec(ValuationDate, EndDates, Compounding);
HJMTree = hjmtree(VolSpec, RateSpec, TimeSpec);
```

Define the callable bond instruments.

```
Settle = datetime(2007,1,1);
Maturity = [datetime(2009,1,1) ; datetime(2010,1,1)];
CouponRate = {{datetime(2009,1,1) .0325;datetime(2010,1,1) .0375}};
OptSpec='call';
Strike= [90;92];
ExerciseDates= [datetime(2009,1,1) ; datetime(2010,1,1)];
```

Price the instruments.

```
[Price, PriceTree]= optembndbyhjm(HJMTree, CouponRate,  Settle, Maturity, OptSpec, Strike,...
ExerciseDates, 'Period', 1,'AmericanOpt',1)

Price = 2×1

   86.0874
   84.1482
```

```
PriceTree = struct with fields:
                FinObj: 'HJMPriceTree'
                  tObs: [0 1 2 3]
                 PBush: {1x4 cell}
              ProbBush: {[1]  [1x1x2 double]  [1x2x2 double]  [0.2500 ... ]}
                ExBush: {1x4 cell}
            ExProbBush: {1x4 cell}
    ExProbsByBushLevel: [2x4 double]
```

Examine the output `PriceTree.ExBush`. `PriceTree.ExBush` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it is not.

```
PriceTree.ExBush{4}
```

```
ans = 2×4

     0     0     0     0
     1     1     1     1
```

There is no exercise for instrument 1 and instrument 2 is exercised at all nodes.

```
PriceTree.ExBush{3}
```

```
ans =
ans(:,:,1) =

     1     1
     0     0


ans(:,:,2) =

     1     1
     0     0
```

There is exercise for instrument 1 at all nodes and no exercise for instrument 2.

```
PriceTree.ExBush{2}
```

```
ans =
ans(:,:,1) =

     0
     0


ans(:,:,2) =

     0
     0
```

There is no exercise for instrument 1 or instrument 2.

Next view the probability of reaching each node from the root node using `PriceTree.ProbBush`.

```
PriceTree.ProbBush{2}

ans =
ans(:,:,1) =

    0.5000


ans(:,:,2) =

    0.5000
```

```
PriceTree.ProbBush{3}

ans =
ans(:,:,1) =

    0.2500    0.2500


ans(:,:,2) =

    0.2500    0.2500
```

```
PriceTree.ProbBush{4}

ans = 1×4

    0.2500    0.2500    0.2500    0.2500
```

View the exercise probabilities using `PriceTree.ExProbBush`. `PriceTree.ExProbBush` contains the exercise probabilities. Each element in the cell array is an array containing `0`'s where there is no exercise, or the probability of reaching that node where exercise happens.

```
PriceTree.ExProbBush{4}

ans = 2×4

         0         0         0         0
    0.2500    0.2500    0.2500    0.2500
```

```
PriceTree.ExProbBush{3}

ans =
ans(:,:,1) =

    0.2500    0.2500
         0         0


ans(:,:,2) =

    0.2500    0.2500
         0         0
```

```
PriceTree.ExProbBush{2}

ans =
ans(:,:,1) =

     0
     0


ans(:,:,2) =

     0
     0
```

View the exercise probabilities at each tree level using `PriceTree.ExProbsByBushLevel`. `PriceTree.ExProbsByBushLevel` is an array with each row holding the exercise probability for a given option at each tree observation time.

```
PriceTree.ExProbsByBushLevel

ans = 2×4

     0     0     1     0
     0     0     0     1
```

Price Single Stepped Callable Bonds Using an HJM Interest-Rate Tree Model

Price the following single stepped callable bonds using the following data: The data for the interest rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2010,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
Compounding = 1;

% Create RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);

% Instrument
Settle = datetime(2010,1,1);
Maturity = [datetime(2013,1,1) ; datetime(2014,1,1)];
CouponRate = {{datetime(2012,1,1) .0425;datetime(2014,1,1) .0750}};
OptSpec = 'call';
Strike = 100;
ExerciseDates = datetime(2012,1,1);  %Callable in two years

% Build the tree with the following data
Volatility = [.2; .19; .18; .17];
CurveTerm = [ 1;  2;   3;   4];
HJMTimeSpec = hjmtimespec(ValuationDate, EndDates);
HJMVolSpec = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
```

```
HJMT = hjmtree(HJMVolSpec, RS, HJMTimeSpec);

% The first row corresponds to the price of the callable bond with maturity
% of three years. The second row corresponds to the price of the callable
% bond with maturity of four years.
PHJM=  optembndbyhjm(HJMT, CouponRate, Settle, Maturity ,OptSpec, Strike,...
ExerciseDates, 'Period', 1)

PHJM = 2×1

  100.0484
   99.8009
```

Price a Sinking Fund Bond Using an HJM Interest-Rate Tree Model

A corporation issues a three year bond with a sinking fund obligation requiring the company to sink 1/3 of face value after the first year and 1/3 after the second year. The company has the option to buy the bonds in the market or call them at $99. The following data describes the details needed for pricing the sinking fund bond:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
Compounding = 1;
```

Create the RateSpec.

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

RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
       StartDates: 734504
    ValuationDate: 734504
            Basis: 0
     EndMonthRule: 1
```

Build the HJM tree.

```
Sigma = 0.1;
HJMTimeSpec = hjmtimespec(ValuationDate, EndDates, Compounding);
HJMVolSpec = hjmvolspec('Constant', Sigma);
HJMT = hjmtree(HJMVolSpec, RateSpec, HJMTimeSpec)

HJMT = struct with fields:
      FinObj: 'HJMFwdTree'
     VolSpec: [1x1 struct]
```

```
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734504 734869 735235 735600]
        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]}
```

Define the sinking fund instrument. The bond has a coupon rate of 4.5%, a period of one year and matures in 1-Jan-2013. Face decreases 1/3 after the first year.

```
CouponRate = 0.045;
Settle = datetime(2011,1,1);
Maturity =  datetime(2013,1,1);
Period = 1;
Face = { {datetime(2012,1,1) 100; ...
          datetime(2013,1,1) 66.6666}};
```

Define the option provision.

```
OptSpec = 'call';
Strike = 99;
ExerciseDates = datetime(2012,1,1);
```

Price of non-sinking fund bond.

```
PNSF = bondbyhjm(HJMT, CouponRate, Settle, Maturity, Period)
```

```
PNSF = 100.5663
```

Price of the bond with the option sinking provision.

```
PriceSF = optembndbyhjm(HJMT, CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates, 'Period', Period, 'Face', Face)
```

```
PriceSF = 98.8736
```

Price an Amortizing Callable Bond Using an HJM Interest-Rate Tree Model

This example shows how to price an amortizing callable bond and a vanilla callable bond using an HJM lattice model.

Create a `RateSpec`.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ;datetime(2016,1,1)];
Compounding = 1;
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);
```

Build a HJM tree.

```
VolSpec = hjmvolspec('Constant', 0.01)
```

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

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

```
TimeSpec = struct with fields:
           FinObj: 'HJMTimeSpec'
    ValuationDate: 734869
         Maturity: [4x1 double]
      Compounding: 1
            Basis: 0
     EndMonthRule: 1
```

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

```
HJMTree = struct with fields:
      FinObj: 'HJMFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734869 735235 735600 735965]
        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]}
```

Define the callable bond.

```
CouponRate = 0.05;
Settle = datetime(2012,1,1);
Maturity =  datetime(2016,1,1);
Period = 1;

    Face = {
                 {datetime(2014,1,1) 100;
                  datetime(2015,1,1) 70;
                  datetime(2016,1,1) 50};
                 };

OptSpec = 'call';
Strike = [97 95 93];
ExerciseDates = [datetime(2014,1,1) datetime(2015,1,1) datetime(2016,1,1)];
```

Price a callable amortizing bond using the HJM tree.

```
BondType = 'amortizing';
Pcallbonds = optembndbyhjm(HJMTree, CouponRate,  Settle, Maturity ,OptSpec, Strike,...
ExerciseDates, 'Period', 1,'Face',Face,'BondType', BondType)
```

```
Pcallbonds = 98.6000
```

Input Arguments

`HJMTree` — Interest-rate tree structure
structure

Interest-rate tree structure, specified by using `hjmtree`.

Data Types: `struct`

`CouponRate` — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optembndbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char`

`Strike` — Option strike price values
nonnegative integer

Option strike price value, specified as a `NINST`-by-`1` or `NINST`-by-`NSTRIKES` depending on the type of option:

- European option — NINST-by-1 vector of strike price values.
- Bermuda option — NINST by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- American option — NINST-by-1 vector of strike price values for each option.

Data Types: double

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a NINST-by-1, NINST-by-2, or NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a NINST-by-1 vector, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

To support existing code, optembndbyhjm also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: Price = optembndbyhjm(HJMTree,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates,'Period',1,'AmericanOp',1)

AmericanOpt — Option type
0 European/Bermuda (default) | integer with values 0 or 1

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and NINST-by-1 positive integer flags with values:

- 0 — European/Bermuda
- 1 — American

Data Types: double

Period — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of 'Period' and a NINST-by-1 vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

FirstCouponDate — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

LastCouponDate — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

StartDate — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

Face — Face value
`100` (default) | `NINST`-by-`1` vector | `NINST`-by-`1` cell array

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector or a `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated face value. The date indicates the last day that the face value is valid.

Note Instruments without a `Face` schedule are treated as either vanilla bonds or stepped coupon bonds with embedded options.

Data Types: `double`

BondType — Type of underlying bond
'vanilla' for scalar Face values, 'callablesinking' for scheduled Face values (default) | cell array of character vectors with values 'vanilla','amortizing', or 'callablesinking' | string array with values "vanilla", "amortizing", or "callablesinking"

Type of underlying bond, specified as the comma-separated pair consisting of 'BondType' and a NINST-by-1 cell array of character vectors or string array specifying if the underlying is a vanilla bond, an amortizing bond, or a callable sinking fund bond. The supported types are:

- 'vanilla' is a standard callable or puttable bond with a scalar Face value and a single coupon or stepped coupons.
- 'callablesinking' is a bond with a schedule of Face values and a sinking fund call provision with a single or stepped coupons.
- 'amortizing' is an amortizing callable or puttable bond with a schedule of Face values with single or stepped coupons.

Data Types: char | string

Options — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of 'Options' and a structure that is created with derivset.

Data Types: struct

Output Arguments

Price — Expected prices of embedded option at time 0
matrix

Expected price of the embedded option at time 0, returned as a NINST-by-1 matrix.

PriceTree — Structure containing trees of vectors of instrument prices and exercise probabilities for each node
structure

Structure containing trees of vectors of instrument prices, a vector of observation times for each node, and exercise probabilities. Values are:

- PriceTree.ExBush contains the exercise indicator arrays. Each element of the cell array is an array containing 1's where an option is exercised and 0's where it isn't.
- PriceTree.tObs contains the observation times.
- PriceTree.ProbBush contains the probability of reaching each node from root node.
- PriceTree.ExProbBush contains the exercise probabilities. Each element in the cell array is an array containing 0's where there is no exercise, or the probability of reaching that node where exercise happens.
- PriceTree.ExProbsByBushLevel is an array with each row holding the exercise probability for a given option at each tree observation time.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2008a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optembndbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmprice` | `cfamounts` | `hjmtree` | `instoptembnd`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Bond with Embedded Options” on page 2-7
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

optembndbyhw

Price bonds with embedded options by Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = optembndbyhw(HWTree,CouponRate,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[Price,PriceTree] = optembndbyhjm( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optembndbyhw(HWTree,CouponRate,Settle,Maturity,OptSpec, Strike,ExerciseDates)` calculates price for bonds with embedded options from a Hull-White interest-rate tree and returns exercise probabilities in `PriceTree`.

`optembndbyhw` computes prices of vanilla bonds with embedded options, stepped coupon bonds with embedded options, amortizing bonds with embedded options, and sinking fund bonds with call embedded option. For more information, see “More About” on page 11-1505.

Note Alternatively, you can use the `OptionEmbeddedFixedBond` object to price embedded fixed-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optembndbyhjm( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Callable Bond Using an HW Interest-Rate Tree Model

Create a `HWTree` with the following data:

```
ZeroRates = [ 0.035;0.04;0.045];
Compounding = 1;
StartDates = [datetime(2007,1,1) ; datetime(2008,1,1) ; datetime(2009,1,1)];
EndDates   = [datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1)];
ValuationDate = datetime(2007,1,1);
```

Create a `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [3x1 double]
            Rates: [3x1 double]
```

```
         EndTimes: [3x1 double]
       StartTimes: [3x1 double]
         EndDates: [3x1 double]
       StartDates: 733043
    ValuationDate: 733043
            Basis: 0
     EndMonthRule: 1
```

Create a `VolSpec`.

```
VolDates = [datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2010,1,1);
AlphaCurve = 0.1;
HWVolSpec = hwvolspec(ValuationDate, VolDates, VolCurve, AlphaDates, AlphaCurve)
```

```
HWVolSpec = struct with fields:
             FinObj: 'HWVolSpec'
      ValuationDate: 733043
           VolDates: [3x1 double]
           VolCurve: [3x1 double]
         AlphaCurve: 0.1000
         AlphaDates: 734139
    VolInterpMethod: 'linear'
```

Create a `TimeSpec`.

```
HWTimeSpec = hwtimespec(ValuationDate, EndDates, Compounding)
```

```
HWTimeSpec = struct with fields:
           FinObj: 'HWTimeSpec'
    ValuationDate: 733043
         Maturity: [3x1 double]
      Compounding: 1
            Basis: 0
     EndMonthRule: 1
```

Build the `HWTree`.

```
HWTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec)
```

```
HWTree = struct with fields:
      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2]
        dObs: [733043 733408 733774]
      CFlowT: {[3x1 double]  [2x1 double]  [3]}
       Probs: {[3x1 double]  [3x3 double]}
     Connect: {[2]  [2 3 4]}
     FwdTree: {[1.0350]  [1.0633 1.0451 1.0271]  [1.0925 1.0737 1.0553 ... ]}
```

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 = datetime(2007,1,1);
BondMaturity   = datetime(2010,1,1);
CouponRate = 0.0525;
Period = 1;
OptSpec = 'put';
Strike = [100];
ExerciseDates = [datetime(2008,1,1) datetime(2010,1,1)];
AmericanOpt = 1;

PricePutBondHW = optembndbyhw(HWTree, CouponRate, BondSettlement, BondMaturity,...
OptSpec, Strike, ExerciseDates,'Period', 1, 'AmericanOpt', 1)

PricePutBondHW = 102.9127
```

Obtain Callable Bond Exercise Information Using an HW Interest-Rate Tree Model

Create a `HWTree` with the following data:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2019,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2020,1,1) ; datetime(2021,1,1) ; datetime(2022,1,1) ; datetime(2023,1,1)];
Compounding = 1;
```

Create a `RateSpec`.

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

Define the callable bond instruments.

```
Settle = datetime(2019,1,1);
Maturity = [datetime(2022,1,1) ; datetime(2023,1,1)];
CouponRate = {{datetime(2021,1,1) .0425 ; datetime(2023,1,1) .0450}};
OptSpec='call';
Strike= [98;95];
ExerciseDates= [datetime(2021,1,1) ; datetime(2022,1,1)];
```

Build the HW tree with the following data.

```
VolDates = [datetime(2020,1,1) ; datetime(2021,1,1) ; datetime(2022,1,1) ; datetime(2023,1,1)];
VolCurve = 0.05;
AlphaDates = datetime(2023,1,1);
AlphaCurve = 0.05;


HWVolSpec = hwvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RS.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RS, HWTimeSpec);
```

Price the instruments.

```
[Price, PriceTree]= optembndbyhw(HWT, CouponRate,  Settle, Maturity,OptSpec, Strike,...
ExerciseDates, 'Period', 1)
```

```
Price = 2×1

   96.4131
   92.9341


PriceTree = struct with fields:
                FinObj: 'HWPriceTree'
                  tObs: [0 1 2 3 4]
                 PTree: {1x5 cell}
              ProbTree: {1x5 cell}
                 ExTree: {1x5 cell}
            ExProbTree: {1x5 cell}
    ExProbsByTreeLevel: [2x5 double]
               Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
```

Examine the output `PriceTree.ExTree`. `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it is not.

```
PriceTree.ExTree{5}
```

```
ans = 2×7

     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
```

There is no exercise for instrument 1 or 2.

```
PriceTree.ExTree{4}
```

```
ans = 2×7

     0     0     0     0     0     0     0
     0     0     0     1     1     1     1
```

There is no exercise for instrument 1 and instrument 2 is exercised at some nodes.

```
PriceTree.ExTree{3}
```

```
ans = 2×5

     0     0     1     1     1
     0     0     0     0     0
```

There is the exercise for instrument 1 at some node and no exercise for instrument 2.

```
PriceTree.ExTree{2}
```

```
ans = 2×3

     0     0     0
     0     0     0
```

There is no exercise for instrument 1 or 2.

Next view the probability of reaching each node from the root node using `PriceTree.ProbTree`.

```
PriceTree.ProbTree{2}
```

```
ans = 1×3

    0.1667    0.6667    0.1667
```

```
PriceTree.ProbTree{3}
```

```
ans = 1×5

    0.0238    0.2218    0.5087    0.2218    0.0238
```

```
PriceTree.ProbTree{4}
```

```
ans = 1×7

    0.0029    0.0473    0.2374    0.4247    0.2374    0.0473    0.0029
```

```
PriceTree.ProbTree{5}
```

```
ans = 1×7

    0.0029    0.0473    0.2374    0.4247    0.2374    0.0473    0.0029
```

Then view the exercise probabilities using `PriceTree.ExProbTree`. `PriceTree.ExProbTree` contains the exercise probabilities. Each element in the cell array is an array containing 0's where there is no exercise, or the probability of reaching that node where exercise happens.

```
PriceTree.ExProbTree{5}
```

```
ans = 2×7

     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
```

```
PriceTree.ExProbTree{4}
```

```
ans = 2×7

         0         0         0         0         0         0         0
         0         0         0    0.4247    0.2374    0.0473    0.0029
```

```
PriceTree.ExProbTree{3}
```

```
ans = 2×5

         0         0    0.5087    0.2218    0.0238
         0         0         0         0         0
```

```
PriceTree.ExProbTree{2}
```

```
ans = 2×3

     0     0     0
     0     0     0
```

View the exercise probabilities at each tree level using `PriceTree.ExProbsByTreeLevel`. `PriceTree.ExProbsByTreeLevel` is an array with each row holding the exercise probability for a given option at each tree observation time.

```
PriceTree.ExProbsByTreeLevel
```

```
ans = 2×5

         0         0    0.7544         0         0
         0         0         0    0.7124         0
```

Price Single Stepped Callable Bonds Using an HW Interest-Rate Tree Model

Price the following single stepped callable bonds using the following data: The data for the interest rate term structure is as follows:

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2010,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1); datetime(2014,1,1)];
Compounding = 1;

% Create RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);

% Instrument
Settle = datetime(2010,1,1);
Maturity = [datetime(2013,1,1) ; datetime(2014,1,1)];
CouponRate = {{datetime(2012,1,1) .0425;datetime(2014,1,1) .0750}};
OptSpec = 'call';
Strike = 100;
ExerciseDates = datetime(2012,1,1);  %Callable in two years

% Build the tree with the following data
VolDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2014,1,1);
AlphaCurve = 0.1;


HWVolSpec = hwvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RS.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RS, HWTimeSpec);

% The first row corresponds to the price of the callable bond with maturity
% of three years. The second row corresponds to the price of the callable
```

```
% bond with maturity of four years.

PHW= optembndbyhw(HWT, CouponRate,  Settle, Maturity,OptSpec, Strike,...
ExerciseDates, 'Period', 1)

PHW = 2×1

  100.0326
   99.7987
```

Price a Sinking Fund Bond Using an HW Interest-Rate Tree Model

A corporation issues a two year bond with a sinking fund obligation requiring the company to sink 1/3 of face value after the first year. The company has the option to buy the bonds in the market or call them at $99. The following data describes the details needed for pricing the sinking fund bond:

```
Rates = [0.1;0.1;0.1;0.1];
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
Compounding = 1;

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

% Build the HW tree
% The data to build the tree is as follows:
VolDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2015,1,1);
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RateSpec, HWTimeSpec);


% Instrument
% The bond has a coupon rate of 9%, a period of one year and matures in
% 1-Jan-2013. Face decreases 1/3 after the first year.
CouponRate = 0.09;
Settle = datetime(2011,1,1);
Maturity =  datetime(2013,1,1);
Period = 1;
Face = { ...
            {datetime(2012,1,1) 100; ...
             datetime(2013,1,1) 66.6666}; ...
        };

% Option provision
OptSpec = 'call';
Strike = 99;
```

```
ExerciseDates = datetime(2012,1,1);

% Price of non-sinking fund bond.
PNSF = bondbyhw(HWT, CouponRate, Settle, Maturity, Period)

PNSF = 98.2645
```

Price of the bond with the option sinking provision.

```
PriceSF = optembndbyhw(HWT, CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates, 'Period', Period, 'Face', Face)

PriceSF = 98.1553
```

Price an Amortizing Callable Bond Using an HW Interest-Rate Tree Model

This example shows how to price an amortizing callable bond and a vanilla callable bond using an HW lattice model.

Create a `RateSpec`.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates,'Rates', Rates, 'Compounding', Compounding);
```

Build a HW tree.

```
VolDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2016,1,1);
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RS.ValuationDate, EndDates, Compounding);
HWT = hwtree(HWVolSpec, RS, HWTimeSpec);
```

Define the callable bond.

```
CouponRate = 0.05;
Settle = datetime(2012,1,1);
Maturity = datetime(2016,1,1);
Period = 1;

    Face = {
                {datetime(2014,1,1) 100;
                 datetime(2015,1,1) 70;
                 datetime(2016,1,1) 50};
                };

OptSpec = 'call';
```

```
Strike = [97 95 93];
ExerciseDates = [datetime(2014,1,1) datetime(2015,1,1) datetime(2016,1,1) ];
```

Price a callable amortizing bond using the HW tree.

```
BondType = 'amortizing';
Pcallbonds = optembndbyhw(HWT, CouponRate,  Settle, Maturity ,OptSpec, Strike,...
ExerciseDates, 'Period', 1,'Face',Face,'BondType', BondType)
```

```
Pcallbonds = 98.6554
```

Input Arguments

`HWTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hwtree`.

Data Types: `struct`

`CouponRate` — Bond coupon rate

positive decimal value

Bond coupon rate, specified as an `NINST`-by-`1` decimal annual rate or `NINST`-by-`1` cell array, where each element is a `NumDates`-by-`2` cell array. The first column of the `NumDates`-by-`2` cell array is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Data Types: `double` | `cell`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date for the bond option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optembndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as a NINST-by-1 cell array of character vectors.

Data Types: char

Strike — Option strike price values

nonnegative integer

Option strike price value, specified as a NINST-by-1 or NINST-by-NSTRIKES depending on the type of option:

- European option — NINST-by-1 vector of strike price values.
- Bermuda option — NINST by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- American option — NINST-by-1 vector of strike price values for each option.

Data Types: double

ExerciseDates — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a NINST-by-1, NINST-by-2, or NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the type of option:

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a NINST-by-1 vector, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

To support existing code, optembndbyhw also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: Price = optembndbyhw(HWTree,CouponRate,Settle,Maturity,OptSpec,Strike,ExerciseDates,'Period',1,'AmericanOp',1)

AmericanOpt — Option type

0 European/Bermuda (default) | integer with values 0 or 1

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and NINST-by-1 positive integer flags with values:

- 0 — European/Bermuda

- `1` — American

Data Types: `double`

`Period` — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
datetime array | string array | date character vector

Bond issue date,specified as the comma-separated pair consisting of `'IssueDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

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. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optembndbyhw` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

`Face` — Face value
`100` (default) | `NINST`-by-`1` vector | `NINST`-by-`1` cell array

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector or a `NINST`-by-`1` cell array where each element is a `NumDates`-by-`2` cell array where the first

column is dates and the second column is associated face value. The date indicates the last day that the face value is valid.

Note Instruments without a `Face` schedule are treated as either vanilla bonds or stepped coupon bonds with embedded options.

Data Types: `double`

`BondType` — Type of underlying bond
`'vanilla'` for scalar `Face` values, `'callablesinking'` for scheduled `Face` values (default) | cell array of character vectors with values `'vanilla'`,`'amortizing'`, or `'callablesinking'` | string array with values `"vanilla"`, `"amortizing"`, or `"callablesinking"`

Type of underlying bond, specified as the comma-separated pair consisting of `'BondType'` and a `NINST`-by-`1` cell array of character vectors or string array specifying if the underlying is a vanilla bond, an amortizing bond, or a callable sinking fund bond. The supported types are:

- `'vanilla'` is a standard callable or puttable bond with a scalar `Face` value and a single coupon or stepped coupons.
- `'callablesinking'` is a bond with a schedule of `Face` values and a sinking fund call provision with a single or stepped coupons.
- `'amortizing'` is an amortizing callable or puttable bond with a schedule of `Face` values with single or stepped coupons.

Data Types: `char` | `string`

`Options` — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of embedded option at time `0`
matrix

Expected price of the embedded option at time `0`, returned as a `NINST`-by-`1` matrix.

`PriceTree` — Structure containing trees of vectors of instrument prices and exercise probabilities for each node
structure

Structure containing trees of vectors of instrument prices, a vector of observation times for each node, and exercise probabilities. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it isn't.

- `PriceTree.ProbTree` contains the probability of reaching each node from root node.
- `PriceTree.ExProbTree` contains the exercise probabilities. Each element in the cell array is an array containing `0`'s where there is no exercise, or the probability of reaching that node where exercise happens.
- `PriceTree.ExProbsByTreeLevel` is an array with each row holding the exercise probability for a given option at each tree observation time.

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Sinking Fund Bond with Call Embedded Option

A sinking fund bond is a coupon bond with a sinking fund provision.

This provision obligates the issuer to amortize portions of the principal prior to maturity, affecting bond prices since the time of the principal repayment changes. This means that investors receive the coupon and a portion of the principal paid back over time. These types of bonds reduce credit risk, since it lowers the probability of investors not receiving their principal payment at maturity.

The bond may have a sinking fund call option provision allowing the issuer to retire the sinking fund obligation either by purchasing the bonds to be redeemed from the market or by calling the bond via a sinking fund call, whichever is cheaper. If interest rates are high, then the issuer buys back the requirement amount of bonds from the market since bonds are cheap, but if interest rates are low (bond prices are high), then most likely the issuer is buying the bonds at the call price. Unlike a call feature, however, if a bond has a sinking fund call option provision, it is an obligation, not an option, for the issuer to buy back the increments of the issue as stated. Because of this, a sinking fund bond trades at a lower price than a non-sinking fund bond.

Amortizing Callable or Puttable Bond

Amortizing callable or puttable bonds work under a scheduled `Face`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Face` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Version History

Introduced in R2008a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optembndbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hwprice` | `cfamounts` | `hwtree` | `instoptembnd` | `OptionEmbeddedFixedBond`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond with Embedded Options” on page 2-7
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optemfloatbybdt

Price embedded option on floating-rate note for Black-Derman-Toy interest-rate tree

Syntax

```
[Price,PriceTree] = optemfloatbybdt(BDTTree,Spread,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[Price,PriceTree] = optemfloatbybdt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optemfloatbybdt(BDTTree,Spread,Settle,Maturity,OptSpec, Strike,ExerciseDates)` prices embedded options on floating-rate notes from a Black-Derman-Toy interest rate tree. `optemfloatbybdt` computes prices of vanilla floating-rate notes with embedded options.

Note Alternatively, you can use the `OptionEmbeddedFloatBond` object to price embedded floating-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optemfloatbybdt( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price Callable Embedded Option for Floating-Rate Note

Define the interest-rate term structure.

```
Rates = [0.03;0.034;0.038;0.04];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
```

Create the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
```

```
      StartDates: 734869
  ValuationDate: 734869
          Basis: 0
   EndMonthRule: 1
```

Build the BDT tree and assume a volatility of 10%.

```
Sigma = 0.1;
BDTTimeSpec = bdttimespec(ValuationDate, EndDates);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Define the floater instruments with the embedded call option.

```
Spread = 10;
Settle = datetime(2012,1,1);
Maturity =  [datetime(2015,1,1) ; datetime(2016,1,1)];
Period = 1;
OptSpec = {'call'};
Strike = 101;
ExerciseDates = datetime(2015,1,1);
```

Compute the price of the floaters with the embedded call.

```
Price= optemfloatbybdt(BDTT, Spread, Settle, Maturity, OptSpec, Strike,...
ExerciseDates)
```

```
Price = 2×1

  100.2800
  100.3655
```

Input Arguments

BDTTree — Interest-rate tree structure

binomial tree structure

Interest-rate tree specified as a structure by using `bdttree`.

Data Types: `struct`

Spread — Number of basis points over the reference rate

nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (`NINST`)-by-1).

Data Types: `double`

Settle — Settlement dates of floating-rate note

`ValuationDate` of BDT Tree (default) | datetime array | string array | date character vector

Settlement dates of floating-rate note specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optemfloatbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Floating-rate note maturity date
datetime array | string array | date character vector

Floating-rate note maturity date specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors of dates.

To support existing code, `optemfloatbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector | cell array of character vectors

Definition of option as `'call'` or `'put'` specified as a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `cell` | `char`

`Strike` — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values specified nonnegative integers using as `NINST`-by-`NSTRIKES` vector of strike price values.

Data Types: `double`

`ExerciseDates` — Exercise date for option (European, Bermuda, or American)
datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American) specified as a `NINST`-by-`NSTRIKES` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors.

To support existing code, `optemfloatbybdt` also accepts serial date numbers as inputs, but they are not recommended.

- If a European or Bermuda option, the `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.
- If an American option, then `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`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = optemfloatbybdt(BDTTree,Spread,Settle,Maturity,OptSpec,Strike,ExerciseDates,'AmericanOpt',1,'FloatReset',6,'Basis',8)`

`AmericanOpt` — Option type
scalar | vector of positive integers `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`FloatReset` — Frequency of payments per year
`1` (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of payments per year, specified as the comma-separated pair consisting of `'FloatReset'` and positive integers for the values `[1,2,3,4,6,12]` in a `NINST`-by-`1` vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there will be more than one possible path for connecting the two payment dates.

Data Types: `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `NINST`-by-`1` vector. The `Basis` value represents the basis used when annualizing the input forward-rate tree.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`Principal` — Principal values
`100` (default) | vector of nonnegative values | cell array of nonnegative values

Principal values, specified as the comma-separated pair consisting of `'Principal'` and nonnegative values using a `NINST`-by-`1` vector or `NINST`-by-`1` cell array of notional principal amounts. When using a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of the floating-rate note option at time 0
scalar | vector

Expected prices of the floating-rate note option at time 0 are returned as a scalar or an `NINST`-by-`1` vector.

`PriceTree` — Structure of trees containing vectors of option prices at each node
tree structure

Structure of trees containing vectors of instrument prices and accrued interest and a vector of observation times for each node returned as:

- `PriceTree.PTree` contains option prices.
- `PriceTree.tObs` contains the observation times.

More About

Floating-Rate Note with Embedded Options

A floating-rate note with an embedded option enables floating-rate notes to have early redemption features.

A FRN with an embedded option gives investors or issuers the option to retire the outstanding principal prior to maturity. An embedded call option gives the right to retire the note prior to the maturity date (callable floater), and an embedded put option gives the right to sell the note back at a specific price (puttable floater).

For more information, see “Floating-Rate Note with Embedded Options” on page 2-11.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optemfloatbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`optembndbybdt` | `optemfloatbyhjm` | `optemfloatbybk` | `optemfloatbyhw` | `instoptemfloat` | `OptionEmbeddedFloatBond`

Topics

“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Floating-Rate Note with Embedded Options” on page 2-11
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optemfloatbybk

Price embedded option on floating-rate note for Black-Karasinski interest-rate tree

Syntax

```
[Price,PriceTree] = optemfloatbybk(BKTree,Spread,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[Price,PriceTree] = optemfloatbybk( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optemfloatbybk(BKTree,Spread,Settle,Maturity,OptSpec, Strike,ExerciseDates)` prices embedded options on floating-rate notes from a Black-Karasinski interest rate tree. `optemfloatbybk` computes prices of vanilla floating-rate notes with embedded options.

Note Alternatively, you can use the `OptionEmbeddedFloatBond` object to price embedded floating-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optemfloatbybk( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price European Callable Embedded Option for Floating-Rate Note

Define the interest-rate term structure.

```
Rates = [0.03;0.034;0.038;0.04];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
```

Create the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
```

```
       StartDates: 734869
    ValuationDate: 734869
            Basis: 0
     EndMonthRule: 1
```

Build the BK tree.

```
VolDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2016,1,1);
AlphaCurve = 0.1;

BKVolSpec = bkvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
            AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RateSpec.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RateSpec, BKTimeSpec)
```

```
BKT = struct with fields:
      FinObj: 'BKFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734869 735235 735600 735965]
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
     FwdTree: {1x4 cell}
```

The floater instrument has a spread of 15, a period of one year, and matures and is callable on Jan-1-2015.

```
Spread = 15;
Settle = datetime(2012,1,1);
Maturity =  datetime(2015,1,1);
Period = 1;
OptSpec = {'call'};
Strike =101;
ExerciseDates = datetime(2015,1,1);
```

Compute the price of the floater with the embedded call.

```
Price= optemfloatbybk(BKT, Spread, Settle, Maturity,...
OptSpec, Strike, ExerciseDates)
```

```
Price = 100.4201
```

Input Arguments

BKTree — Interest-rate tree structure

binomial tree structure

Interest-rate tree specified as a structure by using `bktree`.

Data Types: `struct`

Spread — Number of basis points over the reference rate
nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (NINST)-by-1).

Data Types: single | double

Settle — Settlement dates of floating-rate note
ValuationDate of BDT Tree (default) | datetime array | string array | date character vector

Settlement dates of floating-rate note specified a NINST-by-1 vector using a datetime array, string array, or date character vectors.

Note The Settle date for every floating-rate note with an embedded option is set to the ValuationDate of the BK tree. The floating-rate note argument Settle is ignored.

To support existing code, optemfloatbybk also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Floating-rate note maturity date
datetime array | string array | date character vector

Floating-rate note maturity date specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optemfloatbybk also accepts serial date numbers as inputs, but they are not recommended.

OptSpec — Definition of option
character vector | cell array of character vectors

Definition of option as 'call' or 'put' specified as a NINST-by-1 cell array of character vectors for 'call' or 'put'.

Data Types: cell | char

Strike — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values specified nonnegative integers using as NINST-by-NSTRIKES vector of strike price values.

Data Types: single | double

ExerciseDates — Exercise date for option (European, Bermuda, or American)
datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American) specified as a NINST-by-NSTRIKES or NINST-by-2 vector using a datetime array, string array, or date character vectors.

To support existing code, optemfloatbybk also accepts serial date numbers as inputs, but they are not recommended.

- If a European or Bermuda option, the 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.
- If an American option, then 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.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree] = optemfloatbybk(BKTree,Spread,Settle,Maturity,OptSpec,Strike,ExerciseDates,'AmericanOpt',1,'FloatReset',6,'Basis',8)

AmericanOpt — Option type

scalar | vector of positive integers [0,1]

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and NINST-by-1 positive integer scalar flags with values:

- 0 — European/Bermuda
- 1 — American

Data Types: double

FloatReset — Frequency of payments per year

1 (default) | positive integer from the set [1,2,3,4,6,12] | vector of positive integers from the set [1,2,3,4,6,12]

Frequency of payments per year, specified as the comma-separated pair consisting of 'FloatReset' and positive integers for the values [1,2,3,4,6,12] in a NINST-by-1 vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there will be more than one possible path for connecting the two payment dates.

Data Types: double

Basis — Day-count basis of the instrument

0 (actual/actual) (default) | positive integers of the set [1...13] | vector of positive integers of the set [1...13]

Day-count basis of the instrument, specified as the comma-separated pair consisting of 'Basis' and a positive integer using a NINST-by-1 vector. The Basis value represents the basis used when annualizing the input forward-rate tree.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`Principal` — Principal values
`100` (default) | vector of nonnegative values | cell array of nonnegative values

Principal values, specified as the comma-separated pair consisting of `'Principal'` and nonnegative values using a `NINST`-by-`1` vector or `NINST`-by-`1` cell array of notional principal amounts. When using a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of the floating-rate note option at time 0

scalar | vector

Expected prices of the floating-rate note option at time 0 are returned as a scalar or an `NINST`-by-`1` vector.

`PriceTree` — Structure of trees containing vectors of option prices at each node

tree structure

Structure of trees containing vectors of instrument prices and accrued interest and a vector of observation times for each node returned as:

- `PriceTree.PTree` contains option prices.
- `PriceTree.tObs` contains the observation times.

More About

Floating-Rate Note with Embedded Options

A floating-rate note with an embedded option enables floating-rate notes to have early redemption features.

A FRN with an embedded option gives investors or issuers the option to retire the outstanding principal prior to maturity. An embedded call option gives the right to retire the note prior to the maturity date (callable floater), and an embedded put option gives the right to sell the note back at a specific price (puttable floater).

For more information, see “Floating-Rate Note with Embedded Options” on page 2-11.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optemfloatbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`optembndbybk` | `optemfloatbyhjm` | `optemfloatbybdt` | `optemfloatbyhw` | `instoptemfloat` | `OptionEmbeddedFloatBond`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Floating-Rate Note with Embedded Options” on page 2-11
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optemfloatbycir

Price embedded option on floating-rate note for Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = optemfloatbycir(CIRTree,Spread,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[Price,PriceTree] = optemfloatbycir( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optemfloatbycir(CIRTree,Spread,Settle,Maturity,OptSpec, Strike,ExerciseDates)` prices embedded options on floating-rate notes from a Cox-Ingersoll-Ross (CIR) interest rate tree. `optemfloatbycir` computes prices of vanilla floating-rate notes with embedded options using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = optemfloatbycir( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price European Callable Embedded Option for a Floating-Rate Note Using a CIR Interest-Rate Tree

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = datetime(2017,1,1);
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',EndD
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = datetime(2017,1,1);
Maturity = datetime(2020,1,1);
CIRTimeSpec = cirtimespec(Settle, Maturity, 3);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
```

```
        tObs: [0 1 2]
        dObs: [736696 737061 737426]
     FwdTree: {[1.0350]  [1.0790 1.0500 1.0298]  [1.1275 1.0887 1.0594 ... ]}
     Connect: {[3x1 double]  [3x3 double]}
       Probs: {[3x1 double]  [3x3 double]}
```

Define the floater instruments with the embedded call option.

```
Spread = 10;
Settle = datetime(2017,1,1);
Maturity =  [datetime(2019,1,1) ; datetime(2020,1,1)];
Period = 1;
OptSpec = {'call'};
Strike = 101;
ExerciseDates = datetime(2019,1,1);
```

Compute the price of the floaters with the embedded call.

```
[Price,PriceTree] = optemfloatbycir(CIRT,Spread,Settle,Maturity,OptSpec,Strike,ExerciseDates)
```

```
Price = 2×1

  100.1887
  100.2757


PriceTree = struct with fields:
    FinObj: 'CIRPriceTree'
      tObs: [0 1 2 3]
     PTree: {[2x1 double]  [2x3 double]  [2x5 double]  [2x5 double]}
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree specified as a structure by using `cirtree`.

Data Types: `struct`

`Spread` — Number of basis points over the reference rate

nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (`NINST`-by-1).

Data Types: `double`

`Settle` — Settlement dates of floating-rate note

`ValuationDate` of CIR tree (default) | datetime array | string array | date character vector

Settlement dates of floating-rate note specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every floating-rate note with an embedded option is set to the `ValuationDate` of the CIR tree. The floating-rate note argument `Settle` is ignored.

To support existing code, `optemfloatbycir` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Floating-rate note maturity date
datetime array | string array | date character vector

Floating-rate note maturity date specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optemfloatbycir` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with value of `'call'` or `'put'` | cell array of character vectors with value of `'call'` or `'put'` | string array with value of `"call"` or `"put"`

Definition of option, specified as a `NINST`-by-`1` cell array of character vectors or string arrays with a value of `'call'` or `'put'`.

Data Types: `cell` | `char` | `string`

`Strike` — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values specified nonnegative integers using as `NINST`-by-`NSTRIKES` vector of strike price values.

Data Types: `double`

`ExerciseDates` — Exercise date for option (European, Bermuda, or American)
datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American) specified as a `NINST`-by-`NSTRIKES` or an `NINST`-by-`2` vector using a datetime array, string array, or date character vectors.

- For a European or Bermuda option, the `ExerciseDates` is a `1`-by-`1` (European) or `1`-by-`NSTRIKES` (Bermuda) vector of exercise dates. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For an American option, the `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 `ExerciseDates`.

To support existing code, `optemfloatbycir` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = optemfloatbycir(CIRTree,Spread,Settle,Maturity,OptSpec,Strike,ExerciseDates,'AmericanOpt',1,'FloatReset',6,'Basis',8)`

`AmericanOpt` — Option type
scalar | vector of positive integers `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a `NINST`-by-`1` vector of flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`FloatReset` — Frequency of payments per year
`1` (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of payments per year, specified as the comma-separated pair consisting of `'FloatReset'` and positive integers for the values `[1,2,3,4,6,12]` in a `NINST`-by-`1` vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there will be more than one possible path for connecting the two payment dates.

Data Types: `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `NINST`-by-`1` vector. The `Basis` value represents the basis used when annualizing the input forward-rate tree.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`Principal` — Principal values
`100` (default) | vector of nonnegative values | cell array of nonnegative values

Principal values, specified as the comma-separated pair consisting of `'Principal'` and nonnegative values using a `NINST`-by-`1` vector or `NINST`-by-`1` cell array of notional principal amounts.

When using a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array where the first column is dates, and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and the output from `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of the floating-rate note option at time 0
scalar | vector

Expected prices of the floating-rate note option at time 0 are returned as a scalar or an `NINST`-by-`1` vector.

`PriceTree` — Structure of trees containing vectors of option prices at each node
tree structure

Structure of trees containing vectors of instrument prices and accrued interest and a vector of observation times for each node returned as:

- `PriceTree.tObs` contains the observation times.
- `PriceTree.PTree` contains option prices.

More About

Floating-Rate Note with Embedded Options

A floating-rate note with an embedded option enables floating-rate notes to have early redemption features.

A FRN with an embedded option gives investors or issuers the option to retire the outstanding principal prior to maturity. An embedded call option gives the right to retire the note prior to the maturity date (callable floater), and an embedded put option gives the right to sell the note back at a specific price (puttable floater).

For more information, see “Floating-Rate Note with Embedded Options” on page 2-11.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optemfloatbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J., and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica.* Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice.* Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance.* CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling.* Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir` | `instoptemfloat`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Floating-Rate Note with Embedded Options” on page 2-11
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

optemfloatbyhjm

Price embedded option on floating-rate note for Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree] = optemfloatbyhjm(HJMTree,Spread,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[Price,PriceTree] = optemfloatbyhjm( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optemfloatbyhjm(HJMTree,Spread,Settle,Maturity,OptSpec, Strike,ExerciseDates)` prices embedded options on floating-rate notes from a Heath-Jarrow-Morton interest rate tree. `optemfloatbybk` computes prices of vanilla floating-rate notes with embedded options.

`[Price,PriceTree] = optemfloatbyhjm( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price European Callable Embedded Option for a Floating-Rate Note

Define the interest-rate term structure.

```
Rates = [0.03;0.035;0.040;0.045];
ValuationDate = datetime(2012,1,1);
StartDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
EndDates =[datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
```

Create the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
          EndTimes: [4x1 double]
        StartTimes: [4x1 double]
          EndDates: [4x1 double]
        StartDates: [4x1 double]
    ValuationDate: 734869
            Basis: 0
     EndMonthRule: 1
```

Build the HJM tree.

```
VolSpec = hjmvolspec('Constant', 0.01);
TimeSpec = hjmtimespec(RateSpec.ValuationDate, EndDates, Compounding);
HJMTree = hjmtree(VolSpec, RateSpec, TimeSpec)
```

```
HJMTree = struct with fields:
      FinObj: 'HJMFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734869 735235 735600 735965]
        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]}
```

The floater instrument has a spread of 15, a period of one year, and matures and is callable on Jan-1-2016.

```
Spread = 15;
Settle = datetime(2012,1,1);
Maturity = datetime(2016,1,1);
Period = 1;
OptSpec = {'call'};
Strike = 95;
ExerciseDates = datetime(2016,1,1);
```

Compute the price of the floater with the embedded call.

```
Price = optemfloatbyhjm(HJMTree, Spread, Settle, Maturity,...
OptSpec, Strike, ExerciseDates)
```

```
Price = 96.2355
```

Input Arguments

`HJMTree` — Interest-rate tree structure

binomial tree structure

Interest-rate tree specified as a structure by using `hjmtree`.

Data Types: `struct`

`Spread` — Number of basis points over the reference rate

nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (`NINST`)-by-1).

Data Types: `double`

`Settle` — Settlement dates of floating-rate note

`ValuationDate` of BDT Tree (default) | datetime array | string array | date character vector

Settlement dates of floating-rate note specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every floating-rate note with an embedded option is set to the `ValuationDate` of the HJM tree. The floating-rate note argument `Settle` is ignored.

To support existing code, `optemfloatbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Floating-rate note maturity date
datetime array | string array | date character vector

Floating-rate note maturity date specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optemfloatbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector | cell array of character vectors

Definition of option as `'call'` or `'put'` specified as a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `cell` | `char`

`Strike` — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values specified nonnegative integers using as `NINST`-by-`NSTRIKES` vector of strike price values.

Data Types: `double`

`ExerciseDates` — Exercise date for option (European, Bermuda, or American)
datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American) specified as a `NINST`-by-`NSTRIKES` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors.

To support existing code, `optemfloatbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

- If a European or Bermuda option, the `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.
- If an American option, then `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`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = optemfloatbyhjm(HJMTree,Spread,Settle,Maturity,OptSpec,Strike,ExerciseDates,'AmericanOpt',1,'FloatReset',6,'Basis',8)`

`AmericanOpt` — Option type
scalar | vector of positive integers `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`FloatReset` — Frequency of payments per year
`1` (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of payments per year, specified as the comma-separated pair consisting of `'FloatReset'` and positive integers for the values `[1,2,3,4,6,12]` in a `NINST`-by-`1` vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there will be more than one possible path for connecting the two payment dates.

Data Types: `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `NINST`-by-`1` vector. The `Basis` value represents the basis used when annualizing the input forward-rate tree.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`Principal` — Principal values
`100` (default) | vector of nonnegative values | cell array of nonnegative values

Principal values, specified as the comma-separated pair consisting of `'Principal'` and nonnegative values using a `NINST`-by-`1` vector or `NINST`-by-`1` cell array of notional principal amounts. When using a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of the floating-rate note option at time 0
scalar | vector

Expected prices of the floating-rate note option at time 0 are returned as a scalar or an `NINST`-by-`1` vector.

`PriceTree` — Structure of trees containing vectors of option prices at each node
tree structure

Structure of trees containing vectors of instrument prices and accrued interest and a vector of observation times for each node returned as:

- `PriceTree.PBush` contains the clean prices.
- `PriceTree.tObs` contains the observation times.

More About

Floating-Rate Note with Embedded Options

A floating-rate note with an embedded option enables floating-rate notes to have early redemption features.

A FRN with an embedded option gives investors or issuers the option to retire the outstanding principal prior to maturity. An embedded call option gives the right to retire the note prior to the maturity date (callable floater), and an embedded put option gives the right to sell the note back at a specific price (puttable floater).

For more information, see “Floating-Rate Note with Embedded Options” on page 2-11.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optemfloatbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`optembndbyhjm` | `optemfloatbyhw` | `optemfloatbybdt` | `optemfloatbybk` | `instoptemfloat`

Topics

“Computing Instrument Prices” on page 2-81
“Floating-Rate Note with Embedded Options” on page 2-11
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

optemfloatbyhw

Price embedded option on floating-rate note for Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = optemfloatbyhw(HWTree,Spread,Settle,Maturity,OptSpec,
Strike,ExerciseDates)
[Price,PriceTree] = optemfloatbyhw( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optemfloatbyhw(HWTree,Spread,Settle,Maturity,OptSpec, Strike,ExerciseDates)` prices embedded options on floating-rate notes from a Hull-White interest rate tree. `optemfloatbybk` computes prices of vanilla floating-rate notes with embedded options.

Note Alternatively, you can use the `OptionEmbeddedFloatBond` object to price embedded floating-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optemfloatbyhw( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price European Callable Embedded Option for Floating-Rate Note

Define the interest-rate term structure.

```
Rates = [0.03;0.034;0.038;0.04];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
```

Create the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
       StartDates: 734869
    ValuationDate: 734869
```

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

Build the HW tree using the following:

```
VolDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2016,1,1);
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
            AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RateSpec, HWTimeSpec)
```

```
HWT = struct with fields:
      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734869 735235 735600 735965]
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
     FwdTree: {1x4 cell}
```

Define the floater instruments with the embedded call option.

```
Spread = 10;
Settle = datetime(2012,1,1);
Maturity =  [datetime(2015,1,1) ; datetime(2016,1,1)];
Period = 1;
OptSpec = {'call'};
Strike = 101;
ExerciseDates = datetime(2015,1,1);
```

Compute the price of the floaters with the embedded call.

```
Price= optemfloatbyhw(HWT, Spread, Settle, Maturity, OptSpec, Strike,...
ExerciseDates)
```

```
Price = 2×1

  100.2800
  100.3655
```

Input Arguments

HWTree — Interest-rate tree structure

binomial tree structure

Interest-rate tree specified as a structure by using `hwtree`.

Data Types: `struct`

`Spread` — Number of basis points over the reference rate
nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (`NINST`)-by-`1`).

Data Types: `double`

`Settle` — Settlement dates of floating-rate note
`ValuationDate` of BDT Tree (default) | datetime array | string array | date character vector

Settlement dates of floating-rate note specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note The `Settle` date for every floating-rate note with an embedded option is set to the `ValuationDate` of the HW tree. The floating-rate note argument `Settle` is ignored.

To support existing code, `optemfloatbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Floating-rate note maturity date
datetime array | string array | date character vector

Floating-rate note maturity date specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optemfloatbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector | cell array of character vectors

Definition of option as `'call'` or `'put'` specified as a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `cell` | `char`

`Strike` — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values specified nonnegative integers using as `NINST`-by-`NSTRIKES` vector of strike price values.

Data Types: `double`

`ExerciseDates` — Exercise date for option (European, Bermuda, or American)
datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American) specified as a `NINST`-by-`NSTRIKES` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors.

To support existing code, `optemfloatbyhw` also accepts serial date numbers as inputs, but they are not recommended.

- If a European or Bermuda option, the 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.
- If an American option, then 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.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree] = optemfloatbyhw(HWTree,Spread,Settle,Maturity,OptSpec,Strike,ExerciseDates,'AmericanOpt',1,'FloatReset',6,'Basis',8)

AmericanOpt — Option type

scalar | vector of positive integers [0,1]

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and NINST-by-1 positive integer scalar flags with values:

- 0 — European/Bermuda
- 1 — American

Data Types: double

FloatReset — Frequency of payments per year

1 (default) | positive integer from the set [1,2,3,4,6,12] | vector of positive integers from the set [1,2,3,4,6,12]

Frequency of payments per year, specified as the comma-separated pair consisting of 'FloatReset' and positive integers for the values [1,2,3,4,6,12] in a NINST-by-1 vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there will be more than one possible path for connecting the two payment dates.

Data Types: double

Basis — Day-count basis of the instrument

0 (actual/actual) (default) | positive integers of the set [1...13] | vector of positive integers of the set [1...13]

Day-count basis of the instrument, specified as the comma-separated pair consisting of 'Basis' and a positive integer using a NINST-by-1 vector. The Basis value represents the basis used when annualizing the input forward-rate tree.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`Principal` — Principal values
`100` (default) | vector of nonnegative values | cell array of nonnegative values

Principal values, specified as the comma-separated pair consisting of `'Principal'` and nonnegative values using a `NINST`-by-`1` vector or `NINST`-by-`1` cell array of notional principal amounts. When using a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of the floating-rate note option at time 0

scalar | vector

Expected prices of the floating-rate note option at time 0 are returned as a scalar or an `NINST`-by-`1` vector.

`PriceTree` — Structure of trees containing vectors of option prices at each node

tree structure

Structure of trees containing vectors of instrument prices and accrued interest and a vector of observation times for each node returned as:

- `PriceTree.PTree` contains embedded option prices.
- `PriceTree.tObs` contains the observation times.

More About

Floating-Rate Note with Embedded Options

A floating-rate note with an embedded option enables floating-rate notes to have early redemption features.

A FRN with an embedded option gives investors or issuers the option to retire the outstanding principal prior to maturity. An embedded call option gives the right to retire the note prior to the maturity date (callable floater), and an embedded put option gives the right to sell the note back at a specific price (puttable floater).

For more information, see “Floating-Rate Note with Embedded Options” on page 2-11.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optemfloatbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`optembndbyhw` | `optemfloatbyhjm` | `optemfloatbybdt` | `optemfloatbybk` | `instoptemfloat` | `OptionEmbeddedFloatBond`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Floating-Rate Note with Embedded Options” on page 2-11
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optfloatbybdt

Price options on floating-rate notes for Black-Derman-Toy interest-rate tree

Syntax

```
[Price,PriceTree] = optfloatbybdt(BDTTree,OptSpec,Strike,ExerciseDates,
AmericanOpt,Spread,Settle,Maturity)
[Price,PriceTree] = optfloatbybdt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optfloatbybdt(BDTTree,OptSpec,Strike,ExerciseDates, AmericanOpt,Spread,Settle,Maturity)` prices options on floating-rate notes from a Black-Derman-Toy interest rate tree. `optfloatbybdt` computes prices of options on vanilla floating-rate notes.

Note Alternatively, you can use the `FloatBondOption` object to price floating-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optfloatbybdt( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of American Call and Put Options on a Floating-Rate Note

Define the interest-rate term structure.

```
Rates = [0.03;0.034;0.038;0.04];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
```

Create the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
```

```
       StartDates: 734869
    ValuationDate: 734869
            Basis: 0
     EndMonthRule: 1
```

Build the BDT tree and assume a volatility of 10%.

```
Sigma = 0.1;
BDTTimeSpec = bdttimespec(ValuationDate, EndDates);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec)
```

```
BDTT = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734869 735235 735600 735965]
        TFwd: {[4x1 double]  [3x1 double]  [2x1 double]  [3]}
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
     FwdTree: {[1.0300]  [1.0342 1.0418]  [1.0374 1.0456 1.0558]  [1.0337 ... ]}
```

The floater instrument has a spread of 10, a period of one year, and matures on Jan-1-2016.

```
Spread = 10;
Settle = datetime(2012,1,1);
Maturity =  datetime(2016,1,1);
Period = 1;
```

Define the option for the floating-rate note.

```
OptSpec = {'call'; 'put'};
Strike = [100;101];
ExerciseDates = datetime(2015,1,1);
AmericanOpt = 1;
```

Compute the price of the call and put options.

```
Price= optfloatbybdt(BDTT,  OptSpec, Strike, ExerciseDates,AmericanOpt, Spread,...
Settle, Maturity)
```

```
Price = 2×1

    0.3655
    0.8087
```

Input Arguments

BDTTree — Interest-rate tree structure

binomial tree structure

Interest-rate tree specified as a structure by using `bdttree`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector | cell array of character vectors

Definition of option as `'call'` or `'put'` specified as a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `cell` | `char`

`Strike` — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values specified nonnegative integers using as `NINST`-by-`NSTRIKES` vector of strike price values.

Data Types: `double`

`ExerciseDates` — Exercise date for option (European, Bermuda, or American)
datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American) specified as a `NINST`-by-`NSTRIKES` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors.

To support existing code, `optfloatbybdt` also accepts serial date numbers as inputs, but they are not recommended.

- If a European or Bermuda option, the `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.
- If an American option, then `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`.

`AmericanOpt` — Option type
scalar | vector of positive integers `[0,1]`

Option type specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`Spread` — Number of basis points over the reference rate
nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (`NINST`)-by-`1`).

Data Types: `single` | `double`

`Settle` — Settlement dates of floating-rate note
`ValuationDate` of BDT Tree (default) | datetime array | string array | date character vector

Settlement dates of floating-rate note specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note 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.

To support existing code, `optfloatbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Floating-rate note maturity date
datetime array | string array | date character vector

Floating-rate note maturity date specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optfloatbybdt` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
`[Price,PriceTree]=optfloatbybdt(BDTTree,OptSpec,Strike,ExerciseDates,AmericanOpt,Spread,Settle,Maturity,'FloatReset',4,'Basis',7)`

`FloatReset` — Frequency of payments per year
`1` (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of payments per year, specified as the comma-separated pair consisting of `'FloatReset'` and positive integers for the values `[1,2,3,4,6,12]` in a `NINST`-by-`1` vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there will be more than one possible path for connecting the two payment dates.

Data Types: `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `NINST`-by-`1` vector. The `Basis` value represents the basis used when annualizing the input forward-rate tree.

- 0 = actual/actual
- 1 = 30/360 (SIA)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal values
`100` (default) | vector of nonnegative values | cell array of nonnegative values

Principal values, specified as the comma-separated pair consisting of `'Principal'` and nonnegative values using a `NINST`-by-`1` vector or `NINST`-by-`1` cell array of notional principal amounts. When using a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

Output Arguments

`Price` — Expected prices of the floating-rate note option at time 0

scalar | vector

Expected prices of the floating-rate note option at time 0 is returned as a scalar or an `NINST`-by-1 vector.

`PriceTree` — Structure of trees containing vectors of option prices at each node

tree structure

Structure of trees containing vectors of instrument prices and accrued interest and a vector of observation times for each node returned as:

- `PriceTree.PTree` contains option prices.
- `PriceTree.tObs` contains the observation times.

More About

Floating-Rate Note Options

A floating-rate note option is a put or call option on a floating-rate note.

Financial Instruments Toolbox supports three types of put and call options on floating-rate notes:

- 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 only exercise it on predetermined dates, usually monthly.

For more information, see “Floating-Rate Note Options” on page 2-11.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optfloatbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdttree` | `cfbybdt` | `capbybdt` | `swapbybdt` | `floorbybdt` | `floatbybdt` | `bondbybdt` | `instoptfloat` | `FloatBondOption`

Topics

“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Floating-Rate Note Options” on page 2-11
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optfloatbybk

Price options on floating-rate notes for Black-Karasinski interest-rate tree

Syntax

```
[Price,PriceTree] = optfloatbybdt(BKTree,OptSpec,Strike,ExerciseDates,
AmericanOpt,Spread,Settle,Maturity)
[Price,PriceTree] = optfloatbybdt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optfloatbybdt(BKTree,OptSpec,Strike,ExerciseDates, AmericanOpt,Spread,Settle,Maturity)` prices options on floating-rate notes from a Black-Karasinski interest rate tree. `optfloatbybk` computes prices of options on vanilla floating-rate notes.

Note Alternatively, you can use the `FloatBondOption` object to price floating-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optfloatbybdt( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of American and European Call Options on a Floating-Rate Note

Define the interest-rate term structure.

```
Rates = [0.03;0.034;0.038;0.04];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
```

Create the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
```

```
       StartDates: 734869
    ValuationDate: 734869
            Basis: 0
     EndMonthRule: 1
```

Build the BK tree.

```
VolDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2016,1,1);
AlphaCurve = 0.1;

BKVolSpec = bkvolspec(RateSpec.ValuationDate,VolDates,VolCurve,...
AlphaDates,AlphaCurve);
BKTimeSpec = bktimespec(RateSpec.ValuationDate,VolDates,Compounding);
BKT = bktree(BKVolSpec,RateSpec,BKTimeSpec)
```

```
BKT = struct with fields:
      FinObj: 'BKFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734869 735235 735600 735965]
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
     FwdTree: {1x4 cell}
```

The floater instrument has a spread of 10, a period of one year, and matures on Jan-1-2016.

```
Spread = 10;
Settle = datetime(2012,1,1);
Maturity =  datetime(2016,1,1);
Period = 1;
```

Define the option for the floating-rate note.

```
OptSpec = {'call'};
Strike = 95;
ExerciseDates = datetime(2016,1,1);
AmericanOpt = [0;1];
```

Compute the price of the call options.

```
Price = optfloatbybk(BKT,OptSpec,Strike,ExerciseDates,AmericanOpt,...
Spread,Settle,Maturity)
```

```
Price = 2×1

    4.2740
    5.3655
```

Input Arguments

`BKTree` — Interest-rate tree structure

binomial tree structure

Interest-rate tree specified as a structure by using `bktree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector | cell array of character vectors

Definition of option as `'call'` or `'put'` specified as a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `cell` | `char`

`Strike` — Option strike price values

nonnegative integer | vector of nonnegative integers

Option strike price values specified nonnegative integers using as `NINST`-by-`NSTRIKES` vector of strike price values.

Data Types: `double`

`ExerciseDates` — Exercise date for option (European, Bermuda, or American)

datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American) specified as a `NINST`-by-`NSTRIKES` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors.

To support existing code, `optfloatbybk` also accepts serial date numbers as inputs, but they are not recommended.

- If a European or Bermuda option, the `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.
- If an American option, then `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`.

`AmericanOpt` — Option type

scalar | vector of positive integers`[0,1]`

Option type specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`Spread` — Number of basis points over the reference rate

nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (NINST)-by-1).

Data Types: single | double

Settle — Settlement dates of floating-rate note
ValuationDate of BK tree (default) | datetime array | string array | date character vector

Settlement dates of floating-rate note specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

Note 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.

To support existing code, optfloatbybk also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Floating-rate note maturity date
datetime array | string array | date character vector

Floating-rate note maturity date specified a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optfloatbybk also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,PriceTree] = optfloatbybk(BKTree,OptSpec,Strike,ExerciseDates,AmericanOpt,Spread,Settle,Maturity,'FloatReset',4,'Basis',7)

FloatReset — Frequency of payments per year
1 (default) | positive integer from the set[1,2,3,4,6,12] | vector of positive integers from the set [1,2,3,4,6,12]

Frequency of payments per year, specified as the comma-separated pair consisting of 'FloatReset' and positive integers for the values [1,2,3,4,6,12] in a NINST-by-1 vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there will be more than one possible path for connecting the two payment dates.

Data Types: double

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `NINST`-by-`1` vector. The `Basis` value represents the basis used when annualizing the input forward-rate tree.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal values
`100` (default) | vector of nonnegative values | cell array of nonnegative values

Principal values, specified as the comma-separated pair consisting of `'Principal'` and nonnegative values using a `NINST`-by-`1` vector or `NINST`-by-`1` cell array of notional principal amounts. When using a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

Output Arguments

`Price` — Expected prices of the floating-rate note option at time 0

scalar | vector

Expected prices of the floating-rate note option at time 0 is returned as a scalar or an `NINST`-by-`1` vector.

`PriceTree` — Structure of trees containing vectors of option prices at each node

tree structure

Structure of trees containing vectors of instrument prices and accrued interest and a vector of observation times for each node returned as:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.AITree` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

Floating-Rate Note Options

A floating-rate note option is a put or call option on a floating-rate note.

Financial Instruments Toolbox supports three types of put and call options on floating-rate notes:

- 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 only exercise it on predetermined dates, usually monthly.

For more information, see “Floating-Rate Note Options” on page 2-11.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optfloatbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bktree` | `cfbybk` | `capbybk` | `swapbybk` | `floorbybk` | `floatbybk` | `bondbybk` | `instoptfloat` | `FloatBondOption`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Floating-Rate Note Options” on page 2-11
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optfloatbycir

Price options on floating-rate notes for Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = optfloatbycir(CIRTree,OptSpec,Strike,ExerciseDates,
AmericanOpt,Spread,Settle,Maturity)
[Price,PriceTree] = optfloatbycir( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optfloatbycir(CIRTree,OptSpec,Strike,ExerciseDates, AmericanOpt,Spread,Settle,Maturity)` prices options on floating-rate notes from a Cox-Ingersoll-Ross (CIR) interest-rate tree. `optfloatbycir` computes prices of options on vanilla floating-rate notes using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = optfloatbycir( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of American and European Call Options on a Floating-Rate Note Using a CIR Interest-Rate Tree

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = datetime(2017,1,1);
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',EndD
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = datetime(2017,1,1);
Maturity = datetime(2021,1,1);
CIRTimeSpec = cirtimespec(Settle, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
```

```
        tObs: [0 1 2 3]
        dObs: [736696 737061 737426 737791]
     FwdTree: {1x4 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

The floater instrument has a spread of 10, a period of one year, and matures on Jan-1-2018.

```
Spread = 10;
Settle = datetime(2017,1,1);
Maturity = datetime(2019,1,1);
Period = 1;
```

Define the option for the floating-rate note.

```
OptSpec = {'call'};
Strike = 95;
ExerciseDates = datetime(2018,1,1);
AmericanOpt = [0;1];
```

Compute the price of the call options.

```
[Price,PriceTree] = optfloatbycir(CIRT, OptSpec,Strike,ExerciseDates,AmericanOpt,...
Spread, Settle, Maturity)
```

```
Price = 2×1

    4.9230
    5.1887


PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
      PTree: {1x5 cell}
     AITree: {1x5 cell}
       tObs: [0 1 2 3 4]
    Connect: {[3x1 double]  [3x3 double]  [3x5 double]}
      Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree specified as a structure by using `cirtree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vectors with values of `'call'` or `'put'` | string array with values `'call'` or `'put'`

Definition of option, specified as a `NINST`-by-1 cell array of character vectors or string arrays with values of `'call'` or `'put'`.

Data Types: cell | char | string

Strike — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values, specified as nonnegative integers using an NINST-by-NSTRIKES vector of strike price values.

Data Types: double

ExerciseDates — Exercise date for option (European, Bermuda, or American)
datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American) specified as a NINST-by-NSTRIKES or NINST-by-2 vector using a datetime array, string array, or date character vectors.

To support existing code, optfloatbycir also accepts serial date numbers as inputs, but they are not recommended.

- For a European or Bermuda option, the 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, the 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.

AmericanOpt — Option type
scalar | vector of positive integers[0,1]

Option type specified as NINST-by-1 positive integer scalar flags with values:

- 0 — European/Bermuda
- 1 — American

Data Types: double

Spread — Number of basis points over the reference rate
nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (NINST)-by-1).

Data Types: double

Settle — Settlement dates of floating-rate note
ValuationDate of CIR tree (default) | datetime array | string array | date character vector

Settlement dates of floating-rate note specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optfloatbycir` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Floating-rate note maturity date
datetime array | string array | date character vector

Floating-rate note maturity date specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optfloatbycir` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = optfloatbybk(CIRTree,OptSpec,Strike,ExerciseDates,AmericanOpt,Spread,Settle,Maturity,'FloatReset',4,'Basis',7)`

`FloatReset` — Frequency of payments per year
`1` (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of payments per year, specified as the comma-separated pair consisting of `'FloatReset'` and positive integers for the values `[1,2,3,4,6,12]` in a `NINST`-by-`1` vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there will be more than one possible path for connecting the two payment dates.

Data Types: `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `NINST`-by-`1` vector. The `Basis` value represents the basis used when annualizing the input forward-rate tree.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal values
`100` (default) | vector of nonnegative values | cell array of nonnegative values

Principal values, specified as the comma-separated pair consisting of `'Principal'` and nonnegative values using a `NINST`-by-`1` vector or `NINST`-by-`1` cell array of notional principal amounts.

When using a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array where the first column is dates, and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and the output from `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

Output Arguments

`Price` — Expected prices of the floating-rate note option at time 0

scalar | vector

Expected prices of the floating-rate note option at time 0 is returned as a scalar or an `NINST`-by-`1` vector.

`PriceTree` — Structure of trees containing vectors of option prices at each node

tree structure

Structure of trees containing vectors of instrument prices and accrued interest and a vector of observation times for each node returned as:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.AITree` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

Floating-Rate Note Options

A floating-rate note option is a put or call option on a floating-rate note.

Financial Instruments Toolbox supports three types of put and call options on floating-rate notes:

- 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 only exercise it on predetermined dates, usually monthly.

For more information, see “Floating-Rate Note Options” on page 2-11.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optfloatbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J., and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `swaptionbycir` | `instoptfloat`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Floating-Rate Note Options” on page 2-11
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

optfloatbyhjm

Price options on floating-rate notes for Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree] = optfloatbyhjm(HJMTree,OptSpec,Strike,ExerciseDates,
AmericanOpt,Spread,Settle,Maturity)
[Price,PriceTree] = optfloatbyhjm( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optfloatbyhjm(HJMTree,OptSpec,Strike,ExerciseDates, AmericanOpt,Spread,Settle,Maturity)` prices options on floating-rate notes from a Heath-Jarrow-Morton interest rate tree. `optfloatbyhjm` computes prices of options on vanilla floating-rate notes.

`[Price,PriceTree] = optfloatbyhjm( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of American and European Call Options on a Floating-Rate Note

Define the interest-rate term structure.

```
Rates = [0.03;0.035;0.040;0.045];
ValuationDate = datetime(2012,1,1);
StartDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)]
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
```

Create the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
          EndTimes: [4x1 double]
        StartTimes: [4x1 double]
          EndDates: [4x1 double]
        StartDates: [4x1 double]
    ValuationDate: 734869
            Basis: 0
     EndMonthRule: 1
```

Build the HJM tree.

```
VolSpec = hjmvolspec('Constant', 0.01);
TimeSpec = hjmtimespec(RateSpec.ValuationDate, EndDates, Compounding);
HJMTree = hjmtree(VolSpec, RateSpec, TimeSpec)
```

```
HJMTree = struct with fields:
      FinObj: 'HJMFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734869 735235 735600 735965]
        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]}
```

The floater instrument has a spread of 10, a period of one year, and matures on Jan-1-2015.

```
Spread = 10;
Settle = datetime(2012,1,1);
Maturity = datetime(2015,1,1);
Period = 1;
```

Define the option for the floating-rate note.

```
OptSpec = {'call'};
Strike = 95;
ExerciseDates = datetime(2015,1,1);
AmericanOpt = [0;1];
```

Compute the price of the call options.

```
Price= optfloatbyhjm(HJMTree,  OptSpec, Strike, ExerciseDates,AmericanOpt,...
Spread, Settle, Maturity)
```

```
Price = 2×1

    4.5098
    5.2811
```

Input Arguments

`HJMTree` — Interest-rate tree structure

binomial tree structure

Interest-rate tree specified as a structure by using `hjmtree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector | cell array of character vectors

Definition of option as `'call'` or `'put'` specified as a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `cell` | `char`

`Strike` — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values specified nonnegative integers using as `NINST`-by-`NSTRIKES` vector of strike price values.

Data Types: `double`

`ExerciseDates` — Exercise date for option (European, Bermuda, or American)
datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American) specified as a `NINST`-by-`NSTRIKES` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors.

To support existing code, `optfloatbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

- If a European or Bermuda option, the `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.
- If an American option, then `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`.

`AmericanOpt` — Option type
scalar | vector of positive integers `[0,1]`

Option type specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`Spread` — Number of basis points over the reference rate
nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (`NINST`)-by-`1`).

Data Types: `double`

`Settle` — Settlement dates of floating-rate note
`ValuationDate` of HJM tree (default) | datetime array | string array | date character vector

Settlement dates of floating-rate note specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note 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.

To support existing code, `optfloatbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Floating-rate note maturity date

datetime array | string array | date character vector

Floating-rate note maturity date specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optfloatbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = optfloatbyhjm(HJMTree,OptSpec,Strike,ExerciseDates,AmericanOpt,Spread,Settle, Maturity,'FloatReset',4,'Basis',7)`

`FloatReset` — Frequency of payments per year

`1` (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of payments per year, specified as the comma-separated pair consisting of `'FloatReset'` and positive integers for the values `[1,2,3,4,6,12]` in a `NINST`-by-`1` vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there will be more than one possible path for connecting the two payment dates.

Data Types: `double`

`Basis` — Day-count basis of the instrument

`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `NINST`-by-`1` vector. The `Basis` value represents the basis used when annualizing the input forward-rate tree.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal values
`100` (default) | vector of nonnegative values | cell array of nonnegative values

Principal values, specified as the comma-separated pair consisting of `'Principal'` and nonnegative values using a `NINST`-by-`1` vector or `NINST`-by-`1` cell array of notional principal amounts. When using a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

Output Arguments

`Price` — Expected prices of the floating-rate note option at time 0
scalar | vector

Expected prices of the floating-rate note option at time 0 is returned as a scalar or an `NINST`-by-`1` vector.

`PriceTree` — Structure of trees containing vectors of option prices at each node
tree structure

Structure of trees containing vectors of instrument prices and accrued interest and a vector of observation times for each node returned as:

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

More About

Floating-Rate Note Options

A floating-rate note option is a put or call option on a floating-rate note.

Financial Instruments Toolbox supports three types of put and call options on floating-rate notes:

- 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 only exercise it on predetermined dates, usually monthly.

For more information, see “Floating-Rate Note Options” on page 2-11.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optfloatbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmtree` | `cfbyhjm` | `capbyhjm` | `swapbyhjm` | `floorbyhw` | `floatbyhjm` | `bondbyhjm` | `instoptfloat`

Topics

“Computing Instrument Prices” on page 2-81

“Floating-Rate Note Options” on page 2-11
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

optfloatbyhw

Price options on floating-rate notes for Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = optfloatbyhw(HWTree,OptSpec,Strike,ExerciseDates,
AmericanOpt,Spread,Settle,Maturity)
[Price,PriceTree] = optfloatbyhw( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optfloatbyhw(HWTree,OptSpec,Strike,ExerciseDates, AmericanOpt,Spread,Settle,Maturity)` prices options on floating-rate notes from a Hull-White interest rate tree. `optfloatbyhw` computes prices of options on vanilla floating-rate notes.

Note Alternatively, you can use the `FloatBondOption` object to price floating-rate bond option instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optfloatbyhw( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of American and European Call Options on a Floating-Rate Note

Define the interest-rate term structure.

```
Rates = [0.03;0.034;0.038;0.04];
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
Compounding = 1;
```

Create the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [4x1 double]
            Rates: [4x1 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
       StartDates: 734869
    ValuationDate: 734869
```

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

Build the HW tree using the following:

```
VolDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2016,1,1);
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
            AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RateSpec, HWTimeSpec)
```

```
HWT = struct with fields:
      FinObj: 'HWFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3]
        dObs: [734869 735235 735600 735965]
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]}
     Connect: {[2]  [2 3 4]  [2 3 4 5 6]}
     FwdTree: {1x4 cell}
```

The floater instrument has a spread of 10, a period of one year, and matures on Jan-1-2016.

```
Spread = 10;
Settle = datetime(2012,1,1);
Maturity =  datetime(2016,1,1);
Period = 1;
```

Define the option for the floating-rate note.

```
OptSpec = {'call'};
Strike = 95;
ExerciseDates = datetime(2016,1,1);
AmericanOpt = [0;1];
```

Compute the price of the call options.

```
Price= optfloatbyhw(HWT,  OptSpec, Strike, ExerciseDates,AmericanOpt,...
Spread, Settle, Maturity)
```

```
Price = 2×1

    4.2740
    5.3655
```

Input Arguments

HWTree — Interest-rate tree structure

binomial tree structure

Interest-rate tree specified as a structure by using `hwtree`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector | cell array of character vectors

Definition of option as `'call'` or `'put'` specified as a `NINST`-by-`1` cell array of character vectors for `'call'` or `'put'`.

Data Types: `cell` | `char`

`Strike` — Option strike price values
nonnegative integer | vector of nonnegative integers

Option strike price values specified nonnegative integers using as `NINST`-by-`NSTRIKES` vector of strike price values.

Data Types: `double`

`ExerciseDates` — Exercise date for option (European, Bermuda, or American)
datetime array | string array | date character vector

Exercise date for option (European, Bermuda, or American) specified as a `NINST`-by-`NSTRIKES` or `NINST`-by-`2` vector using a datetime array, string array, or date character vectors.

To support existing code, `optfloatbyhw` also accepts serial date numbers as inputs, but they are not recommended.

- If a European or Bermuda option, the `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.
- If an American option, then `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`.

`AmericanOpt` — Option type
scalar | vector of positive integers `[0,1]`

Option type specified as `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

`Spread` — Number of basis points over the reference rate
nonnegative integer | vector of nonnegative integers

Number of basis points over the reference rate specified as a vector of nonnegative integers for the number of instruments (`NINST`)-by-`1`).

Data Types: `double`

`Settle` — Settlement dates of floating-rate note
`ValuationDate` of HW tree (default) | datetime array | string array | date character vector

Settlement dates of floating-rate note specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note 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.

To support existing code, `optfloatbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Floating-rate note maturity date
datetime array | string array | date character vector

Floating-rate note maturity date specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optfloatbyhw` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
`[Price,PriceTree]=optfloatbyhw(HWTree,OptSpec,Strike,ExerciseDates,AmericanOpt,Spread,Settle,Maturity,'FloatReset',4,'Basis',7)`

`FloatReset` — Frequency of payments per year
`1` (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of payments per year, specified as the comma-separated pair consisting of `'FloatReset'` and positive integers for the values `[1,2,3,4,6,12]` in a `NINST`-by-`1` vector.

Note Payments on floating-rate notes (FRNs) are determined by the effective interest-rate between reset dates. If the reset period for an FRN spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there will be more than one possible path for connecting the two payment dates.

Data Types: `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `NINST`-by-`1` vector. The `Basis` value represents the basis used when annualizing the input forward-rate tree.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal values
`100` (default) | vector of nonnegative values | cell array of nonnegative values

Principal values, specified as the comma-separated pair consisting of `'Principal'` and nonnegative values using a `NINST`-by-`1` vector or `NINST`-by-`1` cell array of notional principal amounts. When using a `NINST`-by-`1` cell array, each element is a `NumDates`-by-`2` cell array where the first column is dates and the second column is associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`Options` — Structure containing derivatives pricing options
structure

Structure containing derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer [0,1]

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

Output Arguments

`Price` — Expected prices of the floating-rate note option at time 0

scalar | vector

Expected prices of the floating-rate note option at time 0 is returned as a scalar or an `NINST`-by-`1` vector.

`PriceTree` — Structure of trees containing vectors of option prices at each node

tree structure

Structure of trees containing vectors of instrument prices and accrued interest and a vector of observation times for each node returned as:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.AITree` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

Floating-Rate Note Options

A floating-rate note option is a put or call option on a floating-rate note.

Financial Instruments Toolbox supports three types of put and call options on floating-rate notes:

- 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 only exercise it on predetermined dates, usually monthly.

For more information, see “Floating-Rate Note Options” on page 2-11.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optfloatbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hwtree` | `cfbyhw` | `capbyhw` | `swapbyhw` | `floorbyhw` | `floatbyhw` | `bondbyhw` | `instoptfloat` | `FloatBondOption`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Floating-Rate Note Options” on page 2-11
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

optsensbysabr

Calculate option sensitivities using SABR model

Syntax

```
Sens = optsensbysabr(ZeroCurve,Alpha,Beta,Rho,Nu,Settle,ExerciseDate,
ForwardValue,Strike,OptSpec)
Sens = optsensbysabr( ___ ,Name,Value)
```

Description

`Sens = optsensbysabr(ZeroCurve,Alpha,Beta,Rho,Nu,Settle,ExerciseDate, ForwardValue,Strike,OptSpec)` returns the sensitivities of an option value by using the SABR stochastic volatility model.

`Sens = optsensbysabr( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Calculate the Sensitivity Values for an Interest-Rate Swaption

Define the interest rate and the model parameters.

```
SwapRate = 0.0357;
Strike = 0.03;
Alpha = 0.036;
Beta = 0.5;
Rho = -0.25;
Nu = 0.35;
Rates = 0.05;
```

Define the `Settle`, `ExerciseDate`, and `OptSpec` for an interest-rate swaption.

```
Settle = datetime(2013,9,15);
ExerciseDate = datetime(2015,9,15);
OptSpec = 'call';
```

Define the `RateSpec` for the interest-rate curve.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9048
            Rates: 0.0500
        EndTimes: 2
       StartTimes: 0
         EndDates: 736222
```

```
       StartDates: 735492
    ValuationDate: 735492
            Basis: 1
     EndMonthRule: 1
```

Calculate the `Delta` and `Vega` sensitivity values for the interest-rate swaption.

```
[SABRDelta, SABRVega] = optsensbysabr(RateSpec, Alpha, Beta, Rho, Nu, Settle, ...
ExerciseDate, SwapRate, Strike, OptSpec, 'OutSpec',  {'Delta', 'Vega'})

SABRDelta = 0.7025

SABRVega = 0.0772
```

Calculate the Sensitivity Values for a Swaption Using the Shifted SABR Model

Define the interest rate and the model parameters.

```
SwapRate = 0.0002;
Strike = -0.001;  % -0.1% strike.
Alpha = 0.01;
Beta = 0.5;
Rho = -0.1;
Nu = 0.15;
Shift = 0.005;  % 0.5 percent shift
Rates = 0.0002;
```

Define the `Settle`, `ExerciseDate`, and `OptSpec` for the swaption.

```
Settle = datetime(2016,3,1);
ExerciseDate = datetime(2017,3,1);
OptSpec = 'call';
```

Define the `RateSpec` for the interest-rate curve.

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

RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9998
            Rates: 2.0000e-04
         EndTimes: 1
       StartTimes: 0
         EndDates: 736755
       StartDates: 736390
    ValuationDate: 736390
            Basis: 1
     EndMonthRule: 1
```

Calculate the `Delta` and `Vega` sensitivity values for the swaption.

```
[ShiftedSABRDelta,ShiftedSABRVega] = optsensbysabr(RateSpec, ...
Alpha,Beta,Rho,Nu,Settle,ExerciseDate,SwapRate,Strike,OptSpec, ...
'OutSpec',{'Delta','Vega'},'Shift',Shift)

ShiftedSABRDelta = 0.9628

ShiftedSABRVega = 0.0060
```

Calculate the Sensitivity Values for an Interest-Rate Swaption with Normal (Bachelier) Implied Volatility

This example shows how to use `optsensbysabr` to calculate sensitivities for an interest-rate swaption using the Normal model for the case where the `Beta` parameter is `> 0` and where `Beta = 0`.

For the case where the `Beta` parameter is `> 0`, select the Normal (Bachelier) implied volatility model in `optsensbysabr`, specify the `'Model'` name-value pair to `'normal'`.

```
% Define the interest rate and the model parameters.

SwapRate = 0.025;
Strike = 0.02;
Alpha = 0.044;
Beta = 0.5;
Rho = -0.21;
Nu = 0.31;
Rates = 0.028;

% Define the Settle, ExerciseDate, and OptSpec for the swaption.

Settle = datetime(2018,3,7);
ExerciseDate = datetime(2020,3,7);
OptSpec = 'call';

% Define the RateSpec for the interest-rate curve.

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

% Calculate the Delta and Vega sensitivity values for the swaption. Set the
% 'Model' name-value pair to 'normal' in order to select the Normal
% (Bachelier) implied volatility model.

[SABRDelta, SABRVega] = optsensbysabr(RateSpec, Alpha, Beta, Rho, Nu, ...
    Settle, ExerciseDate, SwapRate, Strike, OptSpec, ...
    'OutSpec',  {'Delta', 'Vega'}, 'Model', 'normal')

SABRDelta = 0.7171

SABRVega = 0.0686
```

Calculate Sensitivities for a Swaption with Normal Implied Volatility Using the Normal SABR Model

When the `Beta` parameter is set to zero, the SABR model becomes the Normal SABR model. Negative interest rates are allowed when the Normal SABR model is used in combination with Normal

(Bachelier) implied volatility. To select the Normal (Bachelier) implied volatility model in `optsensbysabr`, specify the 'Model' name-value pair to 'normal'.

```
% Define the interest rate and the model parameters.

SwapRate = -0.00254;
Strike = -0.002;
Alpha = 0.0047;
Beta = 0;  % Set the Beta parameter to zero to use the Normal SABR model
Rho = -0.20;
Nu = 0.28;
Rates = 0.0001;

% Define the Settle, ExerciseDate, and OptSpec for the swaption.

Settle = datetime(2018,4,11);
ExerciseDate = datetime(2019,4,11);
OptSpec = 'call';

% Define the RateSpec for the interest-rate curve.

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

% Calculate the Delta and Vega sensitivity values for the swaption. Set the
% 'Model' name-value pair to 'normal' in order to select the Normal
% (Bachelier) implied volatility model.

[SABRDelta, SABRVega] = optsensbysabr(RateSpec, Alpha, Beta, Rho, Nu, ...
    Settle, ExerciseDate, SwapRate, Strike, OptSpec, ...
    'OutSpec',  {'Delta', 'Vega'}, 'Model', 'normal')
```

```
SABRDelta = 0.4644
```

```
SABRVega = 0.3987
```

Input Arguments

`ZeroCurve` — Annualized interest-rate term structure for zero-coupon bonds

structure

Annualized interest-rate term structure for zero-coupon bonds, specified by using the `RateSpec` obtained from `intenvset` or an `IRDataCurve` with multiple rates using the `IRDataCurve` constructor.

Data Types: `struct`

`Alpha` — Current SABR volatility

scalar numeric

Current SABR volatility, specified as a scalar numeric.

Data Types: `double`

`Beta` — SABR constant elasticity of variance (CEV) exponent

scalar numeric

SABR CEV exponent, specified as a scalar numeric.

Data Types: `double`

`Rho` — Correlation between forward value and volatility
scalar numeric

Correlation between forward value and volatility, specified as a scalar numeric.

Data Types: `double`

`Nu` — Volatility of volatility
scalar numeric

Volatility of volatility, specified as a scalar numeric.

Data Types: `double`

`Settle` — Settlement date
datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `optsensbysabr` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDate` — Option exercise date
datetime scalar | string scalar | date character vector

Option exercise date, specified as a scalar datetime, string, or date character vector.

To support existing code, `optsensbysabr` also accepts serial date numbers as inputs, but they are not recommended.

`ForwardValue` — Current forward value of underlying asset
scalar numeric | vector

Current forward value of the underlying asset, specified as a scalar numeric or vector of size `NINST`-by-`1`.

Data Types: `double`

`Strike` — Option strike price values
scalar numeric | vector

Option strike price values, specified as a scalar numeric or a vector of size `NINST`-by-`1`.

Data Types: `double`

`OptSpec` — Definition of option
character vector with value of `'call'` or `'put'`

Definition of the option, specified as `'call'` or `'put'` using a character vector.

Data Types: `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ModifiedSABRDelta = optsensbysabr(RateSpec,Alpha,Beta,Rho,Nu,Settle,ExerciseDate,ForwardValue,Strike,OptSpec,'OutSpec','ModifiedDelta')`

`OutSpec` — Sensitivity outputs

`'Delta'` (default) | character vector with values `'Delta'`, `'Vega'`, `'ModifiedDelta'`, `'ModifiedVega'`, `'dSigmadF'`, `'dSigmadAlpha'` | cell array of character vectors with values `'Delta'`, `'Vega'`, `'ModifiedDelta'`, `'ModifiedVega'`, `'dSigmadF'`, `'dSigmadAlpha'`

Sensitivity outputs, specified as the comma-separated pair consisting of `'OutSpec'` and an `NOUT`-by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Delta'`, `'Vega'`, `'ModifiedDelta'`, `'ModifiedVega'`, `'dSigmadF'`, and `'dSigmadAlpha'` where:

- `'Delta'` is SABR Delta by Hagan et al. (2002).
- `'Vega'` is SABR Vega by Hagan et al. (2002).
- `'ModifiedDelta'` is SABR Delta modified by Bartlett (2006).
- `'ModifiedVega'` is SABR Vega modified by Bartlett (2006).
- `'dSigmadF'` is the sensitivity of implied volatility with respect to the underlying current forward value, *F*. The implied volatility type depends on `Shift` and `Model`.
- `'dSigmadAlpha'` is the sensitivity of implied volatility with respect to the `Alpha` parameter. The implied volatility type depends on `Shift` and `Model`.

Example: `OutSpec = {'Delta','Vega','ModifiedDelta','ModifiedVega','dSigmadF','dSigmadAlpha'}`

Data Types: `char` | `cell`

`Shift` — Shift in decimals for shifted SABR model

`0` (no shift) (default) | scalar positive decimal

Shift in decimals for the shifted SABR model (to be used with the Shifted Black model), specified as the comma-separated pair consisting of `'Shift'` and a scalar positive decimal value. Set this parameter to a positive shift in decimals to add a positive shift to `ForwardValue` and `Strike`, which effectively sets a negative lower bound for `ForwardValue` and `Strike`. For example, a `Shift` value of `0.01` is equal to a 1% positive shift.

Note If the `Model` is set to `'normal'`, the `Shift` parameter must be `0`.

Data Types: `double`

`Model` — Model used by the implied volatility sigma

`'lognormal'` (default) | character vector with value `'lognormal'` or `'normal'` | string with value `"lognormal"` or `"normal"`

Model used by the implied volatility sigma, specified as the comma-separated pair consisting of `'Model'` and a character vector with a value of `'lognormal'` or `'normal'`, or a string with a value of `"lognormal"` or `"normal"`.

Note The setting for `Model` affects the interpretation of the implied volatility "sigma". Depending on the setting for `Model`, the "sigma" has the following interpretations:

- If `Model` is `'lognormal'` (default), "sigma" can be either Implied Black (no shift) or Implied Shifted Black volatility.
- If `Model` is `'normal'`, "sigma" is the Implied Normal (Bachelier) volatility and `Shift` must be zero.

Data Types: `char` | `string`

Output Arguments

Sens — Sensitivity values

array

Sensitivity values, returned as an `NINST`-by-`1` array as defined by the `OutSpec`.

Algorithms

In the SABR model, an option with value V is defined by the modified Black formula B, where σ_B is the SABR implied Black volatility.

$$V = B(F, K, T, \sigma_B(\alpha, \beta, \rho, \nu, F, K, T))$$

The `Delta` and `Vega` sensitivities under the SABR model are expressed in terms of partial derivatives in the original paper by Hagan (2002).

$$\text{SABR } Delta = \frac{\partial V}{\partial F} = \frac{\partial B}{\partial F} + \frac{\partial B}{\partial \sigma_B}\frac{\partial \sigma_B}{\partial F}$$

$$\text{SABR } Vega = \frac{\partial V}{\partial \alpha} = \frac{\partial B}{\partial \sigma_B}\frac{\partial \sigma_B}{\partial \alpha}$$

Later, Bartlett (2006) made better use of the model dynamics by incorporating the correlated changes between F and α

$$\text{Modified SABR } Delta = \frac{\partial B}{\partial F} + \frac{\partial B}{\partial \sigma_B}\left(\frac{\partial \sigma_B}{\partial F} + \frac{\partial \sigma_B}{\partial \alpha}\frac{\rho \upsilon}{F^{\beta}}\right)$$

$$\text{Modified SABR } Vega = \frac{\partial B}{\partial \sigma_B}\left(\frac{\partial \sigma_B}{\partial \alpha} + \frac{\partial \sigma_B}{\partial F}\frac{\rho F^{\beta}}{\upsilon}\right)$$

where $\frac{\partial B}{\partial F}$ is the classic Black `Delta` and $\frac{\partial B}{\partial \sigma_B}$ is the classic Black `Vega`. The Black implied volatility σ_B is computed internally by calling `blackvolbysabr`, while its partial derivatives $\frac{\partial \sigma_B}{\partial F}$ and $\frac{\partial \sigma_B}{\partial \alpha}$ are computed using closed-form expressions by `optsensbysabr`.

Similar expressions apply to the implied Normal volatility σ_N. For more information, see `normalvolbysabr`.

Version History

Introduced in R2014b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optsensbysabr` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hagan, P. S., D. Kumar. A. S. Lesniewski, and D. E. Woodward. *"Managing Smile Risk."* Wilmott Magazine, 2002.

[2] Bartlett, B. *"Hedging under SABR Model."* Wilmott Magazine, 2006.

See Also

`blackvolbysabr` | `normalvolbysabr` | `intenvset` | `toRateSpec` | `IRDataCurve`

Topics

"Calibrate the SABR Model" on page 2-33
"Price a Swaption Using the SABR Model" on page 2-38
"Price Swaptions with Negative Strikes Using the Shifted SABR Model" on page 2-26
"Work with Negative Interest Rates Using Functions" on page 2-18
"Supported Interest-Rate Instrument Functions" on page 2-3

External Websites

How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

optstockbybaw

Calculate American options prices using Barone-Adesi and Whaley option pricing model

Syntax

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

Description

`Price = optstockbybaw(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike)` calculates American options prices using the Barone-Adesi and Whaley option pricing model.

Examples

Compute American Option Prices Using the Barone-Adesi and Whaley Option Pricing Model

Consider an American call option with an exercise price of $120. The option expires on Jan 1, 2018. The stock has a volatility of 14% per annum, and the annualized continuously compounded risk-free rate is 4% per annum as of Jan 1, 2016. Using this data, calculate the price of the American call, assuming the price of the stock is $125 and pays a dividend of 2%.

```
StartDate  = datetime(2016,1,1);
EndDate = datetime(2018,1,1);
Basis = 1;
Compounding = -1;
Rates = 0.04;
```

Define the `RateSpec`.

```
RateSpec = intenvset('ValuationDate',StartDate,'StartDate',StartDate,'EndDate',EndDate, ...
'Rates',Rates,'Basis',Basis,'Compounding',Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9231
            Rates: 0.0400
         EndTimes: 2
       StartTimes: 0
         EndDates: 737061
       StartDates: 736330
    ValuationDate: 736330
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec`.

```
Dividend = 0.02;
AssetPrice = 125;
Volatility = 0.14;
```

```
StockSpec = stockspec(Volatility,AssetPrice,'Continuous',Dividend)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1400
         AssetPrice: 125
       DividendType: {'continuous'}
    DividendAmounts: 0.0200
    ExDividendDates: []
```

Define the American option.

```
OptSpec = 'call';
Strike = 120;
Settle = datetime(2016,1,1);
Maturity = datetime(2018,1,1);
```

Compute the price for the American option.

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

```
Price = 14.5180
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement date

datetime array | string array | date character vector

Settlement date for the American option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbybaw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date for the American option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbybaw` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'` | string array with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors or string arrays with values `'call'` or `'put'`.

Data Types: `char` | `cell` | `string`

`Strike` — American option strike price value
nonnegative scalar | nonnegative vector

American Option strike price value, specified as a nonnegative scalar or `NINST`-by-`1` matrix of strike price values. Each row is the schedule for one option.

Data Types: `single` | `double`

Output Arguments

`Price` — Expected prices for American options
vector

Expected prices for American options, returned as a `NINST`-by-`1` vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2017a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbybaw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Barone-Aclesi, G. and Robert E. Whaley. “Efficient Analytic Approximation of American Option Values.” *The Journal of Finance*. Volume 42, Issue 2 (June 1987), 301–320.

[2] Haug, E. *The Complete Guide to Option Pricing Formulas. Second Edition*. McGraw-Hill Education, January 2007.

See Also

`optstocksensbybaw` | `impvbybaw`

Topics

“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstocksensbybaw

Calculate American options prices and sensitivities using Barone-Adesi and Whaley option pricing model

Syntax

```
PriceSens = optstocksensbybaw(RateSpec,StockSpec,Settle,Maturity,OptSpec,
Strike)
PriceSens = optstocksensbybaw( ___ ,Name,Value)
```

Description

`PriceSens = optstocksensbybaw(RateSpec,StockSpec,Settle,Maturity,OptSpec, Strike)` calculates American options prices using the Barone-Adesi and Whaley option pricing model.

`PriceSens = optstocksensbybaw( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute an American Option Price and Sensitivities Using the Barone-Adesi and Whaley Option Pricing Model

Consider an American call option with an exercise price of $120. The option expires on Jan 1, 2018. The stock has a volatility of 14% per annum, and the annualized continuously compounded risk-free rate is 4% per annum as of Jan 1, 2016. Using this data, calculate the price of the American call, assuming the price of the stock is $125 and pays a dividend of 2%.

```
StartDate  = datetime(2016,1,1);
EndDate = datetime(2018,1,1);
Basis = 1;
Compounding = -1;
Rates = 0.04;
```

Define the `RateSpec`.

```
RateSpec = intenvset('ValuationDate',StartDate,'StartDate',StartDate,'EndDate',EndDate, ...
'Rates',Rates,'Basis',Basis,'Compounding',Compounding)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9231
            Rates: 0.0400
         EndTimes: 2
       StartTimes: 0
         EndDates: 737061
       StartDates: 736330
    ValuationDate: 736330
```

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

Define the `StockSpec`.

```
Dividend = 0.02;
AssetPrice = 125;
Volatility = 0.14;

StockSpec = stockspec(Volatility,AssetPrice,'Continuous',Dividend)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1400
         AssetPrice: 125
       DividendType: {'continuous'}
    DividendAmounts: 0.0200
    ExDividendDates: []
```

Define the American option.

```
OptSpec = 'call';
Strike = 120;
Settle = datetime(2016,1,1);
Maturity = datetime(2018,1,1);
```

Compute the price and sensitivities for the American option.

```
OutSpec = {'price';'delta';'theta'};

[Price,Delta,Theta] = optstocksensbybaw(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,'OutSpe
```

```
Price = 14.5180

Delta = 0.6672

Theta = -3.1861
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date for the American option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstocksensbybaw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date for the American option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstocksensbybaw` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | string array with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors or string arrays with values `'call'` or `'put'`.

Data Types: `char` | `string`

`Strike` — American option strike price value
nonnegative scalar | nonnegative vector

American option strike price value, specified as a nonnegative scalar or `NINST`-by-`1` matrix of strike price values. Each row is the schedule for one option.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,Delta,Theta] = optstocksensbybaw(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,'OutSpec',OutSpec)`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities for American options
matrix

Expected prices or sensitivities for American options, returned as a `NINST`-by-`1` matrix.

Note All sensitivities are evaluated by computing a discrete approximation of the partial derivative. This means that the option is revalued with a fractional change for each relevant parameter. The change in the option value divided by the increment is the approximated sensitivity value.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2017a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `optstocksensbybaw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Barone-Aclesi, G. and Robert E. Whaley. “Efficient Analytic Approximation of American Option Values.” *The Journal of Finance.* Volume 42, Issue 2 (June 1987), 301–320.

[2] Haug, E. *The Complete Guide to Option Pricing Formulas. Second Edition.* McGraw-Hill Education, January 2007.

See Also

`optstockbybaw` | `impvbybaw`

Topics

“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19

optstockbybjs

Price American options using Bjerksund-Stensland 2002 option pricing model

Syntax

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

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.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Compute the American Option Prices With Continuous Dividend Yield Using the Bjerksund-Stensland 2002 Option Pricing Model

This example shows how to compute the American option prices with continuous dividend yield using the Bjerksund-Stensland 2002 option pricing model. 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 = datetime(2008,1,1);
Maturity = datetime(2008,4,1);
Strike = 100;
AssetPrice = [90;120];
DivYield = 0.04;
Rate = 0.08;
Sigma = 0.20;

% define the 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'; 'put'};

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

```
Price = 2×1

    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).

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbybjs` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for option

datetime array | string array | date character vector

Maturity date for option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbybjs` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option

cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price value
nonnegative vector

Option strike price value, specified as a nonnegative `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected option prices
vector

Expected option prices, returned as a `NINST`-by-`1` vector.

Data Types: `double`

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2008b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `optstockbybjs` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bjerksund, P. and G. Stensland. “Closed-Form Approximation of American Options.” *Scandinavian Journal of Management.* Vol. 9, 1993, Suppl., pp. S88–S99.

[2] Bjerksund, P. and G. Stensland. *“Closed Form Valuation of American Options.”* Discussion paper, 2002.

See Also

`intenvset` | `toRateSpec` | `IRDataCurve` | `stockspec` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstockbyblk

Price options on futures and forwards using Black option pricing model

Syntax

```
Price = optstockbyblk(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike)
Price = optstockbyblk( ___ ,Name,Value)
```

Description

`Price = optstockbyblk(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike)` computes option prices on futures or forward using the Black option pricing model.

Note `optstockbyblk` calculates option prices on futures and forwards. If `ForwardMaturity` is not passed, the function calculates prices of future options. If `ForwardMaturity` is passed, the function computes prices of forward options. This function handles several types of underlying assets, for example, stocks and commodities. For more information on the underlying asset specification, see `stockspec`.

`Price = optstockbyblk( ___ ,Name,Value)` adds an optional name-value pair argument for `ForwardMaturity` to compute option prices on forwards using the Black option pricing model.

Examples

Compute Option Prices on Futures Using the Black Option Pricing Model

This example shows how to compute option prices on futures using the Black option pricing 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 = datetime(2008,5,1);
Maturity = datetime(2008,9,1);

% define the 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'};
```

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

```
Price = 2×1

    1.5903
    0.3037
```

Compute Option Prices on a Forward

This example shows how to compute option prices on forwards using the Black pricing model. Consider two European options, a call and put on the Brent Blend forward contract that expires on January 1, 2015. The options expire on October 1, 2014 with an exercise price of $200 and $90 respectively. Assume that on January 1, 2014 the forward price is at $107, the annualized continuously compounded risk-free rate is 3% per annum and volatility is 28% per annum. Using this data, compute the price of the options.

Define the `RateSpec`.

```
ValuationDate = datetime(2014,1,1);
EndDates = datetime(2015,1,1);
Rates = 0.03;
Compounding = -1;
Basis = 1;
RateSpec  = intenvset('ValuationDate', ValuationDate, ...
'StartDates', ValuationDate, 'EndDates', EndDates, 'Rates', Rates,....
'Compounding', Compounding, 'Basis', Basis')
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9704
            Rates: 0.0300
         EndTimes: 1
       StartTimes: 0
         EndDates: 735965
       StartDates: 735600
    ValuationDate: 735600
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec`.

```
AssetPrice = 107;
Sigma = 0.28;
StockSpec  = stockspec(Sigma, AssetPrice);
```

Define the options.

```
Settle = datetime(2014,1,1);
Maturity = datetime(2014,10,1);  %Options maturity
Strike = [200;90];
OptSpec = {'call'; 'put'};
```

Price the forward call and put options.

```
ForwardMaturity = 'Jan-1-2015';  % Forward contract maturity
Price = optstockbyblk(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike,...
'ForwardMaturity', ForwardMaturity)
```

```
Price = 2×1

    0.0535
    3.2111
```

Compute the Option Price on a Future

Consider a call European option on the Crude Oil Brent futures. The option expires on December 1, 2014 with an exercise price of $120. Assume that on April 1, 2014 futures price is at $105, the annualized continuously compounded risk-free rate is 3.5% per annum and volatility is 22% per annum. Using this data, compute the price of the option.

Define the `RateSpec`.

```
ValuationDate = datetime(2014,1,1);
EndDates = datetime(2015,1,1);
Rates = 0.035;
Compounding = -1;
Basis = 1;
RateSpec  = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate,...
'EndDates', EndDates, 'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis')
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9656
            Rates: 0.0350
         EndTimes: 1
       StartTimes: 0
         EndDates: 735965
       StartDates: 735600
    ValuationDate: 735600
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec`.

```
AssetPrice = 105;
Sigma = 0.22;
StockSpec  = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2200
         AssetPrice: 105
       DividendType: []
    DividendAmounts: 0
```

```
    ExDividendDates: []
```

Define the option.

```
Settle = datetime(2014,4,1);
Maturity = datetime(2014,12,1);
Strike = 120;
OptSpec = {'call'};
```

Price the futures call option.

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

```
Price = 2.5847
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbyblk` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for option

datetime array | string array | date character vector

Maturity date for option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbyblk` also accepts serial date numbers as inputs, but they are not recommended.

OptSpec — Definition of option
cell array of character vectors with values 'call' or 'put'

Definition of the option as 'call' or 'put', specified as a NINST-by-1 cell array of character vectors with values 'call' or 'put'.

Data Types: cell

Strike — Option strike price value
nonnegative vector

Option strike price value, specified as a nonnegative NINST-by-1 vector.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: Price = optstockbyblk(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,'ForwardMaturity',ForwardMaturity)

ForwardMaturity — Maturity date or delivery date of forward contract
Maturity of option (default) | date character vector

Maturity date or delivery date of forward contract, specified as the comma-separated pair consisting of 'ForwardMaturity' and a NINST-by-1 vector using date character vectors.

To support existing code, optstockbyblk also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

Price — Expected option prices
vector

Expected option prices, returned as a NINST-by-1 vector.

More About

Futures Option

A futures option is a standardized contract between two parties to buy or sell a specified asset of standardized quantity and quality for a price agreed upon today (the futures price) with delivery and payment occurring at a specified future date, the delivery date.

The futures contracts are negotiated at a futures exchange, which acts as an intermediary between the two parties. The party agreeing to buy the underlying asset in the future, the "buyer" of the contract, is said to be "long," and the party agreeing to sell the asset in the future, the "seller" of the contract, is said to be "short."

A futures contract is the delivery of item J at time T and:

- There exists in the market a quoted price $F(t,T)$, which is known as the futures price at time t for delivery of J at time T.
- The price of entering a futures contract is equal to zero.
- During any time interval $[t,s]$, the holder receives the amount $F(s,T) - F(t,T)$ (this reflects instantaneous marking to market).
- At time T, the holder pays $F(T,T)$ and is entitled to receive J. Note that $F(T,T)$ should be the spot price of J at time T.

For more information, see “Futures Option” on page 3-32.

Forwards Option

A forwards option is a non-standardized contract between two parties to buy or to sell an asset at a specified future time at a price agreed upon today.

The buyer of a forwards option contract has the right to hold a particular forward position at a specific price any time before the option expires. The forwards option seller holds the opposite forward position when the buyer exercises the option. A call option is the right to enter into a long forward position and a put option is the right to enter into a short forward position. A closely related contract is a futures contract. A forward is like a futures in that it specifies the exchange of goods for a specified price at a specified future date.

The payoff for a forwards option, where the value of a forward position at maturity depends on the relationship between the delivery price (K) and the underlying price (S_T) at that time, is:

- For a long position: $f_T = S_T - K$
- For a short position: $f_T = K - S_T$

For more information, see “Forwards Option” on page 3-31.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbyblk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`impvbyblk` | `intenvset` | `optstocksensbyblk` | `stockspec`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing Using the Black Model” on page 3-83
“Forwards Option” on page 3-31
“Futures Option” on page 3-32
“Black Model” on page 3-80
“Supported Equity Derivative Functions” on page 3-19

optstockbybls

Price options using Black-Scholes option pricing model

Syntax

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

Description

`Price = optstockbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike)` returns option prices using the Black-Scholes option pricing model.

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;
```

For example, see “Compute Option Prices Using the Black-Scholes Option Pricing Model” on page 11-1603.

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. For example, see “Compute Option Prices on Foreign Currencies Using the Garman-Kohlhagen Option Pricing Model” on page 11-1604.

Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Compute Option Prices Using the Black-Scholes Option Pricing Model

This example shows how to compute option prices using the Black-Scholes option pricing model. Consider two European options, a call and a put, with an exercise price of $29 on January 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 = datetime(2008,1,1);
Maturity = datetime(2008,5,1);

% define the RateSpec and StockSpec
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates',...
Maturity, 'Rates', Rates, 'Compounding', -1);

DividendType = {'cash';'continuous'};
DividendAmounts = [0.50; 0.045];
ExDividendDates = {datetime(2008,2,15);NaN};

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

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

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

```
Price = 2×1

    2.2030
    1.2025
```

Compute Option Prices on Foreign Currencies Using the Garman-Kohlhagen Option Pricing Model

This example shows how to compute option prices on foreign currencies using the Garman-Kohlhagen option pricing model. Consider a European put option on a currency with an exercise price of $0.50 on October 1, 2015. The option expires on June 1, 2016. Assume that the current exchange rate is $0.52 and has a volatility of 12% per annum. The annualized continuously compounded domestic risk-free rate is 4% per annum and the foreign risk-free rate is 8% per annum. Using this data, compute the price of the option using the Garman-Kohlhagen model.

```
Settle = datetime(2015,10,1);
Maturity = datetime(2016,6,1);
AssetPrice = 0.52;
Strike = 0.50;
Sigma = .12;
Rates = 0.04;
ForeignRate = 0.08;
```

Define the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9737
```

```
           Rates: 0.0400
         EndTimes: 0.6667
       StartTimes: 0
         EndDates: 736482
       StartDates: 736238
    ValuationDate: 736238
           Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec`.

```
DividendType = 'Continuous';
DividendAmounts = ForeignRate;

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

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1200
         AssetPrice: 0.5200
       DividendType: {'continuous'}
    DividendAmounts: 0.0800
    ExDividendDates: []
```

Price the European put option.

```
OptSpec = {'put'};
Price = optstockbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)
```

```
Price = 0.0162
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for option
datetime array | string array | date character vector

Maturity date for option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbybls` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price value
nonnegative vector

Option strike price value, specified as a nonnegative `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected option prices
vector

Expected option prices, returned as a `NINST`-by-`1` vector.

Data Types: `double`

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`impvbybls` | `intenvset` | `optstocksensbybls` | `stockspec` | `Vanilla`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Pricing Using the Black-Scholes Model” on page 3-82
“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstockbycrr

Price stock option from Cox-Ross-Rubinstein tree

Syntax

```
[Price,PriceTree] = optstockbycrr(CRRTree,OptSpec,Strike,Settle,
ExerciseDates)
[Price,PriceTree] = optstockbycrr( ___ ,AmericanOpt)
```

Description

`[Price,PriceTree] = optstockbycrr(CRRTree,OptSpec,Strike,Settle, ExerciseDates)` returns the price of a European, Bermuda, or American stock option from a Cox-Ross-Rubinstein tree.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optstockbycrr( ___ ,AmericanOpt)` adds an optional argument for `AmericanOpt`.

Examples

Price an American Stock Option Using a CRR Binomial Tree

This example shows how to price an American stock option using a CRR binomial tree by loading the file `deriv.mat`, which provides `CRRTree`. The `CRRTree` structure contains the stock specification and time information needed to price the American option.

```
load deriv.mat;

OptSpec = 'Call';
Strike = 105;
Settle = datetime(2003,1,1);
ExerciseDates = datetime(2005,1,1);
AmericanOpt = 1;

Price = optstockbycrr(CRRTree, OptSpec, Strike, Settle, ...
ExerciseDates, AmericanOpt)

Price = 8.2863
```

Price a Bermudan 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 Bermudan option.

```
load deriv.mat;

% Option
OptSpec = 'Call';
Strike = [110,111,112,113]
```

```
Strike = 1×4

   110   111   112   113
```

```
Settle = datetime(2003,1,1);
ExerciseDatesBerm= [datetime(2004,1,1)  datetime(2005,1,1)  datetime(2006,1,1)  datetime(2007,1,1
```

Price the Bermudan option.

```
Price = optstockbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDatesBerm)
```

```
Price = 11.6017
```

Input Arguments

`CRRTree` — Stock tree structure

structure

Stock tree structure, specified by using `crrtree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a `NINST`-by-1 cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values

nonnegative integer

Option strike price value, specified with a `NINST`-by-1 or `NINST`-by-`NSTRIKES` depending on the option type:

- For a European option, use a `NINST`-by-1 vector of strike prices.
- For a Bermuda option, use a`NINST`-by-`NSTRIKES` matrix of strike prices. 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, use a `NINST`-by-1 of strike prices.

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.

Data Types: `double`

`Settle` — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optstockbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1`,`NINST`-by-`2`, or `NINST`-by-`NSTRIKES` vector using a datetime array, string array, or date character vectors, depending on the option type:

- For a European option, use a `NINST`-by-`1` vector of dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a `NINST`-by-`NSTRIKES` vector of dates. Each row is the schedule for one option.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-`1` vector, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

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.

To support existing code, `optstockbycrr` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
`0` European or Bermuda (default) | integer with values of `0` or `1`

(Optional) Option type, specified as `NINST`-by-`1` vector of integer flags with values:

- `0` — European or Bermuda
- `1` — American

Data Types: `double`

Output Arguments

`Price` — Expected price of option at time `0`

vector

Expected price of the vanilla option at time `0`, returned as a `NINST`-by-`1` vector.

`PriceTree` — Structure containing trees of vectors of instrument prices for each node

structure

Structure containing trees of vectors of instrument prices and a vector of observation times for each node. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.dObs` contains the observation dates.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbycrr` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`crrtree` | `instoptstock` | `Vanilla`

Topics

“Computing Prices Using CRR” on page 3-65
“Examining Output from the Pricing Functions” on page 3-70
“Computing Equity Instrument Sensitivities” on page 3-75
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Computing Instrument Prices” on page 3-64
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstockbyeqp

Price stock option from Equal Probabilities binomial tree

Syntax

```
[Price,PriceTree] = optstockbyeqp(EQPTree,OptSpec,Strike,Settle,
ExerciseDates)
[Price,PriceTree] = optstockbyeqp( ___ ,AmericanOpt)
```

Description

`[Price,PriceTree] = optstockbyeqp(EQPTree,OptSpec,Strike,Settle, ExerciseDates)` returns the price of a European, Bermuda, or American stock option from an Equal Probabilities binomial tree.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optstockbyeqp( ___ ,AmericanOpt)` adds an optional argument for `AmericanOpt`.

Examples

Price an American Stock Option Using an EQP Equity Tree

This example shows how to price an American stock option using an EQP equity tree by loading the file `deriv.mat`, which provides `EQPTree`. The `EQPTree` structure contains the stock specification and time information needed to price the American option.

```
load deriv.mat

OptSpec = 'Call';
Strike = 105;
Settle = datetime(2003,1,1);
ExerciseDates = datetime(2006,1,1);
AmericanOpt = 1;

Price = optstockbyeqp(EQPTree, OptSpec, Strike, Settle, ...
ExerciseDates, AmericanOpt)

Price = 12.2632
```

Price a Bermudan Stock Option Using a 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 Bermudan option.

```
load deriv.mat;

% Option
OptSpec = 'Call';
Strike = 105;
Settle = datetime(2003,1,1);
ExerciseDatesBerm= [datetime(2004,1,15) , datetime(2005,6,15) , datetime(2005,1,15) , datetime(20
```

Price the Bermudan option.

```
Price= optstockbyeqp(EQPTree, OptSpec, Strike, Settle, ExerciseDatesBerm)
```

```
Warning: Some ExerciseDates are not aligned with tree nodes. Result will be approximated.
```

```
Price = 10.0335
```

Input Arguments

`EQPTree` — Stock tree structure

structure

Stock tree structure, specified by using `eqptree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values

nonnegative integer

Option strike price value, specified with a `NINST`-by-`1` or `NINST`-by-`NSTRIKES` depending on the option type:

- For a European option, use a `NINST`-by-`1` vector of strike prices.
- For a Bermuda option, use a`NINST`-by-`NSTRIKES` matrix of strike prices. 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, use a `NINST`-by-`1` of strike prices.

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.

Data Types: `double`

`Settle` — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optstockbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1`,`NINST`-by-`2`, or `NINST`-by-`NSTRIKES` vector using a datetime array, string array, or date character vectors, depending on the option type:

- For a European option, use a `NINST`-by-`1` vector of dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a `NINST`-by-`NSTRIKES` vector of dates. Each row is the schedule for one option.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-`1` vector, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

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.

To support existing code, `optstockbyeqp` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
`0` European or Bermuda (default) | integer with values of `0` or `1`

(Optional) Option type, specified as `NINST`-by-`1` vector of integer flags with values:

- `0` — European or Bermuda
- `1` — American

Data Types: `single` | `double`

Output Arguments

`Price` — Expected price of option at time `0`
vector

Expected price of the vanilla option at time `0`, returned as a `NINST`-by-`1` vector.

`PriceTree` — Structure containing trees of vectors of instrument prices for each node
structure

Structure containing trees of vectors of instrument prices and a vector of observation times for each node. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.dObs` contains the observation dates.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbyeqp` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`eqptree` | `instoptstock` | `Vanilla`

Topics

“Computing Prices Using EQP” on page 3-66
“Examining Output from the Pricing Functions” on page 3-70
“Computing Equity Instrument Sensitivities” on page 3-75
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Computing Instrument Prices” on page 3-64
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstockbyfd

Calculate vanilla option prices using finite difference method

Syntax

```
[Price,PriceGrid,AssetPrices,Times] = optstockbyfd(RateSpec,StockSpec,
OptSpec,Strike,Settle,ExerciseDates)
[Price,PriceGrid,AssetPrices,Times] = optstockbyfd( ___ ,Name,Value)
```

Description

`[Price,PriceGrid,AssetPrices,Times] = optstockbyfd(RateSpec,StockSpec, OptSpec,Strike,Settle,ExerciseDates)` calculates vanilla option prices using the finite difference method.

`[Price,PriceGrid,AssetPrices,Times] = optstockbyfd( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Vanilla Call Option Using Finite Difference Method

Create a `RateSpec`.

```
AssetPrice = 50;
Strike = 45;
Rate = 0.035;
Volatility = 0.30;
Settle = datetime(2015,1,1);
Maturity = datetime(2016,1,1);
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9656
            Rates: 0.0350
         EndTimes: 1
       StartTimes: 0
         EndDates: 736330
       StartDates: 735965
    ValuationDate: 735965
            Basis: 1
     EndMonthRule: 1
```

Create a `StockSpec`.

```
StockSpec = stockspec(Volatility,AssetPrice)

StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3000
         AssetPrice: 50
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Calculate the price of a European vanilla call option using the finite difference method.

```
ExerciseDates = datetime(2015,5,1);
OptSpec = 'Call';
Price = optstockbyfd(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)

Price = 6.7352
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'` | string array with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a character vector or string array with values `'call'` or `'put'`.

Data Types: `char` | `string`

`Strike` — Option strike price value

nonnegative scalar | nonnegative vector

Option strike price value, specified as a nonnegative scalar or vector.

- For a European option, use a scalar of strike price.

- For a Bermuda option, use a 1-by-NSTRIKES vector of strike prices.
- For an American option, use a scalar of strike price.

Data Types: double

Settle — Settlement or trade date
datetime scalar | string scalar | date character vector

Settlement or trade date for the barrier option, specified as a scalar datetime, string, or date character vector.

To support existing code, optstockbyfd also accepts serial date numbers as inputs, but they are not recommended.

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors:

- For a European option, use a 1-by-1 vector of dates. For a Bermuda option, use a 1-by-NSTRIKES vector of dates.
- For an American option, use a 1-by-2 vector of dates. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a 1-by-1 vector dates, the option can be exercised between Settle and the single listed date in ExerciseDates.

To support existing code, optstockbyfd also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: Price = optstockbyfd(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'AssetGridSize',1000)

AssetGridSize — Size of asset grid used for finite difference grid
400 (default) | positive scalar

Size of the asset grid used for a finite difference grid, specified as the comma-separated pair consisting of 'AssetGridSize' and a positive scalar.

Data Types: double

AssetPriceMax — Maximum price for price grid boundary
if unspecified, StockSpec values are calculated using asset distributions at maturity (default) | positive scalar

Maximum price for price grid boundary, specified as the comma-separated pair consisting of 'AssetPriceMax' as a positive scalar.

Data Types: `single` | `double`

`TimeGridSize` — Size of time grid used for finite difference grid
`100` (default) | positive scalar

Size of the time grid used for a finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive scalar.

Data Types: `double`

`AmericanOpt` — Option type
`0` (European/Bermuda) (default) | scalar with values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

Output Arguments

`Price` — Expected prices for vanilla options
scalar

Expected prices for vanilla options, returned as a `1`-by-`1` matrix.

`PriceGrid` — Grid containing prices calculated by finite difference method
grid

Grid containing prices calculated by the finite difference method, returned as a grid that is two-dimensional with size `PriceGridSize*length(Times)`. The number of columns does not have to be equal to the `TimeGridSize`, because ex-dividend dates in the `StockSpec` are added to the time grid. The price for `t = 0` is contained in `PriceGrid(:, end)`.

`AssetPrices` — Prices of asset defined by `StockSpec`
vector

Prices of the asset defined by the `StockSpec` corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Times` — Times corresponding to second dimension of `PriceGrid`
vector

Times corresponding to second dimension of the `PriceGrid`, returned as a vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2016b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbyfd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Haug, E. G., J. Haug, and A. Lewis. "*Back to basics: a new approach to the discrete dividend problem.*" Vol. 9, Wilmott magazine, 2003, pp. 37–47.

[2] Wu, L. and Y. K. Kwok. "A front-fixing finite difference method for the valuation of American options." *Journal of Financial Engineering*. Vol. 6.4, 1997, pp. 83–97.

See Also

`optstocksensbyfd` | `optstockbyls` | `optstockbylr` | `optstockbyblk`

Topics

“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstocksensbyfd

Calculate vanilla option prices or sensitivities using finite difference method

Syntax

```
[PriceSens,PriceGrid,AssetPrices,Times] = optstocksensbyfd(RateSpec,
StockSpec,OptSpec,Strike,Settle,ExerciseDates)
[PriceSens,PriceGrid,AssetPrices,Times] = optstocksensbyfd( ___ ,Name,Value)
```

Description

`[PriceSens,PriceGrid,AssetPrices,Times] = optstocksensbyfd(RateSpec, StockSpec,OptSpec,Strike,Settle,ExerciseDates)` calculates vanilla option prices or sensitivities using the finite difference method.

`[PriceSens,PriceGrid,AssetPrices,Times] = optstocksensbyfd( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Price and Sensitivities for a Vanilla Call Option Using Finite Difference Method

Create a `RateSpec`.

```
AssetPrice = 50;
Strike = 45;
Rate = 0.035;
Volatility = 0.30;
Settle = datetime(2015,1,1);
Maturity = datetime(2016,1,1);
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9656
            Rates: 0.0350
          EndTimes: 1
        StartTimes: 0
          EndDates: 736330
        StartDates: 735965
    ValuationDate: 735965
            Basis: 1
     EndMonthRule: 1
```

Create a `StockSpec`.

```
StockSpec = stockspec(Volatility,AssetPrice)

StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3000
         AssetPrice: 50
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Calculate the price and sensitivities for of a European vanilla call option using the finite difference method.

```
ExerciseDates = datetime(2015,5,1);
OptSpec = 'Call';
OutSpec = {'price'; 'delta'; 'theta'};
[PriceSens, Delta, Theta] = optstocksensbyfd(RateSpec,StockSpec,OptSpec,Strike,Settle,...
ExerciseDates,'OutSpec',OutSpec)

PriceSens = 6.7352

Delta = 0.7765

Theta = -4.9999
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

OptSpec — Definition of option

character vector with values `'call'` or `'put'` | string array with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a character vector or string array with values `'call'` or `'put'`.

Data Types: `char` | `string`

`Strike` — Option strike price value

nonnegative scalar | nonnegative vector

Option strike price value, specified as a nonnegative scalar or vector.

- For a European option, use a scalar of strike price.
- For a Bermuda option, use a 1-by-`NSTRIKES` vector of strike prices.
- For an American option, use a scalar of strike price.

Data Types: `double`

`Settle` — Settlement or trade date

datetime scalar | string scalar | date character vector

Settlement or trade date for the barrier option, specified as a scalar datetime, string, or date character vector.

To support existing code, `optstocksensbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates

datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors as follows:

- For a European option, use a 1-by-1 vector of dates. For a Bermuda option, use a 1-by-`NSTRIKES` vector of dates.
- For an American option, use a 1-by-2 vector of dates. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a 1-by-1 cell array of date character vectors, the option can be exercised between `Settle` and the single listed date in `ExerciseDates`.

To support existing code, `optstocksensbyfd` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = optstocksensbyfd(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'OutSpec',{'All'},'AssetGridSize',1000)`

`OutSpec` — Define outputs

`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-1 or a 1-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

`AssetGridSize` — Size of asset grid used for finite difference grid
`400` (default) | positive scalar

Size of asset grid used for finite difference grid, specified as the comma-separated pair consisting of `'AssetGridSize'` and a positive scalar.

Data Types: `double`

`AssetPriceMax` — Maximum price for price grid boundary
if unspecified, `StockSpec` values are calculated using asset distributions at maturity (default) | positive scalar

Maximum price for price grid boundary, specified as the comma-separated pair consisting of `'AssetPriceMax'` and a positive scalar.

Data Types: `single` | `double`

`TimeGridSize` — Size of time grid used for finite difference grid
`100` (default) | positive scalar

Size of the time grid used for a finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive scalar.

Data Types: `double`

`AmericanOpt` — Option type
`0` (European/Bermuda) (default) | scalar with values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European/Bermuda
- `1` — American

Data Types: `double`

Output Arguments

`PriceSens` — Expected prices or sensitivities for vanilla options
scalar

Expected price or sensitivities (defined by `OutSpec`) of the vanilla option, returned as a `1`-by-`1` array.

`PriceGrid` — Grid containing prices calculated by finite difference method
grid

Grid containing prices calculated by the finite difference method, returned as a two-dimensional grid with size `PriceGridSize*length(Times)`. The number of columns does not have to be equal to the `TimeGridSize`, because ex-dividend dates in the `StockSpec` are added to the time grid. The price for `t = 0` is contained in `PriceGrid(:, end)`.

`AssetPrices` — Prices of asset defined by `StockSpec`
vector

Prices of the asset defined by the `StockSpec` corresponding to the first dimension of `PriceGrid`, returned as a vector.

`Times` — Times corresponding to second dimension of `PriceGrid`
vector

Times corresponding to second dimension of the `PriceGrid`, returned as a vector.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2016b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstocksensbyfd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Haug, E. G., J. Haug, and A. Lewis. "*Back to basics: a new approach to the discrete dividend problem*." Vol. 9, Wilmott magazine, 2003, pp. 37–47.

[2] Wu, L. and Y. K. Kwok. "A front-fixing finite difference method for the valuation of American options." *Journal of Financial Engineering*. Vol. 6.4, 1997, pp. 83–97.

See Also

`optstockbyfd` | `optstockbyls` | `optstockbylr` | `optstockbyblk`

Topics

“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19

optstockbyitt

Price options on stocks using implied trinomial tree (ITT)

Syntax

```
[Price,PriceTree] = optstockbyitt(ITTTree,OptSpec,Strike,Settle,
ExerciseDates)
[Price,PriceTree] = optstockbyitt( ___ ,AmericanOpt)
```

Description

`[Price,PriceTree] = optstockbyitt(ITTTree,OptSpec,Strike,Settle, ExerciseDates)` returns the price of a European, Bermuda, or American stock option from an implied trinomial tree (ITT).

`[Price,PriceTree] = optstockbyitt( ___ ,AmericanOpt)` adds an optional argument for `AmericanOpt`.

Examples

Price an American Stock Option Using an ITT Equity Tree

This example shows how to price an American stock option using an ITT equity tree by loading the file `deriv.mat`, which provides the `ITTTree`. The `ITTTree` structure contains the stock specification and time information needed to price the American option.

```
load deriv.mat

OptSpec = 'Put';
Strike = 30;
Settle = datetime(2006,1,1);
ExerciseDates = datetime(2010,1,1);
AmericanOpt = 1;

Price = optstockbyitt(ITTTree, OptSpec, Strike, Settle,ExerciseDates, AmericanOpt)
```

```
Price = 0.1271
```

Price a Bermudan Stock Option Using an ITT Equity Tree

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

```
load deriv.mat;

% Option
OptSpec = 'Put';
Strike = 30;
Settle = datetime(2006,1,1);
ExerciseDatesBerm = [datetime(2007,1,1) , datetime(2007,6,1) , datetime(2008,1,1) , datetime(2008,6,1)];
```

Price the Bermudan option.

```
Price = optstockbyitt(ITTTree, OptSpec, Strike, Settle, ExerciseDatesBerm)
```

```
Warning: Some ExerciseDates are not aligned with tree nodes. Result will be approximated.
> In procoptions at 171
  In optstockbystocktree at 22
  In optstockbyitt at 68

Price =

    0.0664
```

Input Arguments

`ITTTree` — Stock tree structure

structure

Stock tree structure, specified by using `itttree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values

nonnegative integer

Option strike price value, specified with a `NINST`-by-`1` or `NINST`-by-`NSTRIKES` depending on the option type:

- For a European option, use a `NINST`-by-`1` vector of strike prices.
- For a Bermuda option, use a`NINST`-by-`NSTRIKES` matrix of strike prices. 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, use a `NINST`-by-`1` of strike prices.

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.

Data Types: `double`

`Settle` — Settlement date or trade date

datetime array | string array | date character vector

Settlement date or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

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

To support existing code, `optstockbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1`,`NINST`-by-`2`, or `NINST`-by-`NSTRIKES` vector using a datetime array, string array, or date character vectors, depending on the option type:

- For a European option, use a `NINST`-by-`1` vector of dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a `NINST`-by-`NSTRIKES` vector of dates. Each row is the schedule for one option.
- For an American option, use a `NINST`-by-`2` vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a `NINST`-by-`1` vector, the option can be exercised between `ValuationDate` of the stock tree and the single listed `ExerciseDates`.

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.

To support existing code, `optstockbyitt` also accepts serial date numbers as inputs, but they are not recommended.

`AmericanOpt` — Option type
`0` European or Bermuda (default) | integer with values of `0` or `1`

(Optional) Option type, specified as `NINST`-by-`1` vector of integer flags with values:

- `0` — European or Bermuda
- `1` — American

Data Types: `single` | `double`

Output Arguments

`Price` — Expected price of option at time `0`
vector

Expected price of the vanilla option at time `0`, returned as a `NINST`-by-`1` vector.

`PriceTree` — Structure containing trees of vectors of instrument prices for each node
structure

Structure containing trees of vectors of instrument prices and a vector of observation times for each node. Values are:

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

- `PriceTree.dObs` contains the observation dates.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2007a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbyitt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Chriss, Neil A., E. Derman, and I. Kani. “Implied trinomial trees of the volatility smile.” *Journal of Derivatives.* 1996.

See Also

`instoptstock` | `itttree`

Topics

“Computing Prices Using ITT” on page 3-68
“Examining Output from the Pricing Functions” on page 3-70
“Computing Equity Instrument Sensitivities” on page 3-75
“Graphical Representation of Equity Derivative Trees” on page 3-73
“Vanilla Option” on page 3-27
“Computing Instrument Prices” on page 3-64
“Supported Equity Derivative Functions” on page 3-19

optstockbylr

Price options on stocks using Leisen-Reimer binomial tree model

Syntax

```
[Price,PriceTree] = optstockbylr(LRTree,OptSpec,Strike,Settle,ExerciseDates)
[Price,PriceTree] = optstockbylr( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optstockbylr(LRTree,OptSpec,Strike,Settle,ExerciseDates)` computes option prices on stocks using the Leisen-Reimer binomial tree model.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optstockbylr( ___ ,Name,Value)` adds an optional name-value pair argument for `AmericanOpt`.

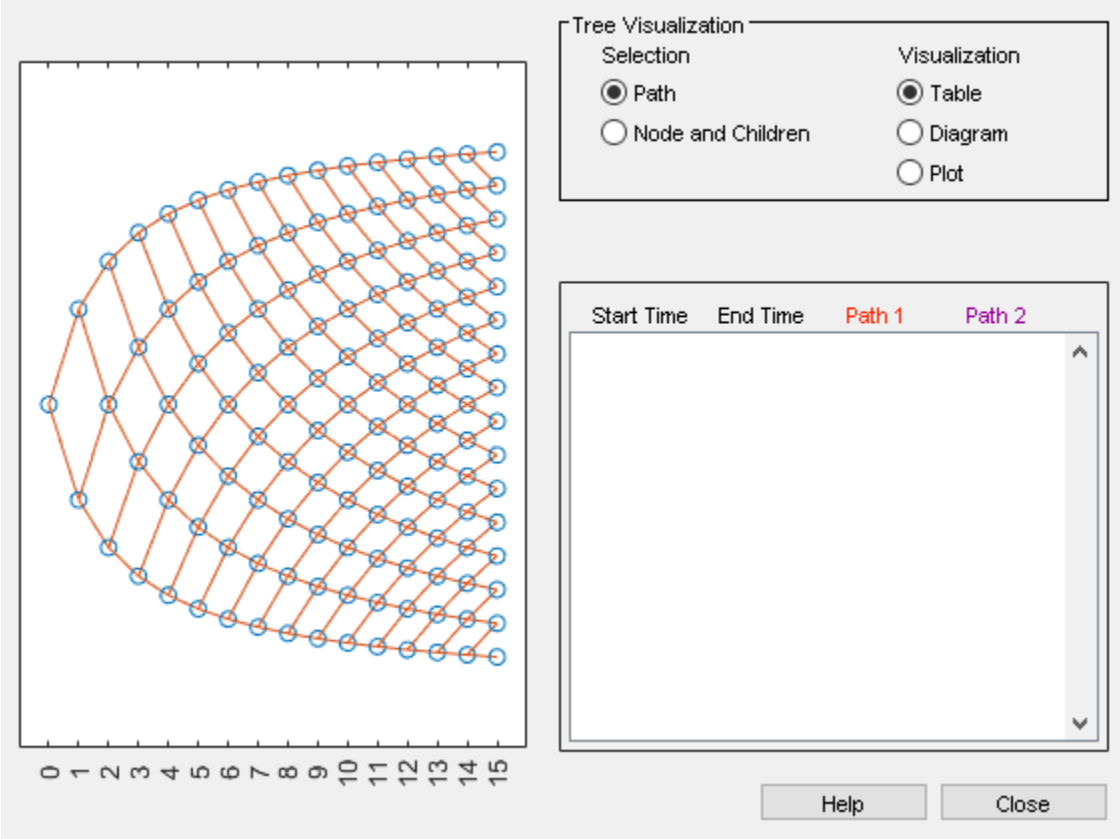
Examples

Price Options on Stocks Using the Leisen-Reimer Binomial Tree Model

This example shows how to price options on stocks using the Leisen-Reimer binomial tree model. Consider European call and put options with an exercise price of $95 that expire on July 1, 2010. The underlying stock is trading at $100 on January 1, 2010, provides a continuous dividend yield of 3% per annum and has a volatility of 20% per annum. The annualized continuously compounded risk-free rate is 8% per annum. Using this data, compute the price of the options using the Leisen-Reimer model with a tree of 15 and 55 time steps.

```
AssetPrice  = 100;
Strike = 95;

ValuationDate = datetime(2010,1,1);
Maturity = datetime(2010,6,1);

% define StockSpec
Sigma = 0.2;
DividendType = 'continuous';
DividendAmounts = 0.03;

StockSpec = stockspec(Sigma, AssetPrice, DividendType, DividendAmounts);

% define RateSpec
Rates = 0.08;
Settle = ValuationDate;
Basis = 1;
```

```
Compounding = -1;

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

% build the Leisen-Reimer (LR) tree with 15 and 55 time steps
LRTimeSpec15  = lrtimespec(ValuationDate, Maturity, 15);
LRTimeSpec55  = lrtimespec(ValuationDate, Maturity, 55);

% use the PP2 method
LRMethod  = 'PP2';

LRTree15 = lrtree(StockSpec, RateSpec, LRTimeSpec15, Strike, 'method', LRMethod);
LRTree55 = lrtree(StockSpec, RateSpec, LRTimeSpec55, Strike, 'method', LRMethod);

% price the call and the put options using the LR model:
OptSpec = {'call'; 'put'};

PriceLR15 = optstockbylr(LRTree15, OptSpec, Strike, Settle, Maturity);
PriceLR55 = optstockbylr(LRTree55, OptSpec, Strike, Settle, Maturity);

% calculate price using the Black-Scholes model (BLS) to compare values with
% the LR model:
PriceBLS = optstockbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike);

% compare values of BLS and LR
[PriceBLS PriceLR15 PriceLR55]

ans = 2×3

    9.0870    9.0826    9.0831
    2.2148    2.2039    2.2044


% use treeviewer to display LRTree of 15 time steps
treeviewer(LRTree15)
```



Input Arguments

`LRTree` — Stock tree structure

structure

Stock tree structure, specified by `lrtree`.

Data Types: `struct`

`OptSpec` — Definition of option

cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price values

vector of nonnegative integers

Option strike price value, specified with nonnegative integer:

- For a European option, use a `NINST`-by-`1` vector of strike prices.
- For a Bermuda option, use a `NINST`-by-`NSTRIKES` vector of strike prices. 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, use a `NINST`-by-`1` vector of strike prices.

Data Types: `double`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbylr` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors, where each row is the schedule for one option and the last element of each row must be the same as the maturity of the tree.

- For a European option, use a `NINST`-by-`1` vector of dates. For a European option, there is only one `ExerciseDate` on the option expiry date.
- For a Bermuda option, use a `NINST`-by-`NSTRIKEDATES` vector of dates.
- For an American option, use a `NINST`-by-`1` vector of exercise dates. For the American type, the option can be exercised on any tree data between the `ValuationDate` and tree maturity.

To support existing code, `optstockbylr` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = optstockbylr(LRTree,OptSpec,Strike,Settle,ExerciseDates,'AmericanOpt','1')`

`AmericanOpt` — Option type
`0` European or Bermuda (default) | values: `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a `NINST`-by-`1` vector of flags with values:

- `0` — European or Bermuda
- `1` — American

Data Types: `double`

Output Arguments

`Price` — Expected prices at time 0
vector

expected prices at time 0, returned as a `NINST`-by-1 vector.

`PriceTree` — Tree structure
structure

Tree structure, returned as a vector of instrument prices at each node. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.dObs` contains the observation dates.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2010b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbylr` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Leisen D.P., M. Reimer. “Binomial Models for Option Valuation – Examining and Improving Convergence.” *Applied Mathematical Finance.* Number 3, 1996, pp. 319–346.

See Also

`instoptstock` | `lrtree` | `Vanilla`

Topics

“Pricing Equity Derivatives Using Trees” on page 3-64
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstockbyls

Price European, Bermudan, or American vanilla options using Monte Carlo simulations

Syntax

```
Price = optstockbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)
Price = optstockbyls( ___ ,Name,Value)

[Price,Path,Times,Z] = optstockbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates)
[Price,Path,Times,Z] = optstockbyls( ___ ,Name,Value)
```

Description

`Price = optstockbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)` returns vanilla option prices using the Longstaff-Schwartz model. `optstockbyls` computes prices of European, Bermudan, and American vanilla options.

For American and Bermudan options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = optstockbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

`[Price,Path,Times,Z] = optstockbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)` returns vanilla option prices using the Longstaff-Schwartz model.

`[Price,Path,Times,Z] = optstockbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of a Vanilla Option

Define the `RateSpec`.

```
StartDates = datetime(2013,1,1);
EndDates = datetime(2015,1,1);
Rates = 0.05;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates, ...
'EndDates', EndDates, 'Rates', Rates)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
```

```
             Disc: 0.9060
            Rates: 0.0500
          EndTimes: 4
        StartTimes: 0
          EndDates: 735965
        StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec` for the asset.

```
AssetPrice = 100;
Sigma = 0.1;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1000
         AssetPrice: 100
        DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Define the vanilla option.

```
OptSpec = 'put';
Settle = datetime(2013,1,1);
ExerciseDates = datetime(2015,1,1);
Strike = 105;
```

Compute the vanilla option price using the Longstaff-Schwartz model.

```
Antithetic = true;
Price = optstockbyls(RateSpec, StockSpec, OptSpec, Strike, Settle, ...
ExerciseDates, 'Antithetic', Antithetic)
```

```
Price = 3.2292
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with value `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a character vector.

Data Types: `char`

`Strike` — Option strike price values
nonnegative scalar integer

Option strike price value, specified with nonnegative scalar integer:

- For a European option, use a scalar of strike price.
- For a Bermuda option, use a 1-by-`NSTRIKES` vector of strike prices.
- For an American option, use a scalar of strike price.

Data Types: `single` | `double`

`Settle` — Settlement date or trade date
datetime scalar | string scalar | date character vector

Settlement date or trade date for the vanilla option, specified as a scalar datetime, string, or date character vector.

To support existing code, `optstockbyls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a datetime array, string array, or date character vectors as follows:

- For a European option, use a 1-by-1 vector of dates. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a 1-by-`NSTRIKES` vector of dates.
- For an American option, use a 1-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a 1-by-1 cell array of character vectors, the option can be exercised between `Settle` and the single listed `ExerciseDates`.

To support existing code, `optstockbyls` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optstockbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'AmericanOpt','1','NumTrials','2000')`

`AmericanOpt` — Option type
`0` European or Bermuda (default) | scalar with values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and positive integer scalar flags with values:

- `0` — European or Bermuda
- `1` — American

Note For American and Bermudan options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `double`

`NumTrials` — Simulation trials
`1000` (default) | scalar

Simulation trials, specified as a scalar number of independent sample paths.

Data Types: `double`

`NumPeriods` — Simulation periods per trial
`100` (default) | scalar

Simulation periods per trial, specified as a scalar number. `NumPeriods` is considered only when pricing European vanilla options. For American and Bermuda vanilla options, `NumPeriod` is equal to the number of `Exercise` days during the life of the option.

Data Types: `double`

`Z` — Dependent random variates
scalar | nonnegative integer

Dependent random variates used to generate the Brownian motion vector (that is, Wiener processes) that drive the simulation, specified as a be `NumPeriods`-by-`1`-by-`NumTrials` 3-D time series array.

Data Types: `double`

`Antithetic` — Indicator for antithetic sampling
`false` (default) | logical flag with value of `true` or `false`

Indicator for antithetic sampling, specified with a value of `true` or `false`.

Data Types: `logical`

Output Arguments

`Price` — Expected price of vanilla option

scalar

Expected price of the vanilla option, returned as a 1-by-1 scalar.

`Path` — Simulated paths of correlated state variables

vector

Simulated paths of correlated state variables, returned as a (`NumPeriods` + 1) -by-1-by-`NumTrials` 3-D time series array. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths

vector

Observation times associated with simulated paths, returned as a (`NumPeriods` + 1)-by-1 column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates

vector

Dependent random variates, if `Z` is specified as an optional input argument, the same value is returned. Otherwise, `Z` contains the random variates generated internally.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`optstocksensbyls` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstocksensbyls

Calculate price and sensitivities for European, Bermudan, or American vanilla options using Monte Carlo simulations

Syntax

```
PriceSens = optstocksensbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,
ExerciseDates)
PriceSens = optstocksensbyls( ___ ,Name,Value)

[PriceSens,Path,Times,Z] = optstocksensbyls(RateSpec,StockSpec,OptSpec,
Strike,Settle,ExerciseDates)
[PriceSens,Path,Times,Z] = optstocksensbyls( ___ ,Name,Value)
```

Description

`PriceSens = optstocksensbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)` returns vanilla option prices or sensitivities using the Longstaff-Schwartz model. `optstocksensbyls` computes prices or sensitivities of European, Bermudan, and American vanilla options.

For American and Bermudan options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = optstocksensbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

`[PriceSens,Path,Times,Z] = optstocksensbyls(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates)` returns vanilla option prices or sensitivities using the Longstaff-Schwartz model.

`[PriceSens,Path,Times,Z] = optstocksensbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price and Sensitivities of a Vanilla Option

Define the `RateSpec`.

```
StartDates = datetime(2013,1,1);
EndDates = datetime(2015,1,1);
Rates = 0.05;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates, ...
'EndDates', EndDates, 'Rates', Rates)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: 0.9060
            Rates: 0.0500
         EndTimes: 4
       StartTimes: 0
         EndDates: 735965
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Define the `StockSpec` for the asset.

```
AssetPrice = 100;
Sigma = 0.1;
DivType = 'continuous';
DivAmounts = 0.04;
StockSpec = stockspec(Sigma, AssetPrice, DivType, DivAmounts)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1000
         AssetPrice: 100
       DividendType: {'continuous'}
    DividendAmounts: 0.0400
    ExDividendDates: []
```

Define the vanilla option.

```
OptSpec = 'call';
Settle = datetime(2013,1,1);
ExerciseDates = datetime(2015,1,1);
Strike = 105;
```

Compute the `Delta` sensitivity for the vanilla option using the Longstaff-Schwartz model.

```
Antithetic = true;
OutSpec = {'Delta'};
PriceSens = optstocksensbyls(RateSpec, StockSpec, OptSpec, Strike, ...
Settle, ExerciseDates,'Antithetic', Antithetic, 'OutSpec', OutSpec)
```

```
PriceSens = 0.3945
```

To display the output for `Price`, `Delta`, `Path`, and `Times`, use the following:

```
OutSpec = {'Price','Delta'};
[Price, Delta, Path, Times] = optstocksensbyls(RateSpec, StockSpec, OptSpec, Strike, ...
Settle, ExerciseDates,'Antithetic', Antithetic, 'OutSpec', OutSpec);
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset
structure

Stock specification for underlying asset, specified using `StockSpec` obtained from `stockspec`. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, stocks, stock indices, and commodities.

Data Types: `struct`

`OptSpec` — Definition of option
character vector values `'call'` or `'put'`

Definition of option, specified as `'call'` or `'put'` using a character vector.

Data Types: `char`

`Strike` — Option strike price value
nonnegative scalar integer

Option strike price value, specified with a nonnegative scalar integer:

- For a European option, use a scalar of strike price.
- For a Bermuda option, use a 1-by-`NSTRIKES` vector of strike price.
- For an American option, use a scalar of strike price.

Data Types: `double`

`Settle` — Settlement date or trade date
datetime scalar | string scalar | date character vector

Settlement date or trade date for the vanilla option, specified as a scalar datetime, string, or date character vector.

To support existing code, `optstocksensbyls` also accepts serial date numbers as inputs, but they are not recommended.

`ExerciseDates` — Option exercise date
datetime array | string array | date character vector

Option exercise date, specified using a datetime array, string array, or date character vectors as follows:

- For a European option, use a 1-by-1 vector of dates. For a European option, there is only one `ExerciseDates` on the option expiry date.
- For a Bermuda option, use a 1-by-`NSTRIKES` vector of dates.
- For an American option, use a 1-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-`NaN` date is listed, or if `ExerciseDates` is a 1-by-1 cell array of character vectors, the option can be exercised between `Settle` and the single listed `ExerciseDates`.

To support existing code, `optstocksensbyls` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optstocksensbyls(RateSpec,StockSpec, OptSpec,Strike,Settle,ExerciseDates,'AmericanOpt','1','NumTrials','2000','OutSpec',{'Price','Delta','Gamma'})`

`AmericanOpt` — Option type

`0` European or Bermuda (default) | scalar with values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a positive integer scalar flag with values:

- `0` — European or Bermuda
- `1` — American

Note For American and Bermudan options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `single` | `double`

`NumTrials` — Simulation trials

`1000` (default) | scalar

Simulation trials, specified as the comma-separated pair consisting of `'NumTrials'` and a scalar number of independent sample paths.

Data Types: `double`

`NumPeriods` — Simulation periods per trial

`100` (default) | scalar

Simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar number. `NumPeriods` is considered only when pricing European vanilla options. For American and Bermuda vanilla options, `NumPeriod` is equal to the number of `Exercise` days during the life of the option.

Data Types: `double`

`Z` — Dependent random variates

scalar | nonnegative integer

Dependent random variates used to generate the Brownian motion vector (that is, Wiener processes) that drive the simulation, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-`1`-by-`NumTrials` 3-D time series array.

Data Types: `single` | `double`

`Antithetic` — Indicator for antithetic sampling
`false` (default) | logical flag with value of `true` or `false`

Indicator for antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a value of `true` or `false`.

Data Types: `logical`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values: `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected price or sensitivities of vanilla option
scalar

Expected price or sensitivities (defined by `OutSpec`) of the vanilla option, returned as a `1`-by-`1` array.

`Path` — Simulated paths of correlated state variables
vector

Simulated paths of correlated state variables, returned as a (`NumPeriods` + `1`)-by-`1`-by-`NumTrials` 3-D time series array. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths
vector

Observation times associated with simulated paths, returned as a (`NumPeriods` + `1`)-by-`1` column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates
vector

Dependent random variates, if `Z` is specified as an optional input argument, the same value is returned. Otherwise, `Z` contains the random variates generated internally.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstocksensbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`optstockbyls` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstockbyrgw

Determine American call option prices using Roll-Geske-Whaley option pricing model

Syntax

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

Description

`Price = optstockbyrgw(RateSpec,StockSpec,Settle,Maturity,Strike)` computes the American call option prices using the Roll-Geske-Whaley option pricing model. `optstockbyrgw` computes prices of American calls with a single cash dividend using the Roll-Geske-Whaley option pricing model.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Determine American Call Option Prices Using Roll-Geske-Whaley Option Pricing Model

This example shows how to determine American call option prices using Roll-Geske-Whaley option pricing model. 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 = datetime(2008,6,1);
Maturity = datetime(2009,2,1);
AssetPrice = 20;
Strike = 22;
Sigma  = 0.2;
Rate = 0.0677;
DivAmount = 2;
DivDate = datetime(2008,9,1);

% define the RateSpec and StockSpec
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates',...
Maturity, 'Rates', Rate, 'Compounding', -1, 'Basis', 0);

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

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

Price = 0.3359
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbyrgw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for option

datetime array | string array | date character vector

Maturity date for option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstockbyrgw` also accepts serial date numbers as inputs, but they are not recommended.

`Strike` — Option strike price value

nonnegative vector

Option strike price value, specified as a nonnegative `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected call option prices

vector

Expected call option prices, returned as a `NINST`-by-`1` vector.

Data Types: `double`

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbyrgw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`impvbyrgw` | `intenvset` | `optstocksensbyrgw` | `stockspec` | `Vanilla`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing Using the Roll-Geske-Whaley Model” on page 3-84
“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Roll-Geske-Whaley Model” on page 3-80

“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstocksensbybjs

Determine American option prices or sensitivities using Bjerksund-Stensland 2002 option pricing model

Syntax

```
PriceSens = optstocksensbybjs(RateSpec,StockSpec,Settle,Maturity,OptSpec,
Strike)
PriceSens = optstocksensbybjs( ___ ,Name,Value)
```

Description

`PriceSens = optstocksensbybjs(RateSpec,StockSpec,Settle,Maturity,OptSpec, Strike)` computes American option prices or sensitivities using the Bjerksund-Stensland 2002 option pricing model.

`optstocksensbybjs` computes prices of American options with continuous dividend yield using the Bjerksund-Stensland option pricing model. All sensitivities are evaluated by computing a discrete approximation of the partial derivative. This means that the option is revalued with a fractional change for each relevant parameter, and the change in the option value divided by the increment, is the approximated sensitivity value.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = optstocksensbybjs( ___ ,Name,Value)` adds an optional name-value pair argument for `OutSpec`.

Examples

Compute American Option Prices and Sensitivities Using the Bjerksund-Stensland 2002 Option Pricing Model

This example shows how to compute American option prices and sensitivities using the Bjerksund-Stensland 2002 option pricing model. 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 = datetime(2008,6,1);
Maturity = datetime(2008,10,1);
Strike = 100;
AssetPrice = [90;100;110;120];
Rate = 0.08;
```

```
Sigma = 0.40;
DivYield = 0.04;

% define the 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 = {'put'};

OutSpec = {'Delta', 'Gamma', 'Price'};

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

```
Delta = 4×1

   -0.6236
   -0.4351
   -0.2794
   -0.1673


Gamma = 4×1

    0.0196
    0.0176
    0.0134
    0.0091


Price = 4×1

   13.7594
    8.4830
    4.9451
    2.7475
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstocksensbybjs` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for option
datetime array | string array | date character vector

Maturity date for option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstocksensbybjs` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price value
nonnegative vector

Option strike price value, specified as a nonnegative `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

`OutSpec` — Define outputs
`{'Price'}` (default) | cell array of character vectors with values: `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected future prices or sensitivities values

vector

Expected future prices or sensitivities values, returned as a `NINST`-by-`1` vector.

Data Types: `double`

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstocksensbybjs` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bjerksund, P. and G. Stensland. “Closed-Form Approximation of American Options.” *Scandinavian Journal of Management.* Vol. 9, 1993, Suppl., pp. S88–S99.

[2] Bjerksund, P. and G. Stensland. *“Closed Form Valuation of American Options.”* Discussion paper, 2002.

See Also

`impvbybjs` | `intenvset` | `optstockbybjs` | `stockspec` | `Vanilla`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing Using the Bjerksund-Stensland Model” on page 3-84
“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Bjerksund-Stensland 2002 Model” on page 3-81
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstocksensbyblk

Determine option prices or sensitivities on futures and forwards using Black option pricing model

Syntax

```
PriceSens = optstocksensbyblk(RateSpec,StockSpec,Settle,Maturity,OptSpec,
Strike)
PriceSens = optstocksensbyblk( ___ ,Name,Value)
```

Description

`PriceSens = optstocksensbyblk(RateSpec,StockSpec,Settle,Maturity,OptSpec, Strike)` computes option prices on futures and forwards using the Black option pricing model.

Note `optstocksensbyblk` calculates option prices or sensitivities on futures and forwards. If `ForwardMaturity` is not passed, the function calculates prices or sensitivities of future options. If `ForwardMaturity` is passed, the function computes prices or sensitivities of forward options. This function handles several types of underlying assets, for example, stocks and commodities. For more information on the underlying asset specification, see `stockspec`.

`PriceSens = optstocksensbyblk( ___ ,Name,Value)` adds optional name-value pair arguments for `ForwardMaturity` and `OutSpec` to compute option prices or sensitivities on forwards using the Black option pricing model.

Examples

Compute Option Prices and Sensitivities on Futures Using the Black Pricing Model

This example shows how to compute option prices and sensitivities on futures using the Black pricing model. 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 = datetime(2008,4,1);
Maturity = datetime(2008,6,30);

% define the 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'};

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
```

Compute Forward Option Prices and Delta Sensitivities

This example shows how to compute option prices and sensitivities on forwards using the Black pricing model. Consider two European call options on the Brent Blend forward contract that expires on January 1, 2015. The options expire on October 1, 2014 and Dec 1, 2014 with an exercise price % of $120 and $150 respectively. Assume that on January 1, 2014 the forward price is at $107, the annualized continuously compounded risk-free rate is 3% per annum and volatility is 28% per annum. Using this data, compute the price and delta of the options.

Define the `RateSpec`.

```
ValuationDate = datetime(2014,1,1);
EndDates = datetime(2015,1,1);
Rates = 0.03;
Compounding = -1;
Basis = 1;
RateSpec  = intenvset('ValuationDate', ValuationDate, 'StartDates', ...
ValuationDate, 'EndDates', EndDates, 'Rates', Rates, ...
'Compounding', Compounding, 'Basis', Basis');
```

Define the `StockSpec`.

```
AssetPrice = 107;
Sigma = 0.28;
StockSpec  = stockspec(Sigma, AssetPrice);
```

Define the options.

```
Settle = datetime(2014,1,1);
Maturity = [datetime(2014,10,1) ; datetime(2014,12,1)]; %Options maturity
Strike = [120;150];
OptSpec = {'call'; 'call'};
```

Price the forward call options and return the `Delta` sensitivities.

```
ForwardMaturity = 'Jan-1-2015'; % Forward contract maturity
OutSpec = {'Delta'; 'Price'};
[Delta, Price] = optstocksensbyblk(RateSpec, StockSpec, Settle, Maturity, OptSpec, ...
Strike, 'ForwardMaturity', ForwardMaturity, 'OutSpec', OutSpec)

Delta = 2×1

    0.3518
```

```
    0.1262


Price = 2×1

    5.4808
    1.6224
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstocksensbyblk` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date for option

datetime array | string array | date character vector

Maturity date for option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstocksensbyblk` also accepts serial date numbers as inputs, but they are not recommended.

OptSpec — Definition of option

cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

Strike — Option strike price value
nonnegative vector

Option strike price value, specified as a nonnegative NINST-by-1 vector.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Delta,Gamma,Price] = optstocksensbyblk(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,'OutSpec',OutSpec)

ForwardMaturity — Maturity date or delivery date of forward contract
Maturity of option (default) | date character vector

Maturity date or delivery date of forward contract, specified as the comma-separated pair consisting of 'ForwardMaturity' and a NINST-by-1 vector using date character vectors.

To support existing code, optstocksensbyblk also accepts serial date numbers as inputs, but they are not recommended.

OutSpec — Define outputs
{'Price'} (default) | cell array of character vectors with values: 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and a NOUT- by-1 or 1-by-NOUT cell array of character vectors with possible values of 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'.

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 to include each sensitivity:

Example: OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}

Data Types: char | cell

Output Arguments

PriceSens — Expected future prices or sensitivities values
vector

Expected future prices or sensitivities values, returned as a NINST-by-1 vector.

Data Types: double

More About

Futures Option

A futures option is a standardized contract between two parties to buy or sell a specified asset of standardized quantity and quality for a price agreed upon today (the futures price) with delivery and payment occurring at a specified future date, the delivery date.

The futures contracts are negotiated at a futures exchange, which acts as an intermediary between the two parties. The party agreeing to buy the underlying asset in the future, the "buyer" of the contract, is said to be "long," and the party agreeing to sell the asset in the future, the "seller" of the contract, is said to be "short."

A futures contract is the delivery of item J at time T and:

- There exists in the market a quoted price $F(t, T)$, which is known as the futures price at time t for delivery of J at time T.
- The price of entering a futures contract is equal to zero.
- During any time interval $[t,s]$, the holder receives the amount $F(s, T) - F(t, T)$ (this reflects instantaneous marking to market).
- At time T, the holder pays $F(T, T)$ and is entitled to receive J. Note that $F(T, T)$ should be the spot price of J at time T.

For more information, see “Futures Option” on page 3-32.

Forwards Option

A forwards option is a non-standardized contract between two parties to buy or to sell an asset at a specified future time at a price agreed upon today.

The buyer of a forwards option contract has the right to hold a particular forward position at a specific price any time before the option expires. The forwards option seller holds the opposite forward position when the buyer exercises the option. A call option is the right to enter into a long forward position and a put option is the right to enter into a short forward position. A closely related contract is a futures contract. A forward is like a futures in that it specifies the exchange of goods for a specified price at a specified future date.

The payoff for a forwards option, where the value of a forward position at maturity depends on the relationship between the delivery price (K) and the underlying price (S_T) at that time, is:

- For a long position: $f_T = S_T - K$
- For a short position: $f_T = K - S_T$

For more information, see “Forwards Option” on page 3-31.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstocksensbyblk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`impvbyblk` | `intenvset` | `optstockbyblk` | `stockspec`

Topics

“Pricing Asian Options” on page 3-110
“Forwards Option” on page 3-31
“Futures Option” on page 3-32
“Black Model” on page 3-80
“Supported Equity Derivative Functions” on page 3-19
“Supported Energy Derivative Functions” on page 3-34

optstocksensbybls

Determine option prices or sensitivities using Black-Scholes option pricing model

Syntax

```
PriceSens = optstocksensbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,
Strike)
PriceSens = optstocksensbybls( ___ ,Name,Value)
```

Description

`PriceSens = optstocksensbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec, Strike)` computes option prices or sensitivities using the Black-Scholes option pricing model.

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.

Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = optstocksensbybls( ___ ,Name,Value)` adds an optional name-value pair argument for `OutSpec`.

Examples

Compute Option Prices and Sensitivities Using the Black-Scholes Option Pricing Model

This example shows how to compute option prices and sensitivities using the Black-Scholes option pricing model. 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 = datetime(2008,1,1);
Maturity = datetime(2008,6,1);

% define the 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'};

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

```
Delta = 2×1

    0.5810
   -0.4190


Gamma = 2×1

    0.0673
    0.0673


Price = 2×1

    2.6126
    1.9941
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstocksensbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for option
datetime array | string array | date character vector

Maturity date for option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstocksensbybls` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price value
nonnegative vector

Option strike price value, specified as a nonnegative `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Delta,Gamma,Price] = optstocksensbybls(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,'OutSpec',OutSpec)`

`OutSpec` — Define outputs
`{'Price'}` (default) | cell array of character vectors with values: `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected future prices or sensitivities values

vector

Expected future prices or sensitivities values, returned as a `NINST`-by-`1` vector.

Data Types: `double`

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstocksensbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`impvbybls` | `intenvset` | `optstockbybls` | `stockspec` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Supported Energy Derivative Functions” on page 3-34
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstocksensbylr

Determine option prices or sensitivities using Leisen-Reimer binomial tree model

Syntax

```
PriceSens = optstockbylr(LRTree,OptSpec,Strike,Settle,ExerciseDates)
PriceSens = optstockbylr( ___ ,Name,Value)
```

Description

`PriceSens = optstockbylr(LRTree,OptSpec,Strike,Settle,ExerciseDates)` calculates option prices or sensitivities using a Leisen-Reimer binomial tree model.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = optstockbylr( ___ ,Name,Value)` adds optional name-value pair arguments for `AmericanOpt` and `OutSpec`.

Examples

Compute Option Prices and Sensitivities Using a Leisen-Reimer Binomial Tree Model

This example shows how to compute option prices and sensitivities using a Leisen-Reimer binomial tree model. Consider European call and put options with an exercise price of $100 that expire on December 1, 2010. The underlying stock is trading at $100 on June 1, 2010 and has a volatility of 30% per annum. The annualized continuously compounded risk-free rate is 7% per annum. Using this data, compute the price, delta and gamma of the options using the Leisen-Reimer model with a tree of 25 time steps and the `PP2` method.

```
AssetPrice  = 100;
Strike = 100;

ValuationDate = datetime(2010,6,1);
Maturity = datetime(2010,12,1);

% define StockSpec
Sigma = 0.3;

StockSpec = stockspec(Sigma, AssetPrice);

% define RateSpec
Rates = 0.07;
Settle = ValuationDate;
Basis = 1;
Compounding = -1;
```

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

% build the Leisen-Reimer (LR) tree with 25 time steps
LRTimeSpec  = lrtimespec(ValuationDate, Maturity, 25);

% use the PP2 method
LRMethod  = 'PP2';

TreeLR = lrtree(StockSpec, RateSpec, LRTimeSpec, Strike, 'method', LRMethod);

% compute prices and sensitivities using the LR model:
OptSpec = {'call'; 'put'};
OutSpec = {'Price', 'Delta', 'Gamma'};

[Price, Delta, Gamma] = optstocksensbylr(TreeLR, OptSpec, Strike, Settle, ...
Maturity, 'OutSpec', OutSpec)
```

```
Price = 2×1

   10.1332
    6.6937


Delta = 2×1

    0.6056
   -0.3944


Gamma = 2×1

    0.0185
    0.0185
```

Input Arguments

LRTree — Stock tree structure

structure

Stock tree structure, specified by `lrtree`.

Data Types: `struct`

OptSpec — Definition of option

cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

Strike — Option strike price values

vector of nonnegative integers

Option strike price value, specified with nonnegative integer:

- For a European option, use a NINST-by-1 vector of strike prices.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of strike prices. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- For an American option, use a NINST-by-1 vector of strike prices.

Data Types: double

Settle — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, optstocksensbylr also accepts serial date numbers as inputs, but they are not recommended.

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a NINST-by-1 or NINST-by-NSTRIKEDATES vector using a datetime array, string array, or date character vectors, where each row is the schedule for one option and the last element of each row must be the same as the maturity of the tree.

- For a European option, use a NINST-by-1 vector of dates. For a European option, there is only one ExerciseDate on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKEDATES vector of dates.
- For an American option, use a NINST-by-1 vector of exercise dates. For the American type, the option can be exercised on any tree data between the ValuationDate and tree maturity.

To support existing code, optstocksensbylr also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Price,Delta,Gamma] = optstocksensbylr(LRTree,OptSpec,Strike,Settle,ExerciseDates,'OutSpec',OutSpec)

AmericanOpt — Option type
0 European or Bermuda (default) | values: [0,1]

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and a NINST-by-1 vector of flags with values:

- 0 — European or Bermuda
- 1 — American

Data Types: `double`

`OutSpec` — Define outputs
`{'Price'}` (default) | cell array of character vectors with values: `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected future prices or sensitivities values
vector

Expected future prices or sensitivities values, returned as a `NINST`-by-`1` vector.

Data Types: `double`

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2010b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `optstocksensbylr` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Leisen D.P., M. Reimer. “Binomial Models for Option Valuation – Examining and Improving Convergence.” *Applied Mathematical Finance.* Number 3, 1996, pp. 319–346.

See Also

`optstockbylr` | `lrtree` | `Vanilla`

Topics

“Pricing Equity Derivatives Using Trees” on page 3-64
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstocksensbyrgw

Determine American call option prices or sensitivities using Roll-Geske-Whaley option pricing model

Syntax

```
PriceSens = optstocksensbyrgw(RateSpec,StockSpec,Settle,Maturity,OptSpec,
Strike)
PriceSens = optstocksensbyrgw( ___ ,Name,Value)
```

Description

`PriceSens = optstocksensbyrgw(RateSpec,StockSpec,Settle,Maturity,OptSpec, Strike)` computes American call option prices or sensitivities using the Roll-Geske-Whaley option pricing model.

`optstocksensbyrgw` computes prices of American calls with a single cash dividend using the Roll-Geske-Whaley option pricing model. All sensitivities are evaluated by computing a discrete approximation of the partial derivative. This means that the option is revalued with a fractional change for each relevant parameter, and the change in the option value divided by the increment, is the approximated sensitivity value.

Note Alternatively, you can use the `Vanilla` object to calculate price or sensitivities for vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = optstocksensbyrgw( ___ ,Name,Value)` adds an optional name-value pair argument for `OutSpec`.

Examples

Compute American Call Option Prices and Sensitivities Using the Roll-Geske-Whaley Option Pricing Model

This example shows how to compute American call option prices and sensitivities using the Roll-Geske-Whaley option pricing model. 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 = datetime(2008,1,1);
Maturity = datetime(2008,5,1);
Strike = 82;
Rate = 0.06;
Sigma  = 0.3;
DivAmount = 4;
```

```
DivDate = datetime(2008,4,1);

% define the RateSpec and StockSpec
StockSpec = stockspec(Sigma, AssetPrice, {'cash'}, DivAmount, DivDate);

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

% define the OutSpec
OutSpec = {'Price', 'Delta', 'Gamma'};

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

Price = 4.3860

Delta = 0.5022

Gamma = 0.0336
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstocksensbyrgw` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date for option

datetime array | string array | date character vector

Maturity date for option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `optstocksensbyrgw` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
cell array of character vectors with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors with values `'call'` or `'put'`.

Data Types: `char` | `cell`

`Strike` — Option strike price value
nonnegative vector

Option strike price value, specified as a nonnegative `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Delta,Gamma,Price] = optstocksensbyrgw(RateSpec,StockSpec,Settle,Maturity,OptSpec,Strike,'OutSpec',OutSpec)`

`OutSpec` — Define outputs
`{'Price'}` (default) | cell array of character vectors with values: `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected future prices or sensitivities values
vector

Expected future prices or sensitivities values, returned as a `NINST`-by-`1` vector.

Data Types: `double`

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstocksensbyrgw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`impvbyrgw` | `intenvset` | `optstockbyrgw` | `stockspec` | `Vanilla`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing Using the Roll-Geske-Whaley Model” on page 3-84
“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19

“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optstockbystt

Price vanilla options on stocks using standard trinomial tree

Syntax

```
[Price,PriceTree] = optstockbystt(STTTree,OptSpec,Strike,Settle,
ExerciseDates)
[Price,PriceTree] = optstockbystt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = optstockbystt(STTTree,OptSpec,Strike,Settle, ExerciseDates)` returns vanilla option (American, European, or Bermudan) prices on stocks using a standard trinomial (STT) tree.

Note Alternatively, you can use the `Vanilla` object to price vanilla options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = optstockbystt( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price Call and Put Stock Options Using the Standard Trinomial Tree Model

Create a `RateSpec`.

```
StartDates = datetime(2009,1,1);
EndDates = datetime(2013,1,1);
Rates = 0.035;
Basis = 1;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8694
            Rates: 0.0350
         EndTimes: 4
       StartTimes: 0
         EndDates: 735235
       StartDates: 733774
    ValuationDate: 733774
            Basis: 1
     EndMonthRule: 1
```

Create a `StockSpec`.

```
AssetPrice = 85;
Sigma = 0.15;
StockSpec = stockspec(Sigma, AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1500
         AssetPrice: 85
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Create an `STTTree`.

```
NumPeriods = 4;
TimeSpec = stttimespec(StartDates, EndDates, 4);
STTTree = stttree(StockSpec, RateSpec, TimeSpec)
```

```
STTTree = struct with fields:
       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 1 2 3 4]
         dObs: [733774 734139 734504 734869 735235]
        STree: {1x5 cell}
        Probs: {[3x1 double]  [3x3 double]  [3x5 double]  [3x7 double]}
```

Define the call and put options and compute the price.

```
Settle = datetime(2009,1,1);
ExerciseDates = [datetime(2011,1,1) ; datetime(2012,1,1)];
OptSpec =  {'call';'put'};
Strike =[100;80];

Price = optstockbystt(STTTree, OptSpec, Strike, Settle, ExerciseDates)
```

```
Price = 2×1

    4.5025
    3.0603
```

Input Arguments

STTTree — Stock tree structure for standard trinomial tree

structure

Stock tree structure for a standard trinomial tree, specified by using `stttree`.

Data Types: `struct`

OptSpec — Definition of option
character vector with value 'call' or 'put' | cell array of character vectors with values 'call' or 'put'

Definition of option, specified as 'call' or 'put' using a character vector.

Data Types: char | cell

Strike — Option strike price values
nonnegative integer

Option strike price value, specified with a NINST-by-1 or NINST-by-NSTRIKES depending on the option type:

- For a European option, use a NINST-by-1 vector of strike prices.
- For a Bermuda option, use aNINST-by-NSTRIKES matrix of strike prices. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.
- For an American option, use a NINST-by-1 of strike prices.

Data Types: double

Settle — Settlement date or trade date
datetime array | string array | date character vector

Settlement date or trade date for the vanilla option, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

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

To support existing code, optstockbystt also accepts serial date numbers as inputs, but they are not recommended.

ExerciseDates — Option exercise dates
datetime array | string array | date character vector

Option exercise dates, specified as a NINST-by-1,NINST-by-2, or NINST-by-NSTRIKES vector using a datetime array, string array, or date character vectors, depending on the option type:

- For a European option, use a NINST-by-1 vector of dates. Each row is the schedule for one option. For a European option, there is only one ExerciseDates on the option expiry date.
- For a Bermuda option, use a NINST-by-NSTRIKES vector of dates. Each row is the schedule for one option.
- For an American option, use a NINST-by-2 vector of exercise date boundaries. The option can be exercised on any date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is a NINST-by-1 vector, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDates.

To support existing code, optstockbystt also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optstockbystt(RateSpec,StockSpec,OptSpec,Strike,Settle,ExerciseDates,'AmericanOpt','1')`

`AmericanOpt` — Option type

`0` European or Bermuda (default) | integer with values of `0` or `1`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a `NINST`-by-`1` vector of integer flags with values:

- `0` — European or Bermuda
- `1` — American

Data Types: `single` | `double`

Output Arguments

`Price` — Expected price of vanilla option at time `0`

vector

Expected price of the vanilla option at time `0`, returned as a `NINST`-by-`1` vector.

`PriceTree` — Structure containing trees of vectors of instrument prices and accrued interest for each node

structure

Structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.dObs` contains the observation dates.

More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Version History

Introduced in R2015b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optstockbystt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`stttree` | `stttimespec` | `sttprice` | `sttsens` | `instoptstock` | `Vanilla`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Vanilla Option” on page 3-27
“Supported Equity Derivative Functions” on page 3-19
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

optpricebysim

Price option given simulated underlying values

Syntax

```
Price = optpricebysim(RateSpec,SimulatedPrices,Times,OptSpec,Strike,
ExerciseTimes)
Price = optpricebysim( ___ ,Name,Value)
```

Description

`Price = optpricebysim(RateSpec,SimulatedPrices,Times,OptSpec,Strike, ExerciseTimes)` calculates the price of European, American, and Berumdan call/put options based on risk-neutral simulation of the underlying asset. For American and Bermudan options, the Longstaff-Schwartz least squares method calculates the early exercise premium.

`Price = optpricebysim( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of an American Option Using Monte Carlo Simulation Based on Geometric Brownian Motion

Define the option.

```
S0 = 100; % Initial price of underlying asset
Sigma = .2; % Volatility of underlying asset
Strike = 110; % Strike
OptSpec = 'call'; % Call option
Settle = datetime(2013,1,1); % Settlement date of option
Maturity = datetime(2014,1,1); % Maturity date of option
r = .05; % Risk-free rate (annual, continuous compounding)
Compounding = -1; % Continuous compounding
Basis = 0; % Act/Act day count convention
T = yearfrac(Settle, Maturity, Basis); % Time to expiration in years
```

Set up the `gbm` object and run the Monte Carlo simulation based on Geometric Brownian Motion (GBM) using the `simBySolution` method from Financial Toolbox™.

```
NTRIALS = 1000;
NPERIODS = daysact(Settle, Maturity);
dt = T/NPERIODS;
OptionGBM = gbm(r, Sigma, 'StartState', S0);
[Paths, Times, Z] = simBySolution(OptionGBM, NPERIODS, ...
'NTRIALS',NTRIALS, 'DeltaTime',dt,'Antithetic',true);
```

Create the interest-rate term structure to define `RateSpec`.

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
           'EndDates', Maturity, 'Rate', r, 'Compounding', Compounding, ...
           'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0500
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Price an American option.

```
SimulatedPrices = squeeze(Paths);
OptPrice = optpricebysim(RateSpec, SimulatedPrices, Times, OptSpec, ...
           Strike, T, 'AmericanOpt', 1)
```

```
OptPrice = 5.8172
```

Compute the Price of an American Asian Option Using Monte Carlo Simulation Based on Geometric Brownian Motion

Define the option.

```
S0 = 100; % Initial price of underlying asset
Sigma = .2; % Volatility of underlying asset
Strike = 110; % Strike
OptSpec = 'call'; % Call option
Settle = datetime(2013,1,1); % Settlement date of option
Maturity = datetime(2014,1,1); % Maturity date of option
r = .05; % Risk-free rate (annual, continuous compounding)
Compounding = -1; % Continuous compounding
Basis = 0; % Act/Act day count convention
T = yearfrac(Settle, Maturity, Basis); % Time to expiration in years
```

Set up the `gbm` object and run the Monte Carlo simulation based on Geometric Brownian Motion (GBM) using the `simBySolution` method from Financial Toolbox™.

```
NTRIALS = 1000;
NPERIODS = daysact(Settle, Maturity);
dt = T/NPERIODS;
OptionGBM = gbm(r, Sigma, 'StartState', S0);
[Paths, Times, Z] = simBySolution(OptionGBM, NPERIODS, ...
'NTRIALS',NTRIALS, 'DeltaTime',dt,'Antithetic',true);
```

Create the interest-rate term structure to define `RateSpec`.

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
           'EndDates', Maturity, 'Rate', r, 'Compounding', Compounding, ...
           'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
```

```
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0500
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Price an American Asian option (arithmetic mean) by finding the average price over periods.

```
AvgPrices = zeros(NPERIODS+1, NTRIALS);
    for i = 1:NPERIODS+1
        AvgPrices(i,:) = mean(squeeze(Paths(1:i,:,:)));
    end
    AsianPrice = optpricebysim(RateSpec, AvgPrices, Times, OptSpec, ...
        Strike, T, 'AmericanOpt', 1)
```

```
AsianPrice = 1.8221
```

Compute the Price of an American Lookback Option Using Monte Carlo Simulation Based on Geometric Brownian Motion

Define the option.

```
S0 = 100; % Initial price of underlying asset
Sigma = .2; % Volatility of underlying asset
Strike = 110; % Strike
OptSpec = 'call'; % Call option
Settle = datetime(2013,1,1); % Settlement date of option
Maturity = datetime(2014,1,1); % Maturity date of option
r = .05; % Risk-free rate (annual, continuous compounding)
Compounding = -1; % Continuous compounding
Basis = 0; % Act/Act day count convention
T = yearfrac(Settle, Maturity, Basis); % Time to expiration in years
```

Set up the `gbm` object and run the Monte Carlo simulation based on Geometric Brownian Motion (GBM) using the `simBySolution` method from Financial Toolbox™.

```
NTRIALS = 1000;
NPERIODS = daysact(Settle, Maturity);
dt = T/NPERIODS;
OptionGBM = gbm(r, Sigma, 'StartState', S0);
[Paths, Times, Z] = simBySolution(OptionGBM, NPERIODS, ...
'NTRIALS',NTRIALS, 'DeltaTime',dt,'Antithetic',true);
```

Create the interest-rate term structure to define `RateSpec`.

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
           'EndDates', Maturity, 'Rate', r, 'Compounding', Compounding, ...
           'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
```

```
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0500
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Price an American lookback option by finding the maximum price over periods.

```
MaxPrices = zeros(NPERIODS+1, NTRIALS);
    LastPrice = squeeze(Paths(1,:,:))';
    for i = 1:NPERIODS+1;
        MaxPrices(i,:) = max([LastPrice; Paths(i,:)]);
        LastPrice = MaxPrices(i,:);
    end
    LookbackPrice = optpricebysim(RateSpec, MaxPrices, Times, OptSpec, ...
        Strike, T, 'AmericanOpt', 1)
```

```
LookbackPrice = 10.4410
```

Compute the Price of a Bermudan Option Using Monte Carlo Simulation Based on Geometric Brownian Motion

Define the option.

```
S0 = 80; % Initial price of underlying asset
Sigma = .3; % Volatility of underlying asset
Strike = 75; % Strike
OptSpec = 'put'; % Put option
Settle = datetime(2013,1,1); % Settlement date of option
Maturity = datetime(2014,1,1); % Maturity date of option
ExerciseDates = [datetime(2013,1,1) , datetime(2014,1,1)]; % Exercise dates of option
r = .05; % Risk-free rate (annual, continuous compounding)
Compounding = -1; % Continuous compounding
Basis = 0; % Act/Act day count convention
T = yearfrac(Settle, Maturity, Basis); % Time to expiration in years
ExerciseTimes = yearfrac(Settle, ExerciseDates, Basis); % Exercise times
```

Set up the `gbm` object and run the Monte Carlo simulation based on Geometric Brownian Motion (GBM) using the `simBySolution` method from Financial Toolbox™.

```
NTRIALS = 1000;
NPERIODS = daysact(Settle, Maturity);
dt = T/NPERIODS;
OptionGBM = gbm(r, Sigma, 'StartState', S0);
[Paths, Times, Z] = simBySolution(OptionGBM, NPERIODS, ...
'NTRIALS',NTRIALS, 'DeltaTime',dt,'Antithetic',true);
```

Create the interest-rate term structure to define `RateSpec`.

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
           'EndDates', Maturity, 'Rate', r, 'Compounding', Compounding, ...
           'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0500
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Price the Bermudan option.

```
SimulatedPrices = squeeze(Paths);
BermudanPrice = optpricebysim(RateSpec, SimulatedPrices, Times, ...
OptSpec, Strike, ExerciseTimes)
```

```
BermudanPrice = 4.9889
```

Compute the Price of an American Spread Option Using Monte Carlo Simulation Based on Geometric Brownian Motion

Define the option.

```
S1 = 110; % Price of first underlying asset
S2 = 100; % Price of second underlying asset
Sigma1 = .1;  % Volatility of first underlying asset
Sigma2 = .15; % Volatility of second underlying asset
Strike = 15; % Strike
Rho = .3; % Correlation between underlyings
OptSpec = 'put'; % Put option
Settle = datetime(2013,1,1); % Settlement date of option
Maturity = datetime(2014,1,1); % Maturity date of option
r = .05; % Risk-free rate (annual, continuous compounding)
Compounding = -1; % Continuous compounding
Basis = 0; % Act/Act day count convention
T = yearfrac(Settle, Maturity, Basis); % Time to expiration in years
```

Set up the `gbm` object and run the Monte Carlo simulation based on Geometric Brownian Motion (GBM) using the `simBySolution` method from Financial Toolbox™.

```
NTRIALS = 1000;
NPERIODS = daysact(Settle, Maturity);
dt = T/NPERIODS;
SpreadGBM = gbm(r*eye(2), diag([Sigma1;Sigma2]),'Correlation',...
[1 Rho;Rho 1],'StartState',[S1;S2]);
[Paths, Times, Z] = simBySolution(SpreadGBM, NPERIODS,'NTRIALS',NTRIALS,...
'DeltaTime',dt,'Antithetic',true);
```

Create the interest-rate term structure to define RateSpec.

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
           'EndDates', Maturity, 'Rate', r, 'Compounding', Compounding, ...
           'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9512
            Rates: 0.0500
         EndTimes: 1
       StartTimes: 0
         EndDates: 735600
       StartDates: 735235
    ValuationDate: 735235
            Basis: 0
     EndMonthRule: 1
```

Price the American spread option.

```
Spread = squeeze(Paths(:,1,:) - Paths(:,2,:));
SpreadPrice = optpricebysim(RateSpec, Spread, Times, OptSpec, Strike, ...
T, 'AmericanOpt', 1)
```

```
SpreadPrice = 9.0007
```

Input Arguments

RateSpec — Interest-rate term structure of risk-free rates

structure

Interest-rate term structure of risk-free rates (annualized and continuously compounded), specified by the RateSpec obtained from intenvset. The valuation date must be at the settlement date of the option, and the day-count basis and end-of-month rule must be the same as those used to calculate the Times input. For information on the interest-rate specification, see intenvset.

Data Types: struct

SimulatedPrices — Simulated prices

matrix

Simulated prices, specified using a (NumPeriods + 1)-by-NumTrials matrix of risk-neutral simulated prices. The first element of SimulatedPrices is the initial value at time 0.

Data Types: double

Times — Annual time factors associated with simulated prices

vector

Annual time factors associated with simulated prices, specified using a (NumPeriods + 1)-by-1 column vector. Each element of Times is associated with the corresponding row of SimulatedPrices. The first element of Times must be 0 (current time).

Data Types: double

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'`

Definition of option as `'call'` or `'put'`, specified as a character vector.

Data Types: `char`

`Strike` — Option strike price values
scalar | function handle

Option strike price values, specified as a scalar value `Strike` price. `Strike` for Bermudan options can be specified as a `1`-by-`NSTRIKES` vector or a function handle that returns the value of the strike given the time of the strike.

Data Types: `double` | `function_handle`

`ExerciseTimes` — Exercise time for option
datetime array | string array | date character vector

Exercise time for the option, specified as a datetime array, string array, or date character vectors, as follows:

- For a European or Bermudan option, `ExerciseTimes` is a `1`-by-`1` (European) or `1`-by-`NSTRIKES` (Bermudan) vector of exercise times. For a European option, there is only one `ExerciseTimes` on the option expiry date.
- For an American option, `ExerciseTimes` is a `1`-by-`2` vector of exercise time boundaries. The option exercises on any date between, or including, the pair of times on that row. If `ExerciseTimes` is `1`-by-`1`, the option exercises between time `0` and the single listed `ExerciseTimes`.

To support existing code, `optpricebysim` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = optpricebysim(RateSpec,Prices,Times,OptSpec,Settle,Strike,ExerciseTimes,'AmericanOpt',1)`

`AmericanOpt` — Option type
`0` European or Bermudan (default) | scalar flag with value `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and an integer scalar flag with values:

- `0` — European or Bermudan
- `1` — American

For American options, the Longstaff-Schwartz least squares method calculates the early exercise premium.

Data Types: `double`

Output Arguments

`Price` — Price of option

scalar

Price of the option, returned as a scalar value.

Version History

Introduced in R2014a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `optpricebysim` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`intenvset` | `gbm` | `simBySolution`

Topics

“Pricing Asian Options” on page 3-110
“Creating Geometric Brownian Motion (GBM) Models”
Supported Equity Derivatives on page 3-19

rangefloatbybdt

Price range floating note using Black-Derman-Toy tree

Syntax

```
[Price,PriceTree] = rangefloatbybdt(BDTTree,Spread,Settle,Maturity,RateSched)
[Price,PriceTree] = rangefloatbybdt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = rangefloatbybdt(BDTTree,Spread,Settle,Maturity,RateSched)` prices range floating note using a Black-Derman-Toy tree.

Payments on range floating notes are determined by the effective interest-rate between reset dates. If the reset period for a range spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates.

`[Price,PriceTree] = rangefloatbybdt( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of a Range Note Using a Black-Derman-Toy Tree

This example shows how to compute the price of a range note using a Black-Derman-Toy tree with the following interest-rate term structure data.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
Compounding = 1;

% define RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates, 'Compounding', Compounding);

% range note instrument matures in Jan-1-2014 and has the following RateSchedule:
Spread = 100;
Settle = datetime(2011,1,1);
Maturity = datetime(2014,1,1);
RateSched(1).Dates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
RateSched(1).Rates = [0.045 0.055 ; 0.0525  0.0675; 0.06 0.08];

% data to build the tree is as follows:
% assume the volatility is 10%.
Sigma = 0.1;
BDTTS = bdttimespec(ValuationDate, EndDates, Compounding);
```

```
BDTVS = bdtvolspec(ValuationDate, EndDates, Sigma*ones(1, length(EndDates))');
BDTT = bdttree(BDTVS, RS, BDTTS);

% price the instrument
Price = rangefloatbybdt(BDTT, Spread, Settle, Maturity, RateSched)

Price = 97.5267
```

Input Arguments

BDTTree — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bdttree`.

Data Types: `struct`

Spread — Number of basis points over reference rate

numeric

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

Settle — Settlement date for floating range note

datetime array | string array | date character vector

Settlement date for the floating range note, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every range floating instrument is set to the `ValuationDate` of the BDT tree. The floating range note argument `Settle` is ignored.

To support existing code, `rangefloatbybdt` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date for floating range note

datetime array | string array | date character vector

Maturity date for the floating-rate note, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `rangefloatbybdt` also accepts serial date numbers as inputs, but they are not recommended.

RateSched — Range of rates within which cash flows are nonzero

structure

Range of rates within which cash flows are nonzero, specified as a `NINST`-by-`1` vector of structures. Each element of the structure array contains two fields:

- `RateSched.Dates` — `NDates`-by-`1` cell array of dates corresponding to the range schedule.
- `RateSched.Rates` — `NDates`-by-`2` array with the first column containing the lower bound of the range and the second column containing the upper bound of the range. Cash flow for date `RateSched.Dates`(*n*) is nonzero for rates in the range `RateSched.Rates`(*n*,1) < `Rate` < `RateSched.Rate` (*n*,2).

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = rangefloatbybdt(BDTTree,Spread,Settle,Maturity,RateSched,'Reset',4,'Basis',5,'Principal',10000)`

`Reset` — Frequency payment per year

`1` (default) | numeric

Frequency of payments per year, specified as the comma-separated pair consisting of `'Reset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount

`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating caplet dates
`1` (in effect) (default) | nonnegative integer with value `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer with a value of `0` or `1` using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`Price` — Expected prices of range floating notes at time 0
vector

Expected prices of the range floating notes at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as a structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

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

More About

Range Note

A range note is a structured (market-linked) security whose coupon rate is equal to the reference rate as long as the reference rate is within a certain range.

If the reference rate is outside of the range, the coupon rate is 0 for that period. This type of instrument entitles the holder to cash flows that depend on the level of some reference interest rate and are floored to be positive. The note holder gets direct exposure to the reference rate. In return for the drawback that no interest is paid for the time the range is left, they offer higher coupon rates than comparable standard products, like vanilla floating notes. For more information, see “Range Note” on page 2-13.

Version History

Introduced in R2012a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `rangefloatbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Jarrow, Robert. “Modelling Fixed Income Securities and Interest Rate Options.” *Stanford Economics and Finance*. 2nd Edition. 2002.

See Also

`bdttree` | `cfbybdt` | `floatbybdt` | `swapbybdt` | `floorbybdt` | `fixedbybdt` | `bondbybdt` | `rangefloatbyhjm` | `instrangefloat` | `rangefloatbyhw` | `rangefloatbybk`

Topics

“Computing Instrument Prices” on page 2-81
“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Range Note” on page 2-13
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

rangefloatbybk

Price range floating note using Black-Karasinski tree

Syntax

```
[Price,PriceTree] = rangefloatbybk(BKTree,Spread,Settle,Maturity,RateSched)
[Price,PriceTree] = rangefloatbybk( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = rangefloatbybk(BKTree,Spread,Settle,Maturity,RateSched)` prices range floating note using a Black-Karasinski tree.

Payments on range floating notes are determined by the effective interest-rate between reset dates. If the reset period for a range spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates.

`[Price,PriceTree] = rangefloatbybk( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of a Range Note Using a Black-Karasinski Tree

This example shows how to compute the price of a range note using a Black-Karasinski tree with the following interest-rate term structure data.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
Compounding = 1;

% define RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates, 'Compounding', Compounding);

% range note instrument matures in Jan-1-2014 and has the following RateSchedule:
Spread = 100;
Settle = datetime(2011,1,1);
Maturity = datetime(2014,1,1);
RateSched(1).Dates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
RateSched(1).Rates = [0.045 0.055 ; 0.0525  0.0675; 0.06 0.08];

% data to build the tree is as follows:
VolDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2015,1,1);
```

```
AlphaCurve = 0.1;

BKVS = bkvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTS = bktimespec(RS.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVS, RS, BKTS);

% price the instrument
Price = rangefloatbybk(BKT, Spread, Settle, Maturity, RateSched)

Price = 96.8542
```

Input Arguments

BKTree — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `bktree`.

Data Types: `struct`

Spread — Number of basis points over reference rate

numeric

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

Settle — Settlement date for floating range note

datetime array | string array | date character vector

Settlement date for the floating range note, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every range floating instrument is set to the `ValuationDate` of the BK tree. The floating range note argument `Settle` is ignored.

To support existing code, `rangefloatbybk` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date for floating range note

datetime array | string array | date character vector

Maturity date for the floating-rate note, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `rangefloatbybk` also accepts serial date numbers as inputs, but they are not recommended.

RateSched — Range of rates within which cash flows are nonzero

structure

Range of rates within which cash flows are nonzero, specified as a `NINST`-by-`1` vector of structures. Each element of the structure array contains two fields:

- `RateSched.Dates` — `NDates`-by-`1` cell array of dates corresponding to the range schedule.
- `RateSched.Rates` — `NDates`-by-`2` array with the first column containing the lower bound of the range and the second column containing the upper bound of the range. Cash flow for date

`RateSched.Dates`(*n*) is nonzero for rates in the range `RateSched.Rates`(*n*,1) < `Rate` < `RateSched.Rate` (*n*,2).

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = rangefloatbybk(BKTree,Spread,Settle,Maturity,RateSched,'Reset',4,'Basis',5,'Principal',10000)`

`Reset` — Frequency payment per year
`1` (default) | numeric

Frequency of payments per year, specified as the comma-separated pair consisting of `'Reset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating caplet dates
`1` (in effect) (default) | nonnegative integer with value `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer with a value of `0` or `1` using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`Price` — Expected prices of range floating notes at time 0
vector

Expected prices of the range floating notes at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as a structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

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

More About

Range Note

A range note is a structured (market-linked) security whose coupon rate is equal to the reference rate as long as the reference rate is within a certain range.

If the reference rate is outside of the range, the coupon rate is 0 for that period. This type of instrument entitles the holder to cash flows that depend on the level of some reference interest rate and are floored to be positive. The note holder gets direct exposure to the reference rate. In return for the drawback that no interest is paid for the time the range is left, they offer higher coupon rates than comparable standard products, like vanilla floating notes. For more information, see “Range Note” on page 2-13.

Version History

Introduced in R2012a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `rangefloatbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Jarrow, Robert. “Modelling Fixed Income Securities and Interest Rate Options.” *Stanford Economics and Finance*. 2nd Edition. 2002.

See Also

`bktree` | `cfbybk` | `capbybk` | `swapbybk` | `floorbybk` | `fixedbybk` | `bondbybk` | `rangefloatbyhjm` | `rangefloatbybdt` | `rangefloatbyhw` | `instrangefloat`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Range Note” on page 2-13
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

rangefloatbycir

Price range floating note using Cox-Ingersoll-Ross tree

Syntax

```
[Price,PriceTree] = rangefloatbycir(CIRTree,Spread,Settle,Maturity,RateSched)
[Price,PriceTree] = rangefloatbycir( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = rangefloatbycir(CIRTree,Spread,Settle,Maturity,RateSched)` prices range floating note with a Cox-Ingersoll-Ross (CIR) interest-rate tree using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = rangefloatbycir( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price a Range Floating-Rate Note Using a CIR Interest-Rate Tree

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = {'Jan-1-2017'; 'Jan-1-2018'; 'Jan-1-2019'; 'Jan-1-2020'; 'Jan-1-2021'};
ValuationDate = 'Jan-1-2017';
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',End
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = '01-Jan-2017';
Maturity = '01-Jan-2020';
CIRTimeSpec = cirtimespec(Settle, Maturity, 3);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2]
        dObs: [736696 737061 737426]
     FwdTree: {[1.0350]  [1.0790 1.0500 1.0298]  [1.1275 1.0887 1.0594 ... ]}
```

```
     Connect: {[3x1 double]  [3x3 double]}
       Probs: {[3x1 double]  [3x3 double]}
```

Define the range note instrument that matures in Jan-1-2014 and has the following RateSchedule:

```
Spread = 100;
Settle = 'Jan-1-2017';
Maturity = 'Jan-1-2020';
RateSched(1).Dates = {'Jan-1-2018'; 'Jan-1-2019'  ; 'Jan-1-2020'};
RateSched(1).Rates = [0.045 0.055 ; 0.0525  0.0675; 0.06 0.08];
```

Compute the price of the range floating note.

```
[Price,PriceTree] = rangefloatbycir(CIRT,Spread,Settle,Maturity,RateSched)
```

```
Price = 91.6849
```

```
PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
      PTree: {1x4 cell}
     AITree: {[0]  [0 0 0]  [0 0 0 0 0]  [0 0 0 0 0]}
       tObs: [0 1 2 3]
    Connect: {[3x1 double]  [3x3 double]}
      Probs: {[3x1 double]  [3x3 double]}
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `cirtree`.

Data Types: `struct`

`Spread` — Number of basis points over reference rate

numeric

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date for floating range note

datetime array | string array | date character vector

Settlement date for the floating range note, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every range floating instrument is set to the `ValuationDate` of the CIR tree. The floating range note argument `Settle` is ignored.

To support existing code, `rangefloatbycir` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for floating range note

datetime array | string array | date character vector

Maturity date for the floating-rate note, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `rangefloatbycir` also accepts serial date numbers as inputs, but they are not recommended.

`RateSched` — Range of rates within which cash flows are nonzero
structure

Range of rates within which cash flows are nonzero, specified as a `NINST`-by-`1` vector of structures. Each element of the structure array contains two fields:

- `RateSched.Dates` — `NDates`-by-`1` cell array of dates corresponding to the range schedule.
- `RateSched.Rates` — `NDates`-by-`2` array with the first column containing the lower bound of the range and the second column containing the upper bound of the range. Cash flow for date `RateSched.Dates`(*n*) is nonzero for rates in the range `RateSched.Rates`(*n*,1) < `Rate` < `RateSched.Rate` (*n*,2).

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = rangefloatbycir(CIRTree,Spread,Settle,Maturity,RateSched,'Reset',4,'Basis',5,'Principal',10000)`

`Reset` — Frequency payment per year
`1` (default) | numeric

Frequency of payments per year, specified as the comma-separated pair consisting of `'Reset'` and a `NINST`-by-`1` vector.

Note Payments on range floating notes are determined by the effective interest-rate between reset dates. If the reset period for a range spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

- 0 = actual/actual

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating caplet dates
`1` (in effect) (default) | nonnegative integer with value `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer with a value of `0` or `1` using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`Price` — Expected prices of range floating notes at time 0
vector

Expected prices of the range floating notes at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as a structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.AITree` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

Range Note

A range note is a structured (market-linked) security whose coupon rate is equal to the reference rate as long as the reference rate is within a certain range.

If the reference rate is outside of the range, the coupon rate is `0` for that period. This type of instrument entitles the holder to cash flows that depend on the level of some reference interest rate and are floored to be positive. The note holder gets direct exposure to the reference rate. In return for the drawback that no interest is paid for the time the range is left, they offer higher coupon rates than comparable standard products, vanilla floating notes. For more information, see “Range Note” on page 2-13.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `rangefloatbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J., and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica.* Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `swapbycir` | `swaptionbycir` | `instrangefloat`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Range Note” on page 2-13
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

rangefloatbyhjm

Price range floating note using Heath-Jarrow-Morton tree

Syntax

```
[Price,PriceTree] = rangefloatbyhjm(HJMTree,Spread,Settle,Maturity,RateSched)
[Price,PriceTree] = rangefloatbyhjm(___,Name,Value)
```

Description

`[Price,PriceTree] = rangefloatbyhjm(HJMTree,Spread,Settle,Maturity,RateSched)` prices range floating note using a Heath-Jarrow-Morton tree.

Payments on range floating notes are determined by the effective interest-rate between reset dates. If the reset period for a range spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates.

`[Price,PriceTree] = rangefloatbyhjm(___,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of a Range Note Using a Heath-Jarrow-Morton Tree

This example shows how to compute the price of a range note using a Heath-Jarrow-Morton tree with the following interest-rate term structure data.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
Compounding = 1;

% define RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates, 'Compounding', Compounding);

% range note instrument matures in Jan-1-2014 and has the following RateSchedule:
Spread = 100;
Settle = datetime(2011,1,1);
Maturity = datetime(2014,1,1);
RateSched(1).Dates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
RateSched(1).Rates = [0.045 0.055 ; 0.0525  0.0675; 0.06 0.08];

% data to build the tree is as follows:
Volatility = [.2; .19; .18; .17];
CurveTerm = [ 1;  2;   3;   4];
MaTree = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
```

```
HJMTS = hjmtimespec(ValuationDate, MaTree);
HJMVS = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
HJMT = hjmtree(HJMVS, RS, HJMTS);

% price the instrument
Price = rangefloatbyhjm(HJMT, Spread, Settle, Maturity, RateSched)

Price = 90.2348
```

Input Arguments

`HJMTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hjmtree`.

Data Types: `struct`

`Spread` — Number of basis points over reference rate

numeric

Number of basis points over the reference rate, specified as a `NINST`-by-1 vector.

Data Types: `double`

`Settle` — Settlement date for floating range note

datetime array | string array | date character vector

Settlement date for the floating range note, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors. The `Settle` date for every range floating instrument is set to the `ValuationDate` of the HJM tree. The floating range note argument `Settle` is ignored.

To support existing code, `rangefloatbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for floating range note

datetime array | string array | date character vector

Maturity date for the floating-rate note, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `rangefloatbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`RateSched` — Range of rates within which cash flows are nonzero

structure

Range of rates within which cash flows are nonzero, specified as a `NINST`-by-1 vector of structures. Each element of the structure array contains two fields:

- `RateSched.Dates` — `NDates`-by-1 cell array of dates corresponding to the range schedule.
- `RateSched.Rates` — `NDates`-by-2 array with the first column containing the lower bound of the range and the second column containing the upper bound of the range. Cash flow for date `RateSched.Dates`(*n*) is nonzero for rates in the range `RateSched.Rates`(*n*,1) < `Rate` < `RateSched.Rate` (*n*,2).

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = rangefloatbyhjm(HJMTree,Spread,Settle,Maturity,RateSched,'Reset',4,'Basis',5,'Principal',10000)`

`Reset` — Frequency payment per year

`1` (default) | numeric

Frequency of payments per year, specified as the comma-separated pair consisting of `'Reset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount

`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating caplet dates
`1` (in effect) (default) | nonnegative integer with value `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer with a value of `0` or `1` using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`Price` — Expected prices of range floating notes at time 0
vector

Expected prices of the range floating notes at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as a structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

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

More About

Range Note

A range note is a structured (market-linked) security whose coupon rate is equal to the reference rate as long as the reference rate is within a certain range.

If the reference rate is outside of the range, the coupon rate is 0 for that period. This type of instrument entitles the holder to cash flows that depend on the level of some reference interest rate and are floored to be positive. The note holder gets direct exposure to the reference rate. In return for the drawback that no interest is paid for the time the range is left, they offer higher coupon rates than comparable standard products, like vanilla floating notes. For more information, see “Range Note” on page 2-13.

Version History

Introduced in R2012a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `rangefloatbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Jarrow, Robert. “Modelling Fixed Income Securities and Interest Rate Options.” *Stanford Economics and Finance*. 2nd Edition. 2002.

See Also

`hjmtree` | `cfbyhjm` | `floatbyhjm` | `swapbyhjm` | `floorbyhjm` | `fixedbyhjm` | `bondbyhjm` | `rangefloatbybk` | `rangefloatbyhw` | `instrangefloat` | `rangefloatbybdt`

Topics

“Computing Instrument Prices” on page 2-81
“Range Note” on page 2-13
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

rangefloatbyhw

Price range floating note using Hull-White tree

Syntax

```
[Price,PriceTree] = rangefloatbyhw(HWTree,Spread,Settle,Maturity,RateSched)
[Price,PriceTree] = rangefloatbyhw( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = rangefloatbyhw(HWTree,Spread,Settle,Maturity,RateSched)` prices range floating note using a Hull-White tree.

Payments on range floating notes are determined by the effective interest-rate between reset dates. If the reset period for a range spans more than one tree level, calculating the payment becomes impossible due to the recombining nature of the tree. That is, the tree path connecting the two consecutive reset dates cannot be uniquely determined because there is more than one possible path for connecting the two payment dates.

`[Price,PriceTree] = rangefloatbyhw( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price of a Range Note Using a Hull-White Tree

This example shows how to compute the price of a range note using a Hull-White tree with the following interest-rate term structure data.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
Compounding = 1;

% define RateSpec
RS = intenvset('ValuationDate', ValuationDate, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates, 'Compounding', Compounding);

% range note instrument matures in Jan-1-2014 and has the following RateSchedule:
Spread = 100;
Settle = datetime(2011,1,1);
Maturity = datetime(2014,1,1);
RateSched(1).Dates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1)];
RateSched(1).Rates = [0.045 0.055 ; 0.0525  0.0675; 0.06 0.08];

% data to build the tree is as follows:
VolDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1)];
VolCurve = 0.01;
AlphaDates = datetime(2015,1,1);
```

```
AlphaCurve = 0.1;

HWVS = hwvolspec(RS.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTS = hwtimespec(RS.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVS, RS, HWTS);

% price the instrument
Price = rangefloatbyhw(HWT, Spread, Settle, Maturity, RateSched)

Price = 96.6107
```

Input Arguments

`HWTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hwtree`.

Data Types: `struct`

`Spread` — Number of basis points over reference rate

numeric

Number of basis points over the reference rate, specified as a `NINST`-by-1 vector.

Data Types: `double`

`Settle` — Settlement date for floating range note

datetime array | string array | date character vector

Settlement date for the floating range note, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors. The `Settle` date for every range floating instrument is set to the `ValuationDate` of the HW tree. The floating range note argument `Settle` is ignored.

To support existing code, `rangefloatbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for floating range note

datetime array | string array | date character vector

Maturity date for the floating-rate note, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `rangefloatbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`RateSched` — Range of rates within which cash flows are nonzero

structure

Range of rates within which cash flows are nonzero, specified as a `NINST`-by-1 vector of structures. Each element of the structure array contains two fields:

- `RateSched.Dates` — `NDates`-by-1 cell array of dates corresponding to the range schedule.
- `RateSched.Rates` — `NDates`-by-2 array with the first column containing the lower bound of the range and the second column containing the upper bound of the range. Cash flow for date

`RateSched.Dates`(*n*) is nonzero for rates in the range `RateSched.Rates`(*n*,1) < `Rate` < `RateSched.Rate` (*n*,2).

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = rangefloatbyhw(HWTree,Spread,Settle,Maturity,RateSched,'Reset',4,'Basis',5,'Principal',10000)`

`Reset` — Frequency payment per year
`1` (default) | numeric

Frequency of payments per year, specified as the comma-separated pair consisting of `'Reset'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward rate tree, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector of integers.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating caplet dates
`1` (in effect) (default) | nonnegative integer with value `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer with a value of `0` or `1` using a `NINST`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`Price` — Expected prices of range floating notes at time 0
vector

Expected prices of the range floating notes at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as a structure containing trees of vectors of instrument prices and accrued interest, and a vector of observation times for each node. Values are:

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

More About

Range Note

A range note is a structured (market-linked) security whose coupon rate is equal to the reference rate as long as the reference rate is within a certain range.

If the reference rate is outside of the range, the coupon rate is 0 for that period. This type of instrument entitles the holder to cash flows that depend on the level of some reference interest rate and are floored to be positive. The note holder gets direct exposure to the reference rate. In return for the drawback that no interest is paid for the time the range is left, they offer higher coupon rates than comparable standard products, like vanilla floating notes. For more information, see “Range Note” on page 2-13.

Version History

Introduced in R2012a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `rangefloatbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Jarrow, Robert. “Modelling Fixed Income Securities and Interest Rate Options.” *Stanford Economics and Finance*. 2nd Edition. 2002.

See Also

`hwtree` | `cfbyhw` | `capbyhw` | `swapbyhw` | `floorbyhw` | `fixedbyhw` | `bondbyhw` | `rangefloatbybk` | `rangefloatbybdt` | `rangefloatbyhjm` | `instrangefloat`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Range Note” on page 2-13
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

rate2disc

Discount factors from interest rates

Syntax

```
Disc = rate2disc(Compounding,Rates,EndTimes,StartTimes)
[Disc,EndTimes,StartTimes] = rate2disc(Compounding,Rates,EndTimes,StartTimes,
ValuationDate,Basis,EndMonthRule)
```

Description

`Disc = rate2disc(Compounding,Rates,EndTimes,StartTimes)` computes discount factors from interest rates where interval points are input as times in periodic units.

The `rate2disc` function 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.

The output `Disc` is an `NPOINTS`-by-`NCURVES` column vector of discount factors in decimal form representing the value at time `StartTimes` of a unit cash flow received at time `EndTimes`.

`[Disc,EndTimes,StartTimes] = rate2disc(Compounding,Rates,EndTimes,StartTimes, ValuationDate,Basis,EndMonthRule)` computes discount factors from interest rates where `ValuationDate` is passed and interval points are input as dates.

You can specify the investment intervals either with input times or with input dates. Entering `ValuationDate` invokes the date interpretation; omitting `ValuationDate` invokes the default time interpretations.

Examples

Compute Discount Factors From Interest Rates

This example shows the two uses of `rate2disc`.

Interval Points Are Input as Times in Periodic Units

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. Use `rate2disc` to compute 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 = 3×1

    0.9756
    0.9426
```

```
    0.8799
```

Interval Points Are Input as Dates

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 = 3×1

    0.9756
    0.9426
    0.8799
```

Input Arguments

`Compounding` — Compounding rate

integer with value of `0`, `1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding rate for which the input zero rates are compounded when annualized, specified as one of the following scalar integers. Compounding determines the formula for the discount factors (`Disc`):

- If `Compounding` = `0` for simple interest:
  - `Disc = 1/(1 + Z * T)`, where `T` is time in years and simple interest assumes annual times `F = 1`.
- If `Compounding` = `1`, `2`, `3`, `4`, `6`, `12`:
  - `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units, for example, `T = F` is one year.
- If `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.
- If `Compounding` = `-1`:
  - `Disc = exp(-T*Z)`, where `T` is time in years.

Data Types: `double`

`Rates` — Rates

decimal

Rates, specified as a number of points (`NPOINTS`) by number of curves (`NCURVES`) matrix of rates in decimal form. `Rates` are the yields over investment intervals from `StartTimes`, when the cash flow is valued, to `EndTimes`, when the cash flow is received. For example, 5% is `0.05` in `Rates`.

Data Types: `double`

`EndTimes` — End times

serial date number | date character vector | numeric

End times, specified as a scalar or an `NPOINTS`-by-`1` column vector using serial date numbers or date character vectors, or with times in periodic units ending the interval to discount over. When `EndTimes` is not a date, the value for `EndTimes` is `T` computed from SIA semi-annual time factors, `Tsemi`, by the formula `T = Tsemi/2 * F`, where `F` is the compounding frequency. `F` is set to `1` for continuous compounding.

Note When `ValuationDate` is not passed, `EndTimes` is interpreted as times. If `Compounding = 365` (daily), `EndTimes` is measured in days.

Data Types: `double` | `char`

`StartTimes` — Start times

serial date number | date character vector | numeric

Start times, specified a scalar or an `NPOINTS`-by-`1` column vector using serial date numbers or date character vectors, or with times in periodic units starting the interval to discount over. `StartDates` must be earlier than `EndDates`. When `StarTimes` is not a date, the value for `StartTimes` is `T` computed from SIA semi-annual time factors, `Tsemi`, by the formula `T = Tsemi/2 * F`, where `F` is the compounding frequency. `F` is set to `1` for continuous compounding.

Note When `ValuationDate` is not passed, `StartTimes` is interpreted as times. If `Compounding = 365` (daily), `StartTimes` is measured in days.

Data Types: `double` | `char`

`ValuationDate` — Observation date of the investment horizons entered in `StartTimes` and `EndTimes`

serial date number | date character vector

Observation date of the investment horizons entered in `StartTimes` and `EndTimes`, specified as scalar serial date number or date character vector.

Note You can specify the investment intervals either with input times or with input dates. Entering `ValuationDate` invokes the date interpretation; omitting `ValuationDate` invokes the default time interpretations.

Data Types: `double` | `char`

`Basis` — Day-count basis

`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis of the instrument when using dates for `StartTimes` and `EndTimes`, specified as a scalar or an `NINST`-by-`1` vector of integers..

- 0 = actual/actual
- 1 = 30/360 (SIA)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with values `0` or `1`

End-of-month rule flag when using dates for `StartTimes` and `EndTimes`, specified as a scalar or an `NINST`-by-`1` vector of nonnegative integers. 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

Output Arguments

`Disc` — Discount factors
vector

Discount factors, returned as an `NPOINTS`-by-`NCURVES` column vector of in decimal form representing the value at time `StartTimes` of a unit cash flow received at time `EndTimes`.

`EndTimes` — Times ending the interval to discount over
vector

Times ending the interval to discount over, returned as an `NPOINTS`-by-`1` column vector, measured in periodic units.

`StartTimes` — Times starting the interval to discount over
vector

Times starting the interval to discount over, returned as an `NPOINTS`-by-`1` column vector, measured in periodic units.

Version History

Introduced before R2006a

See Also

`disc2rate` | `ratetimes`

Topics

“Modeling the Interest-Rate Term Structure” on page 2-57
“Interest-Rate Term Conversions” on page 2-53
“Interest Rates Versus Discount Factors” on page 2-48
“Understanding the Interest-Rate Term Structure” on page 2-48

ratetimes

Change time intervals defining interest-rate environment

Syntax

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

Description

`[Rates,EndTimes,StartTimes] = ratetimes(Compounding,RefRates,RefEndTimes, RefStartTimes,EndTimes,StartTimes)` change time intervals defining an interest-rate environment where interval points are input as times in periodic units.

`ratetimes` takes an interest-rate environment defined by yields over one collection of time intervals and computes the yields over another set of time intervals. The zero rate is assumed to be piecewise linear in time.

`[Rates,EndTimes,StartTimes] = ratetimes(Compounding,RefRates,RefEndTimes, RefStartTimes,EndTimes,StartTimes,ValuationDate)` change time intervals defining an interest-rate environment where `ValuationDate` is passed and interval points are input as dates.

Entering `ValuationDate` invokes the date interpretation; omitting `ValuationDate` invokes the default time interpretations.

Examples

Change Time Intervals Defining Interest-Rate Environment

`ratetimes` takes an interest-rate environment defined by yields over one collection of time intervals and computes the yields over another set of time intervals. The zero rate is assumed to be piecewise linear in time.

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 = 3×1

    0.0600
    0.0688
    0.0700
```

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 = 2×1

    0.0520
    0.0520
```

Input Arguments

`Compounding` — Compounding rate

integer with value of `0`, `1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding rate for which the input zero rates are compounded when annualized, specified as one of the following scalar integers. Compounding determines the formula for the discount factors (`Disc`):

- If `Compounding` = `0` for simple interest:
  - `Disc = 1/(1 + Z * T)`, where `T` is time in years and simple interest assumes annual times `F = 1`.
- If `Compounding` = `1`, `2`, `3`, `4`, `6`, `12`:
  - `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units, for example, `T = F` is one year.
- If `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.
- If `Compounding` = `-1`:
  - `Disc = exp(-T*Z)`, where `T` is time in years.

Data Types: `double`

`RefRates` — Reference rates

decimal

Reference rates, specified as an `NREFPTS`-by-`NCURVES` matrix 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.

Data Types: `double`

`RefEndTimes` — End times

serial date number | date character vector

End times, specified as a scalar or an NREFPTS-by-1 vector using serial date numbers or date character vectors, or with times in periodic units ending the intervals corresponding to RefRates. When RefEndTimes is not a date, the value for RefEndTimes is T computed from SIA semi-annual time factors, Tsemi, by the formula T = Tsemi/2 * F, where F is the compounding frequency. F is set to 1 for continuous compounding.

Note When ValuationDate is not passed, RefEndTimes is interpreted as times. If Compounding = 365 (daily), RefEndTimes is measured in days.

Data Types: double | char

RefStartTimes — Start times
0 (default) | serial date number | date character vector

(Optional) Start times, specified a scalar or an NREFPTS-by-1 vector using serial date numbers or date character vectors, or with times in periodic units starting the intervals corresponding to RefRates. RefStartDates must be earlier than RefEndDates. When RefStarTimes is not a date, the value for RefStartTimes is T computed from SIA semi-annual time factors, Tsemi, by the formula T = Tsemi/2 * F, where F is the compounding frequency. F is set to 1 for continuous compounding.

Note When ValuationDate is not passed, RefStartTimes is interpreted as times. If Compounding = 365 (daily), RefStartTimes is measured in days.

(Optional) NREFPTS-by-1 vector or scalar of times in periodic units starting the intervals corresponding to RefRates. Default = 0.

Data Types: double | char

EndTimes — End of interval where rates are desired
serial date number | date character vector

End of interval where rates are desired, specified as a scalar or NPOINTS-by-1 vector using serial date numbers or date character vectors.

Note You can specify the investment intervals either with input times or with input dates. Entering ValuationDate invokes the date interpretation; omitting ValuationDate invokes the default time interpretations.

Data Types: double | char

StartTimes — Starting new interval where rates are desired
0 (default) | serial date number | date character vector

(Optional) Starting new interval where rates are desired, specified as a scalar or NPOINTS-by-1 vector using serial date numbers or date character vectors.

Note You can specify the investment intervals either with input times or with input dates. Entering `ValuationDate` invokes the date interpretation; omitting `ValuationDate` invokes the default time interpretations.

Data Types: `double` | `char`

`ValuationDate` — Observation date of the investment horizons entered in `RefStartTimes` and `RefEndTimes`
serial date number | date character vector

(Optional) Observation date of the investment horizons entered in `RefStartTimes` and `RefEndTimes`, specified as a serial date number or data character vector.

Note You can specify the investment intervals either with input times or with input dates. Entering `ValuationDate` invokes the date interpretation; omitting `ValuationDate` invokes the default time interpretations.

Data Types: `double` | `char`

Output Arguments

`Rates` — Rates implied by reference interest-rate structure and sampled at new intervals
matrix

Rates implied by the reference interest-rate structure and sampled at new intervals, returned as an `NPOINTS`-by-`NCURVES`.

`EndTimes` — Times ending the new intervals
vector

Times ending the new intervals, returned as an `NPOINTS`-by-`1` column vector, measured in periodic units.

`StartTimes` — Times starting the new intervals where rates are desired
vector

Times starting the new intervals where rates are desired, returned as an `NPOINTS`-by-`1` column vector, measured in periodic units.

Version History

Introduced before R2006a

See Also

`disc2rate` | `rate2disc`

Topics

“Modeling the Interest-Rate Term Structure” on page 2-57
“Interest-Rate Term Conversions” on page 2-53
“Interest Rates Versus Discount Factors” on page 2-48

“Understanding the Interest-Rate Term Structure” on page 2-48

spreadbykirk

Price European spread options using Kirk pricing model

Syntax

```
Price = spreadbykirk(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,
Strike,Corr)
```

Description

`Price = spreadbykirk(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec, Strike,Corr)` returns the price for a European spread option using the Kirk pricing model.

Note Alternatively, you can use the `Spread` object to price spread options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Compute the Price of a Spread Option Using the Kirk Model

Define the spread option dates.

```
Settle = datetime(2012,1,1);
Maturity = datetime(2012,4,1);
```

Define asset 1. Price and volatility of RBOB gasoline

```
   Price1gallon = 2.85;          % $/gallon
   Price1 = Price1gallon * 42;   % $/barrel
   Vol1 = 0.29;
```

Define asset 2. Price and volatility of WTI crude oil

```
  Price2 = 93.20;         % $/barrel
  Vol2 = 0.36;
```

Define the correlation between the underlying asset prices of asset 1 and asset 2.

```
Corr = 0.42;
```

Define the spread option.

```
OptSpec = 'call';
Strike = 20;
```

Define the `RateSpec`.

```
rates = 0.05;
Compounding = -1;
```

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9876
            Rates: 0.0500
         EndTimes: 0.2500
       StartTimes: 0
         EndDates: 734960
       StartDates: 734869
    ValuationDate: 734869
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec` for the two assets.

```
StockSpec1 = stockspec(Vol1, Price1)
```

```
StockSpec1 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2900
         AssetPrice: 119.7000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

```
StockSpec2 = stockspec(Vol2, Price2)
```

```
StockSpec2 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3600
         AssetPrice: 93.2000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Compute the European spread option price based on the Kirk model.

```
Price = spreadbykirk(RateSpec, StockSpec1, StockSpec2, Settle, ...
Maturity, OptSpec, Strike, Corr)
```

```
Price = 11.1904
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec1` — Stock specification for underlying asset 1
structure

Stock specification for underlying asset 1. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`StockSpec2` — Stock specification for underlying asset 2
structure

Stock specification for underlying asset 2. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement dates for spread option
datetime array | string array | date character vector

Settlement dates for the spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadbykirk` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for spread option
datetime array | string array | date character vector

Maturity date for spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadbykirk` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `cell` | `char`

`Strike` — Option strike price values
integer | vector of integers

Option strike price values, specified as an integer using a `NINST`-by-`1` vector of strike price values.

If `Strike` is equal to `0`, the function computes the price of an exchange option.

Data Types: `double`

`Corr` — Correlation between underlying asset prices
integer | vector of integers

Correlation between underlying asset prices, specified as an integer using a `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected prices of spread option
vector

Expected prices of the spread option, returned as a `NINST`-by-`1` vector.

More About

Spread Option

A spread option is an option written on the difference of two underlying assets.

For example, a European call on the difference of two assets *X1* and *X2* would have the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

where:

K is the strike price.

For more information, see “Spread Option” on page 3-30.

Version History

Introduced in R2013b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `spreadbykirk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Carmona, R., Durrleman, V. “Pricing and Hedging Spread Options.” *SIAM Review.* Vol. 45, No. 4, pp. 627–685, Society for Industrial and Applied Mathematics, 2003.

See Also

`spreadsensbykirk` | `spreadbybjs` | `spreadbyfd` | `spreadbyls` | `Spread`

Topics

“Pricing European and American Spread Options” on page 3-97
“Pricing Asian Options” on page 3-110
“Spread Option” on page 3-30
“Supported Energy Derivative Functions” on page 3-34
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

Energy Trading & Risk Management with MATLAB (47 min 31 sec)

spreadbybjs

Price European spread options using Bjerksund-Stensland pricing model

Syntax

```
Price = spreadbybjs(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,
Strike,Corr)
```

Description

`Price = spreadbybjs(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec, Strike,Corr)` returns the price for a European spread option using the Bjerksund-Stensland pricing model.

Note Alternatively, you can use the `Spread` object to price spread options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Compute the Price of a Spread Option Using the Bjerksund-Stensland Model

Define the spread option dates.

```
Settle = datetime(2012,1,1);
Maturity = datetime(2012,4,1);
```

Define asset 1. Price and volatility of RBOB gasoline

```
  Price1gallon = 2.85;          % $/gallon
  Price1 = Price1gallon * 42;   % $/barrel
  Vol1 = 0.29;
```

Define asset 2. Price and volatility of WTI crude oil

```
   Price2 = 93.20;         % $/barrel
   Vol2 = 0.36;
```

Define the correlation between the underlying asset prices of asset 1 and asset 2.

```
Corr = 0.42;
```

Define the spread option.

```
OptSpec = 'call';
Strike = 20;
```

Define the `RateSpec`.

```
rates = 0.05;
Compounding = -1;
```

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9876
            Rates: 0.0500
         EndTimes: 0.2500
       StartTimes: 0
         EndDates: 734960
       StartDates: 734869
    ValuationDate: 734869
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec` for the two assets.

```
StockSpec1 = stockspec(Vol1, Price1)
```

```
StockSpec1 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2900
         AssetPrice: 119.7000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

```
StockSpec2 = stockspec(Vol2, Price2)
```

```
StockSpec2 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3600
         AssetPrice: 93.2000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Compute the European spread option price based on the Bjerksund-Stensland model.

```
Price = spreadbybjs(RateSpec, StockSpec1, StockSpec2, Settle, ...
Maturity, OptSpec, Strike, Corr)
```

```
Price = 11.2000
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec1` — Stock specification for underlying asset 1
structure

Stock specification for underlying asset 1. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`StockSpec2` — Stock specification for underlying asset 2
structure

Stock specification for underlying asset 2. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement dates for spread option
datetime array | string array | date character vector

Settlement dates for the spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadbybjs` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for spread option
datetime array | string array | date character vector

Maturity date for spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadbybjs` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
integer | vector of integers

Option strike price values, specified as an integer using a `NINST`-by-`1` vector of strike price values.

If `Strike` is equal to zero, the function computes the price of an exchange option.

Data Types: `double`

`Corr` — Correlation between underlying asset prices

integer | vector of integers

Correlation between underlying asset prices, specified as an integer using a `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected prices of spread option

vector

Expected prices of the spread option, returned as a `NINST`-by-`1` vector.

More About

Spread Option

A spread option is an option written on the difference of two underlying assets.

For example, a European call on the difference of two assets *X1* and *X2* would have the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

where:

K is the strike price.

For more information, see “Spread Option” on page 3-30.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `spreadbybjs` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Carmona, R., Durrleman, V. “Pricing and Hedging Spread Options.” *SIAM Review.* Vol. 45, No. 4, pp. 627–685, Society for Industrial and Applied Mathematics, 2003.

[2] Bjerksund, Petter, Stensland, Gunnar. *“Closed form spread option valuation.”* Department of Finance, NHH, 2006.

See Also

`spreadsensbykirk` | `spreadbybjs` | `spreadbyfd` | `spreadbyls` | `Spread`

Topics

“Pricing European and American Spread Options” on page 3-97
“Pricing Asian Options” on page 3-110
“Spread Option” on page 3-30
“Supported Energy Derivative Functions” on page 3-34
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

Energy Trading & Risk Management with MATLAB (47 min 31 sec)

spreadbyfd

Price European or American spread options using finite difference method

Syntax

```
Price = spreadbyfd(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,
Strike,Corr)
Price = spreadbyfd( ___ ,Name,Value)

[Price,PriceGrid,AssetPrice1,AssetPrice2,Times] = spreadbyfd(RateSpec,
StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr)
[Price,PriceGrid,AssetPrice1,AssetPrice2,Times] = spreadbyfd( ___ ,Name,Value)
```

Description

`Price = spreadbyfd(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec, Strike,Corr)` returns the price of European or American call or put spread options using the Alternate Direction Implicit (ADI) finite difference method. The spread is between the asset defined in `StockSpec1` minus the asset defined in `StockSpec2`.

`Price = spreadbyfd( ___ ,Name,Value)` adds optional name-value pair arguments.

`[Price,PriceGrid,AssetPrice1,AssetPrice2,Times] = spreadbyfd(RateSpec, StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr)` returns the `Price`, `PriceGrid`, `AssetPrice1`, `AssetPrice2`, and `Times` for a European or American call or put spread options using the Alternate Direction Implicit (ADI) finite difference method. The spread is between the asset defined in `StockSpec1` minus the asset defined in `StockSpec2`.

`[Price,PriceGrid,AssetPrice1,AssetPrice2,Times] = spreadbyfd( ___ ,Name,Value)` returns the `Price`, `PriceGrid`, `AssetPrice1`, `AssetPrice2`, and `Times` and adds optional name-value pair arguments.

Examples

Compute the Price of a Spread Option Using the Alternate Direction Implicit (ADI) Finite Difference Method

Define the spread option dates.

```
Settle = datetime(2012,1,1);
Maturity = datetime(2012,4,1);
```

Define asset 1. Price and volatility of RBOB gasoline

```
  Price1gallon = 2.85;          % $/gallon
  Price1 = Price1gallon * 42;   % $/barrel
  Vol1 = 0.29;
```

Define asset 2. Price and volatility of WTI crude oil

```
  Price2 = 93.20;         % $/barrel
  Vol2 = 0.36;
```

Define the correlation between the underlying asset prices of asset 1 and asset 2.

```
Corr = 0.42;
```

Define the spread option.

```
OptSpec = 'call';
Strike = 20;
```

Define the `RateSpec`.

```
rates = 0.05;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
'EndDates', Maturity, 'Rates', rates, ...
'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9876
            Rates: 0.0500
         EndTimes: 0.2500
       StartTimes: 0
         EndDates: 734960
       StartDates: 734869
    ValuationDate: 734869
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec` for the two assets.

```
StockSpec1 = stockspec(Vol1, Price1)
```

```
StockSpec1 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2900
         AssetPrice: 119.7000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

```
StockSpec2 = stockspec(Vol2, Price2)
```

```
StockSpec2 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3600
         AssetPrice: 93.2000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Compute the spread option price based on the Alternate Direction Implicit (ADI) finite difference method.

```
[Price, PriceGrid, AssetPrice1, AssetPrice2, Times] = ...
 spreadbyfd(RateSpec, StockSpec1, StockSpec2, Settle, ...
 Maturity, OptSpec, Strike, Corr);
```

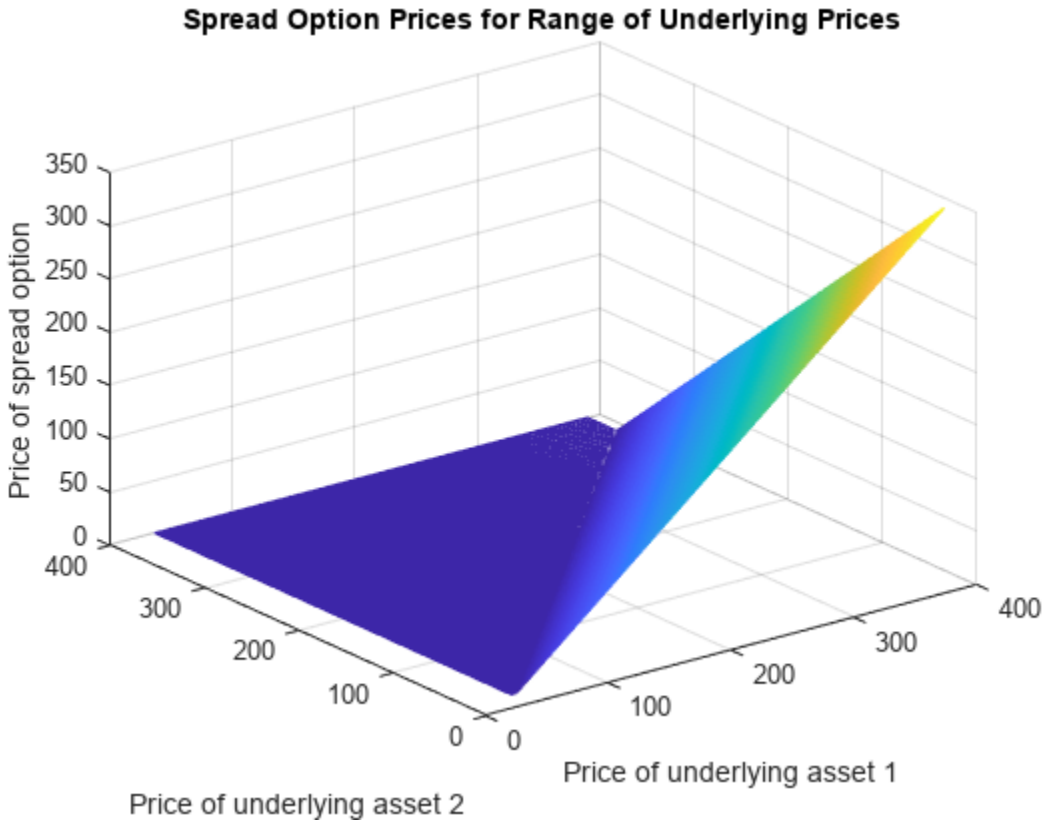
Display the price.

```
Price
```

```
Price = 11.1998
```

Plot the finite difference grid.

```
mesh(AssetPrice1, AssetPrice2, PriceGrid(:, :, 1)');
    title('Spread Option Prices for Range of Underlying Prices');
    xlabel('Price of underlying asset 1');
    ylabel('Price of underlying asset 2');
    zlabel('Price of spread option');
```



Input Arguments

RateSpec — Interest-rate term structure
structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec1` — Stock specification for underlying asset 1
structure

Stock specification for underlying asset 1. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`StockSpec2` — Stock specification for underlying asset 2
structure

Stock specification for underlying asset 2. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement dates for spread option
datetime array | string array | date character vector

Settlement dates for the spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for spread option
datetime array | string array | date character vector

Maturity date for spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
integer | vector of integers

Option strike price values, specified as an integer using a `NINST`-by-`1` vector of strike price values.

Data Types: `double`

`Corr` — Correlation between underlying asset prices

integer | vector of integers

Correlation between underlying asset prices, specified as an integer using a `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceGrid,AssetPrice1,AssetPrice2,Times] = spreadbyfd(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr,'AssetPriceMin','AssetPriceMax','PriceGridSize','TimeGridSize','AmericanOpt',0)`

`AssetPriceMin` — Minimum price for price grid boundary

if unspecified, `StockSpec` values are calculated based on asset distributions at maturity (default) | array

Minimum price for price grid boundary, specified as the comma-separated pair consisting of `'AssetPriceMin'` and a `1`-by-`2` array. The first entry in the array corresponds to the first asset defined by `StockSpec1` and the second entry corresponds to the second asset defined by `StockSpec2`.

For the finite difference method, the composition of the grid affects the quality of the output and the execution time. It is highly recommended to use the optional arguments `AssetPriceMin`, `AssetPriceMax`, `PriceGridSize`, and `TimeGridSize` to control the composition of the grid to ensure the quality of the output and a reasonable execution time.

Data Types: `single` | `double`

`AssetPriceMax` — Maximum price for price grid boundary

if unspecified, `StockSpec` values are calculated based on asset distributions at maturity (default) | array

Maximum price for price grid boundary, specified as the comma-separated pair consisting of `'AssetPriceMax'` and a `1`-by-`2` array. The first entry in the array corresponds to the first asset defined by `StockSpec1` and the second entry corresponds to the second asset defined by `StockSpec2`.

For the finite difference method, the composition of the grid affects the quality of the output and the execution time. It is highly recommended to use the optional arguments `AssetPriceMin`, `AssetPriceMax`, `PriceGridSize`, and `TimeGridSize` to control the composition of the grid to ensure the quality of the output and a reasonable execution time.

Data Types: `single` | `double`

`PriceGridSize` — Size for finite difference grid

`[300,300]` (default) | array

Size for finite difference grid, specified as the comma-separated pair consisting of `'PriceGridSize'` and a 1-by-2 array. The first entry corresponds to the first asset defined by `StockSpec1` and the second entry corresponds to the second asset defined by `StockSpec2`.

For the finite difference method, the composition of the grid affects the quality of the output and the execution time. It is highly recommended to use the optional arguments`AssetPriceMax`, `PriceGridSize`, and `TimeGridSize` to control the composition of the grid to ensure the quality of the output and a reasonable execution time.

Data Types: `single` | `double`

`TimeGridSize` — Size of the time grid for finite difference grid
`100` (default) | scalar | nonnegative integer

Size of the time grid for finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a nonnegative integer.

For the finite difference method, the composition of the grid affects the quality of the output and the execution time. It is highly recommended to use the optional arguments`AssetPriceMax`, `PriceGridSize`, and `TimeGridSize` to control the composition of the grid to ensure the quality of the output and a reasonable execution time.

Data Types: `single` | `double`

`AmericanOpt` — Option type
`0` European (default) | scalar | vector of positive integers`[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer scalar flags with values:

- `0` — European
- `1` — American

Data Types: `single` | `double`

Output Arguments

`Price` — Expected prices of spread option
vector

Expected prices of the spread option, returned as a `NINST`-by-`1` vector.

`PriceGrid` — Grid containing prices calculated by finite difference method
array

Grid containing prices calculated by finite difference method, returned as a 3-D grid with a size of `PriceGridSize(1)` * `PriceGridSize(2)` * `TimeGridSize`. The price for `t` = 0 is contained in `PriceGrid(:, :, 1)`.

`AssetPrice1` — Prices for first asset defined by `StockSpec1`
vector

Prices for first asset defined by `StockSpec1`, corresponding to the first dimension of `PriceGrid`, returned as a vector.

`AssetPrice2` — Prices for second asset defined by `StockSpec2`
vector

Prices for second asset defined by `StockSpec2`, corresponding to the second dimension of `PriceGrid`, returned as a vector.

`Times` — Times corresponding to third dimension of `PriceGrid`
vector

Times corresponding to third dimension of `PriceGrid`, returned as a vector.

More About

Spread Option

A spread option is an option written on the difference of two underlying assets.

For example, a European call on the difference of two assets *X1* and *X2* would have the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

where:

K is the strike price.

For more information, see “Spread Option” on page 3-30.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `spreadbyfd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Carmona, R., Durrleman, V. “Pricing and Hedging Spread Options.” *SIAM Review.* Vol. 45, No. 4, pp. 627–685, Society for Industrial and Applied Mathematics, 2003.

[2] Villeneuve, S., Zanette, A. “Parabolic ADI Methods for Pricing American Options on Two Stocks.” *Mathematics of Operations Research*. Vol. 27, No. 1, pp. 121–149, INFORMS, 2002.

[3] Ikonen, S., Toivanen, J. *Efficient Numerical Methods for Pricing American Options Under Stochastic Volatility*. Wiley InterScience, 2007.

See Also

`spreadsensbyfd` | `spreadbybjs` | `spreadbykirk` | `spreadbyls`

Topics

“Pricing European and American Spread Options” on page 3-97
“Pricing Asian Options” on page 3-110
“Spread Option” on page 3-30
“Supported Energy Derivative Functions” on page 3-34
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

Energy Trading & Risk Management with MATLAB (47 min 31 sec)

spreadbyls

Price European or American spread options using Monte Carlo simulations

Syntax

```
Price = spreadbyls(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,
Strike,Corr)
Price = spreadbyls( ___ ,Name,Value)

[Price,Paths,Times,Z] = spreadbyls(RateSpec,StockSpec1,StockSpec2,Settle,
Maturity,OptSpec,Strike,Corr)
[Price,Paths,Times,Z] = spreadbyls( ___ ,Name,Value)
```

Description

`Price = spreadbyls(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec, Strike,Corr)` returns the price of a European or American call or put spread option using Monte Carlo simulations.

For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium.

Note Alternatively, you can use the `Spread` object to price spread options. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`Price = spreadbyls( ___ ,Name,Value)` returns the price of a European or American call or put spread option using Monte Carlo simulations using optional name-value pair arguments.

`[Price,Paths,Times,Z] = spreadbyls(RateSpec,StockSpec1,StockSpec2,Settle, Maturity,OptSpec,Strike,Corr)` returns the `Price`, `Paths`, `Times`, and `Z` of a European or American call or put spread option using Monte Carlo simulations.

`[Price,Paths,Times,Z] = spreadbyls( ___ ,Name,Value)` returns the `Price`, `Paths`, `Times`, and `Z` of a European or American call or put spread option using Monte Carlo simulations using optional name-value pair arguments.

Examples

Compute the Price of a Spread Option Using Monte Carlo Simulation

Define the spread option dates.

```
Settle = datetime(2012,1,1);
Maturity = datetime(2012,4,1);
```

Define asset 1. Price and volatility of RBOB gasoline

```
Price1gallon = 2.85;          % $/gallon
Price1 = Price1gallon * 42;   % $/barrel
Vol1 = 0.29;
```

Define asset 2. Price and volatility of WTI crude oil

```
Price2 = 93.20;         % $/barrel
Vol2 = 0.36;
```

Define the correlation between the underlying asset prices of asset 1 and asset 2.

```
Corr = 0.42;
```

Define the spread option.

```
OptSpec = 'call';
Strike = 20;
```

Define the `RateSpec`.

```
rates = 0.05;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
'EndDates', Maturity, 'Rates', rates, ...
'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9876
            Rates: 0.0500
        EndTimes: 0.2500
       StartTimes: 0
         EndDates: 734960
       StartDates: 734869
    ValuationDate: 734869
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec` for the two assets.

```
StockSpec1 = stockspec(Vol1, Price1)
```

```
StockSpec1 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2900
         AssetPrice: 119.7000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

```
StockSpec2 = stockspec(Vol2, Price2)
```

```
StockSpec2 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3600
```

```
        AssetPrice: 93.2000
      DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Compute the spread option price using Monte Carlo simulation based on the Longstaff-Schwartz model.

```
Price = spreadbyls(RateSpec, StockSpec1, StockSpec2, Settle, ...
Maturity, OptSpec, Strike, Corr)
```

```
Price = 11.0799
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec1 — Stock specification for underlying asset 1

structure

Stock specification for underlying asset 1. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

StockSpec2 — Stock specification for underlying asset 2

structure

Stock specification for underlying asset 2. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement date for spread option

datetime scalar | string scalar | date character vector

Settlement date for the spread option specified, as a scalar datetime, string, or date character vector.

To support existing code, `spreadbyls` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date for spread option

datetime scalar | string scalar | date character vector

Maturity date for spread option, specified as a scalar datetime, string, or date character vector.

To support existing code, `spreadbyls` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'`

Definition of option as `'call'` or `'put'`, specified as a character vector.

Data Types: `char`

`Strike` — Option strike price value
nonnegative scalar integer

Option strike price value, specified, as a nonnegative scalar integer.

Data Types: `double`

`Corr` — Correlation between underlying asset prices
scalar integer

Correlation between underlying asset prices, specified as a scalar integer.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = spreadbyls(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr,'AmericanOpt',1)`

`AmericanOpt` — Option type
`0` European (default) | scalar with value `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and an integer scalar flag with value:

- `0` — European
- `1` — American

Note For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `single` | `double`

`NumTrials` — Scalar number of independent sample paths
`1000` (default) | nonnegative scalar integer

Scalar number of independent sample paths (simulation trials), specified as the comma-separated pair consisting of `'NumTrials'` and a nonnegative integer.

Data Types: `single` | `double`

`NumPeriods` — Scalar number of simulation periods per trial
`100` (default) | nonnegative scalar integer

Scalar number of simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` and a nonnegative integer. `NumPeriods` is considered only when pricing European basket options. For American spread options, `NumPeriod` is equal to the number of exercise days during the life of the option.

Data Types: `single` | `double`

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-2-by-`NumTrials` 3-D array. The `Z` value generates the Brownian motion vector (that is, Wiener processes) that drives the simulation.

Data Types: `single` | `double`

`Antithetic` — Indicator for antithetic sampling
`false` (default) | scalar logical flag with value of `true` or `false`

Indicator for antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a value of `true` or `false`.

Data Types: `logical`

Output Arguments

`Price` — Expected price of spread option
scalar

Expected price of the spread option, returned as a 1-by-1 scalar.

`Paths` — Simulated paths of correlated state variables
vector

Simulated paths of correlated state variables, returned as a `NumPeriods + 1`-by-2-by-`NumTrials` 3-D time series array. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths
vector

Observation times associated with simulated paths, returned as a `NumPeriods + 1`-by-1 column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

Z — Time series array of dependent random variates
vector

Time series array of dependent random variates, returned as a `NumPeriods`-by-2-by-`NumTrials` 3-D array when `Z` is specified as an input argument. If the `Z` input argument is not specified, then the `Z` output argument contains the random variates generated internally.

More About

Spread Option

A spread option is an option written on the difference of two underlying assets.

For example, a European call on the difference of two assets *X1* and *X2* would have the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

where:

K is the strike price.

For more information, see “Spread Option” on page 3-30.

Version History

Introduced in R2013b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `spreadbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Carmona, R., Durrleman, V. “Pricing and Hedging Spread Options.” *SIAM Review.* Vol. 45, No. 4, pp. 627–685, Society for Industrial and Applied Mathematics, 2003.

See Also

`spreadsensbyls` | `spreadbybjs` | `spreadbyfd` | `spreadbykirk` | `Spread`

Topics

“Pricing European and American Spread Options” on page 3-97
“Pricing Asian Options” on page 3-110
“Spread Option” on page 3-30
“Supported Energy Derivative Functions” on page 3-34
“Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

External Websites

Energy Trading & Risk Management with MATLAB (47 min 31 sec)

spreadsensbykirk

Calculate European spread option prices or sensitivities using Kirk pricing model

Syntax

```
PriceSens = spreadbykirk(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,
OptSpec,Strike,Corr)
PriceSens = spreadsensbykirk( ___ ,Name,Value)
```

Description

`PriceSens = spreadbykirk(RateSpec,StockSpec1,StockSpec2,Settle,Maturity, OptSpec,Strike,Corr)` returns the European spread option prices or sensitivities using the Kirk pricing model.

Note Alternatively, you can use the `Spread` object to calculate price or sensitivities for spread options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = spreadsensbykirk( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price and Sensitivities of a Spread Option Using the Kirk Model

Define the spread option dates.

```
Settle = datetime(2012,6,1);
Maturity = datetime(2012,9,1);
```

Define asset 1. Price and volatility of RBOB gasoline

```
Price1gallon = 2.85;          % $/gallon
Price1 = Price1gallon * 42;   % $/barrel
Vol1 = 0.29;
```

Define asset 2. Price and volatility of WTI crude oil

```
Price2 = 93.20;         % $/barrel
Vol2 = 0.36;
```

Define the correlation between the underlying asset prices of asset 1 and asset 2.

```
Corr = 0.42;
```

Define the spread option.

```
OptSpec = 'call';
Strike = 20;
```

Define the `RateSpec`.

```
rates = 0.05;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
'EndDates', Maturity, 'Rates', rates, ...
'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9876
            Rates: 0.0500
          EndTimes: 0.2500
        StartTimes: 0
          EndDates: 735113
        StartDates: 735021
    ValuationDate: 735021
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec` for the two assets.

```
StockSpec1 = stockspec(Vol1, Price1)
```

```
StockSpec1 = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.2900
          AssetPrice: 119.7000
        DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

```
StockSpec2 = stockspec(Vol2, Price2)
```

```
StockSpec2 = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.3600
          AssetPrice: 93.2000
        DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Compute the spread option price and sensitivities based on the Kirk model.

```
OutSpec = {'Price', 'Delta', 'Gamma'};
[Price, Delta, Gamma] = spreadsensbykirk(RateSpec, StockSpec1, StockSpec2, Settle, ...
Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec)
```

```
Price = 11.1904

Delta = 1×2

    0.6722   -0.6067
```

```
Gamma = 1×2

    0.0191    0.0217
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec1 — Stock specification for underlying asset 1

structure

Stock specification for underlying asset 1. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

StockSpec2 — Stock specification for underlying asset 2

structure

Stock specification for underlying asset 2. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement dates for spread option

datetime array | string array | date character vector

Settlement dates for the spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadsensbykirk` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date for spread option

datetime array | string array | date character vector

Maturity date for spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadsensbykirk` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
integer | vector of integers

Option strike price values, specified as an integer using as `NINST`-by-`1` vector of strike price values.

If `Strike` is equal to zero, this function computes the price and sensitivities of an exchange option.

Data Types: `double`

`Corr` — Correlation between underlying asset prices
integer | vector of integers

Correlation between underlying asset prices, specified as an integer using as `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = spreadsensbykirk(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr,OutSpec,{'All'})`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected price or sensitivities values of spread option
vector

Expected price or sensitivities values (defined by `OutSpec`) of the spread option, returned as a `NINST`-by-1 or `NINST`-by-2 vector.

More About

Spread Option

A spread option is an option written on the difference of two underlying assets.

For example, a European call on the difference of two assets *X1* and *X2* would have the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

where:

K is the strike price.

For more information, see “Spread Option” on page 3-30.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `spreadsensbykirk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Carmona, R., Durrleman, V. “Pricing and Hedging Spread Options.” *SIAM Review.* Vol. 45, No. 4, pp. 627–685, Society for Industrial and Applied Mathematics, 2003.

See Also

`spreadbykirk` | `spreadbybjs` | `spreadbyfd` | `spreadbyls` | `Spread`

Topics

“Pricing European and American Spread Options” on page 3-97
“Pricing Asian Options” on page 3-110

“Spread Option” on page 3-30
“Supported Energy Derivative Functions” on page 3-34

External Websites
Energy Trading & Risk Management with MATLAB (47 min 31 sec)

spreadsensbybjs

Calculate European spread option prices or sensitivities using Bjerksund-Stensland pricing model

Syntax

```
PriceSens = spreadbybjs(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,
OptSpec,Strike,Corr)
PriceSens = spreadsensbybjs( ___ ,Name,Value)
```

Description

`PriceSens = spreadbybjs(RateSpec,StockSpec1,StockSpec2,Settle,Maturity, OptSpec,Strike,Corr)` returns the European spread option prices or sensitivities using the Bjerksund-Stensland pricing model.

Note Alternatively, you can use the `Spread` object to calculate price or sensitivities for spread options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = spreadsensbybjs( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Price and Sensitivities of a Spread Option Using the Bjerksund-Stensland Model

Define the spread option dates.

```
Settle = datetime(2012,6,1);
Maturity = datetime(2012,9,1);
```

Define asset 1. Price and volatility of RBOB gasoline

```
   Price1gallon = 2.85;          % $/gallon
   Price1 = Price1gallon * 42;   % $/barrel
   Vol1 = 0.29;
```

Define asset 2. Price and volatility of WTI crude oil

```
  Price2 = 93.20;         % $/barrel
  Vol2 = 0.36;
```

Define the correlation between the underlying asset prices of asset 1 and asset 2.

```
Corr = 0.42;
```

Define the spread option.

```
OptSpec = 'call';
Strike = 20;
```

Define the RateSpec.

```
rates = 0.05;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
'EndDates', Maturity, 'Rates', rates, ...
'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9876
            Rates: 0.0500
         EndTimes: 0.2500
       StartTimes: 0
         EndDates: 735113
       StartDates: 735021
    ValuationDate: 735021
            Basis: 1
     EndMonthRule: 1
```

Define the StockSpec for the two assets.

```
StockSpec1 = stockspec(Vol1, Price1)
```

```
StockSpec1 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2900
         AssetPrice: 119.7000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

```
StockSpec2 = stockspec(Vol2, Price2)
```

```
StockSpec2 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3600
         AssetPrice: 93.2000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Compute the spread option price and sensitivities based on the Kirk model.

```
OutSpec = {'Price', 'Delta', 'Gamma'};
[Price, Delta, Gamma] = spreadsensbybjs(RateSpec, StockSpec1, StockSpec2, Settle, ...
Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec)
```

```
Price = 11.2000
```

```
Delta = 1×2
```

```
    0.6737   -0.6082

Gamma = 1×2

    0.0190    0.0216
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec1 — Stock specification for underlying asset 1

structure

Stock specification for underlying asset 1. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

StockSpec2 — Stock specification for underlying asset 2

structure

Stock specification for underlying asset 2. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement dates for spread option

datetime array | string array | date character vector

Settlement dates for the spread option, specified a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadsensbybjs` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date for spread option

datetime array | string array | date character vector

Maturity date for spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadsensbybjs` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
integer | vector of integers

Option strike price values, specified as an integer using a `NINST`-by-`1` vector of strike price values.

If `Strike` is equal to zero the function computes the price and sensitivities of an exchange option.

Data Types: `double`

`Corr` — Correlation between underlying asset prices
integer | vector of integers

Correlation between underlying asset prices, specified as an integer using a `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = spreadsensbykirk(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr,OutSpec,{'All'})`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities values of spread option

vector

Expected prices or sensitivities values (defined by `OutSpec`) of the spread option, returned as a `NINST`-by-`1` or `NINST`-by-`2` vector.

More About

Spread Option

A spread option is an option written on the difference of two underlying assets.

For example, a European call on the difference of two assets *X1* and *X2* would have the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

where:

K is the strike price.

For more information, see “Spread Option” on page 3-30.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `spreadsensbybjs` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Carmona, R., Durrleman, V. “Pricing and Hedging Spread Options,” *SIAM Review.* Vol. 45, No. 4, pp. 627–685, Society for Industrial and Applied Mathematics, 2003.

[2] Bjerksund, Petter, Stensland, Gunnar. *“Closed form spread option valuation.”* Department of Finance, NHH, 2006.

See Also

`spreadbykirk` | `spreadbybjs` | `spreadbyfd` | `spreadbyls` | `Spread`

Topics

“Pricing European and American Spread Options” on page 3-97
“Pricing Asian Options” on page 3-110
“Spread Option” on page 3-30
“Supported Energy Derivative Functions” on page 3-34

External Websites

Energy Trading & Risk Management with MATLAB (47 min 31 sec)

spreadsensbyls

Calculate price and sensitivities for European or American spread options using Monte Carlo simulations

Syntax

```
PriceSens = spreadsensbyls(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,
OptSpec,Strike,Corr)
PriceSens = spreadsensbyls( ___ ,Name,Value)

[PriceSens,Paths,Times,Z] = spreadsensbyls(RateSpec,StockSpec1,StockSpec2,
Settle,Maturity,OptSpec,Strike,Corr)
[PriceSens,Paths,Times,Z] = spreadsensbyls( ___ ,Name,Value)
```

Description

`PriceSens = spreadsensbyls(RateSpec,StockSpec1,StockSpec2,Settle,Maturity, OptSpec,Strike,Corr)` returns the price of a European or American call or put spread option using Monte Carlo simulations.

For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium.

Note Alternatively, you can use the `Spread` object to calculate price or sensitivities for spread options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = spreadsensbyls( ___ ,Name,Value)` adds optional name-value pair arguments.

`[PriceSens,Paths,Times,Z] = spreadsensbyls(RateSpec,StockSpec1,StockSpec2, Settle,Maturity,OptSpec,Strike,Corr)` returns the `PriceSens`, `Paths`, `Times`, and `Z` of a European or American call or put spread option using Monte Carlo simulations.

`[PriceSens,Paths,Times,Z] = spreadsensbyls( ___ ,Name,Value)` returns the `PriceSens`, `Paths`, `Times`, and `Z` and adds optional name-value pair arguments.

Examples

Compute the Price and Sensitivities of a Spread Option Using Monte Carlo Simulation

Define the spread option dates.

```
Settle = datetime(2012,6,1);
Maturity = datetime(2012,9,1);
```

Define asset 1. Price and volatility of RBOB gasoline

```
  Price1gallon = 2.85;          % $/gallon
  Price1 = Price1gallon * 42;   % $/barrel
  Vol1 = 0.29;
```

Define asset 2. Price and volatility of WTI crude oil

```
  Price2 = 93.20;         % $/barrel
  Vol2 = 0.36;
```

Define the correlation between the underlying asset prices of asset 1 and asset 2.

```
Corr = 0.42;
```

Define the spread option.

```
OptSpec = 'call';
Strike = 20;
```

Define the `RateSpec`.

```
rates = 0.05;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
'EndDates', Maturity, 'Rates', rates, ...
'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9876
            Rates: 0.0500
        EndTimes: 0.2500
       StartTimes: 0
         EndDates: 735113
       StartDates: 735021
    ValuationDate: 735021
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec` for the two assets.

```
StockSpec1 = stockspec(Vol1, Price1)
```

```
StockSpec1 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2900
         AssetPrice: 119.7000
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

```
StockSpec2 = stockspec(Vol2, Price2)
```

```
StockSpec2 = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.3600
```

```
        AssetPrice: 93.2000
      DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Compute the spread option price and sensitivities using Monte Carlo simulation based on the Longstaff-Schwartz model.

```
OutSpec = {'Price', 'Delta', 'Gamma'};
[Price, Delta, Gamma] = spreadsensbyls(RateSpec, StockSpec1, StockSpec2, ...
Settle, Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec)
```

```
Price = 11.0799

Delta = 1×2

    0.6626   -0.5972


Gamma = 1×2

    0.0209    0.0240
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec1 — Stock specification for underlying asset 1

structure

Stock specification for underlying asset 1. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

StockSpec2 — Stock specification for underlying asset 2

structure

Stock specification for underlying asset 2. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement date for spread option
datetime scalar | string scalar | date character vector

Settlement date for the spread option, specified as a scalar datetime, string, or date character vector.

To support existing code, `spreadsensbyls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for spread option
datetime scalar | string scalar | date character vector

Maturity date for spread option, specified as a scalar datetime, string, or date character vector.

To support existing code, `spreadsensbyls` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'`

Definition of option as `'call'` or `'put'`, specified as a character vector.

Data Types: `char`

`Strike` — Option strike price value
nonnegative scalar integer

Option strike price value, specified as a scalar integer.

Data Types: `single` | `double`

`Corr` — Correlation between underlying asset prices
scalar integer

Correlation between underlying asset prices, specified as a scalar integer.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceSens = spreadbyls(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr,'AmericanOpt',1)`

`AmericanOpt` — Option type
`0` European (default) | scalar with values `[0,1]`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and a scalar integer flag with values:

- `0` — European

- 1 — American

Note For American options, the Longstaff-Schwartz least squares method is used to calculate the early exercise premium. For more information on the least squares method, see https://people.math.ethz.ch/%7Ehjfurrer/teaching/LongstaffSchwartzAmericanOptionsLeastSquareMonteCarlo.pdf.

Data Types: `single` | `double`

`NumTrials` — Number of independent sample paths
`1000` (default) | nonnegative scalar integer

Number of independent sample paths (simulation trials), specified as the comma-separated pair consisting of `'NumTrials'` and a nonnegative scalar integer.

Data Types: `single` | `double`

`NumPeriods` — Number of simulation periods per trial
`100` (default) | nonnegative scalar integer

Number of simulation periods per trial, specified as the comma-separated pair consisting of `'NumPeriods'` and a nonnegative scalar integer. `NumPeriods` is considered only when pricing European basket options. For American spread options, `NumPeriods` is equal to the number of exercise days during the life of the option.

Data Types: `single` | `double`

`Z` — Time series array of dependent random variates
vector

Time series array of dependent random variates, specified as the comma-separated pair consisting of `'Z'` and a `NumPeriods`-by-2-by-`NumTrials` 3-D array. The `Z` value generates the Brownian motion vector (that is, Wiener processes) that drives the simulation.

Data Types: `single` | `double`

`Antithetic` — Indicator for antithetic sampling
`false` (default) | logical flag with value of `true` or `false`

Indicator for antithetic sampling, specified as the comma-separated pair consisting of `'Antithetic'` and a value of `true` or `false`.

Data Types: `logical`

`OutSpec` — Define outputs
`{'Price'}` (default) | character vector with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'`and a `NOUT`- by-`1` or `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

`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` to include each sensitivity:

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected price or sensitivities of spread option

scalar

Expected price or sensitivities of the spread option, returned as a 1-by-1 array as defined by `OutSpec`.

`Paths` — Simulated paths of correlated state variables

vector

Simulated paths of correlated state variables, returned as a `NumPeriods + 1`-by-2-by-`NumTrials` 3-D time series array. Each row of `Paths` is the transpose of the state vector $X(t)$ at time t for a given trial.

`Times` — Observation times associated with simulated paths

vector

Observation times associated with simulated paths, returned as a `NumPeriods + 1`-by-1 column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Time series array of dependent random variates

vector

Time series array of dependent random variates, returned as a `NumPeriods`-by-2-by-`NumTrials` 3-D array when `Z` is specified as an input argument. If the `Z` input argument is not specified, then the `Z` output argument contains the random variates generated internally.

More About

Spread Option

A spread option is an option written on the difference of two underlying assets.

For example, a European call on the difference of two assets *X1* and *X2* would have the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

where:

K is the strike price.

For more information, see “Spread Option” on page 3-30.

Version History

Introduced in R2013b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `spreadsensbyls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Carmona, R., Durrleman, V. “Pricing and Hedging Spread Options.” *SIAM Review.* Vol. 45, No. 4, pp. 627–685, Society for Industrial and Applied Mathematics, 2003.

See Also

`spreadbyls` | `spreadbybjs` | `spreadbyfd` | `spreadbykirk` | `Spread`

Topics
“Pricing European and American Spread Options” on page 3-97
“Pricing Asian Options” on page 3-110
“Spread Option” on page 3-30
“Supported Energy Derivative Functions” on page 3-34

External Websites
Energy Trading & Risk Management with MATLAB (47 min 31 sec)

spreadsensbyfd

Calculate price and sensitivities of European or American spread options using finite difference method

Syntax

```
PriceSens = spreadsensbyfd(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,
OptSpec,Strike,Corr)
PriceSens = spreadsensbyfd( ___ ,Name,Value)

[PriceSens,PriceGrid,AssetPrice1,AssetPrice2,Times] = spreadsensbyfd(
RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr)
[PriceSens,PriceGrid,AssetPrice1,AssetPrice2,Times] = spreadsensbyfd( ___ ,
Name,Value)
```

Description

`PriceSens = spreadsensbyfd(RateSpec,StockSpec1,StockSpec2,Settle,Maturity, OptSpec,Strike,Corr)` returns the price and sensitivities of European or American call or put spread options using the Alternate Direction Implicit (ADI) finite difference method. The spread is between the asset defined in `StockSpec1` minus the asset defined in `StockSpec2`.

`PriceSens = spreadsensbyfd( ___ ,Name,Value)` adds optional name-value pair arguments.

`[PriceSens,PriceGrid,AssetPrice1,AssetPrice2,Times] = spreadsensbyfd( RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike,Corr)` returns the `PriceSens`, `PriceGrid`, `AssetPrice1`, `AssetPrice2`, and `Times` for European or American call or put spread options using the Alternate Direction Implicit (ADI) finite difference method. The spread is between the asset defined in `StockSpec1` minus the asset defined in `StockSpec2`.

`[PriceSens,PriceGrid,AssetPrice1,AssetPrice2,Times] = spreadsensbyfd( ___ , Name,Value)` returns the `PriceSens`, `PriceGrid`, `AssetPrice1`, `AssetPrice2`, and `Times` and adds optional name-value pair arguments.

Examples

Compute the Price of a Spread Option Using the Alternate Direction Implicit (ADI) Finite Difference Method

Define the spread option dates.

```
Settle = datetime(2012,6,1);
Maturity = datetime(2012,9,1);
```

Define asset 1. Price and volatility of RBOB gasoline

```
  Price1gallon = 2.85;          % $/gallon
  Price1 = Price1gallon * 42;   % $/barrel
  Vol1 = 0.29;
```

Define asset 2. Price and volatility of WTI crude oil

```
Price2 = 93.20;          % $/barrel
Vol2 = 0.36;
```

Define the correlation between the underlying asset prices of asset 1 and asset 2.

```
Corr = 0.42;
```

Define the spread option.

```
OptSpec = 'call';
Strike = 20;
```

Define the `RateSpec`.

```
rates = 0.05;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, ...
'EndDates', Maturity, 'Rates', rates, ...
'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9876
            Rates: 0.0500
          EndTimes: 0.2500
        StartTimes: 0
          EndDates: 735113
        StartDates: 735021
    ValuationDate: 735021
            Basis: 1
     EndMonthRule: 1
```

Define the `StockSpec` for the two assets.

```
StockSpec1 = stockspec(Vol1, Price1)
```

```
StockSpec1 = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.2900
         AssetPrice: 119.7000
        DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

```
StockSpec2 = stockspec(Vol2, Price2)
```

```
StockSpec2 = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.3600
         AssetPrice: 93.2000
        DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Compute the spread option price and sensitivities based on the Alternate Direction Implicit (ADI) finite difference method.

```
OutSpec = {'Price', 'Delta', 'Gamma'};
[Price, Delta, Gamma, PriceGrid, AssetPrice1, AssetPrice2, Times] = ...
spreadsensbyfd(RateSpec, StockSpec1, StockSpec2, Settle, ...
Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec);
```

Display the price and sensitivities.

```
Price
```

```
Price = 11.1998
```

```
Delta
```

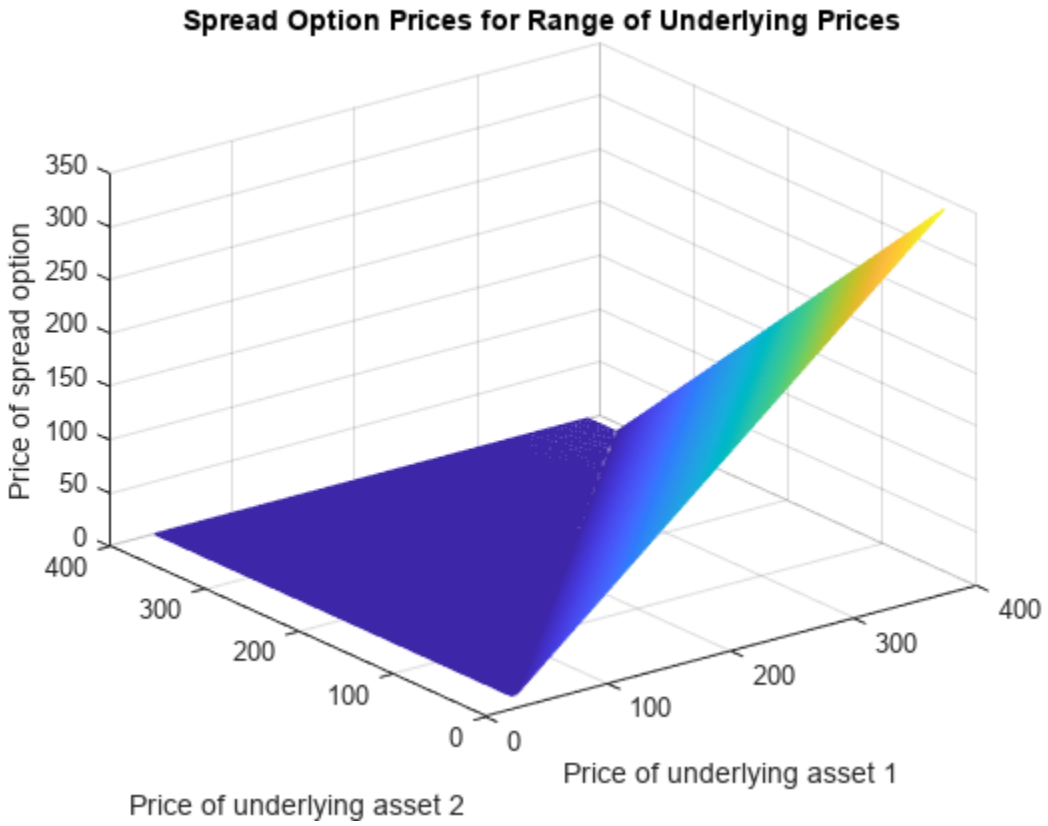
```
Delta = 1×2

    0.6736   -0.6082
```

```
Gamma
```

```
Gamma = 1×2

    0.0190    0.0214
```

Plot the finite difference grid.

```
mesh(AssetPrice1, AssetPrice2, PriceGrid(:, :, 1)');
    title('Spread Option Prices for Range of Underlying Prices');
    xlabel('Price of underlying asset 1');
    ylabel('Price of underlying asset 2');
    zlabel('Price of spread option');
```



Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec1 — Stock specification for underlying asset 1

structure

Stock specification for underlying asset 1. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

StockSpec2 — Stock specification for underlying asset 2

structure

Stock specification for underlying asset 2. For information on the stock specification, see `stockspec`.

`stockspec` can handle other types of underlying assets. For example, for physical commodities the price is represented by `StockSpec.Asset`, the volatility is represented by `StockSpec.Sigma`, and the convenience yield is represented by `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement dates for spread option
datetime array | string array | date character vector

Settlement dates for the spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadsensbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for spread option
datetime array | string array | date character vector

Maturity date for spread option, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `spreadsensbyfd` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vectors

Definition of option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors.

Data Types: `char` | `cell`

`Strike` — Option strike price values
integer | vector of integers

Option strike price values, specified as an integer using a `NINST`-by-`1` vector of strike price values.

Data Types: `double`

`Corr` — Correlation between underlying asset prices
integer | vector of integers

Correlation between underlying asset prices, specified as an integer using a`NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PriceSens,PriceGrid,AssetPrice1,AssetPrice2,Times] = spreadsensbyfd(RateSpec,StockSpec1,StockSpec2,Settle,Maturity,OptSpec,Strike, Corr, 'AssetPriceMin','AssetPriceMax','PriceGridSize','TimeGridSize','AmericanOpt', 0,'OutSpec',{'All'})`

OutSpec — Define outputs
{'Price'} (default) | character vector with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All' | cell array of character vectors with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and a NOUT- by-1 or 1-by-NOUT cell array of character vectors with possible values of 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'.

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 to include each sensitivity:

Example: OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}

Data Types: char | cell

AssetPriceMin — Minimum price for price grid boundary
if unspecified, StockSpec values are calculated based on asset distributions at maturity (default) | array

Minimum price for price grid boundary, specified as the comma-separated pair consisting of 'AssetPriceMin' and a 1-by-2 array. The first entry in the array corresponds to the first asset defined by StockSpec1 and the second entry corresponds to the second asset defined by StockSpec2.

For the finite difference method, the composition of the grid affects the quality of the output and the execution time. It is highly recommended to use the optional arguments AssetPriceMin, AssetPriceMax, PriceGridSize, and TimeGridSize to control the composition of the grid to ensure the quality of the output and a reasonable execution time.

Data Types: single | double

AssetPriceMax — Maximum price for price grid boundary
if unspecified, StockSpec values are calculated based on asset distributions at maturity (default) | array

Maximum price for price grid boundary, specified as the comma-separated pair consisting of 'AssetPriceMax' and a 1-by-2 array. The first entry in the array corresponds to the first asset defined by StockSpec1 and the second entry corresponds to the second asset defined by StockSpec2.

For the finite difference method, the composition of the grid affects the quality of the output and the execution time. It is highly recommended to use the optional arguments AssetPriceMin, AssetPriceMax, PriceGridSize, and TimeGridSize to control the composition of the grid to ensure the quality of the output and a reasonable execution time.

Data Types: single | double

PriceGridSize — Size for finite difference grid
[300,300] (default) | array

Size for finite difference grid, specified as the comma-separated pair consisting of 'PriceGridSize' and a 1-by-2 array. The first entry corresponds to the first asset defined by StockSpec1 and the second entry corresponds to the second asset defined by StockSpec2.

For the finite difference method, the composition of the grid affects the quality of the output and the execution time. It is highly recommended to use the optional argumentsAssetPriceMax,

PriceGridSize, and TimeGridSize to control the composition of the grid to ensure the quality of the output and a reasonable execution time.

Data Types: single | double

TimeGridSize — Size of time grid for finite difference grid
100 (default) | scalar | nonnegative integer

Size of time grid for finite difference grid, specified as the comma-separated pair consisting of 'TimeGridSize' and a nonnegative integer.

For the finite difference method, the composition of the grid affects the quality of the output and the execution time. It is highly recommended to use the optional argumentsAssetPriceMax, PriceGridSize, and TimeGridSize to control the composition of the grid to ensure the quality of the output and a reasonable execution time.

Data Types: single | double

AmericanOpt — Option type
0 European (default) | scalar | vector of positive integers[0,1]

Option type, specified as the comma-separated pair consisting of 'AmericanOpt' and NINST-by-1 positive integer scalar flags with values:

- 0 — European
- 1 — American

Data Types: single | double

Output Arguments

PriceSens — Expected prices or sensitivities of spread option
scalar

Expected price or sensitivities of the spread option, returned as a 1-by-1 array as defined by OutSpec.

PriceGrid — Grid containing prices calculated by finite difference method
array

Grid containing prices calculated by finite difference method, returned as a 3-D grid with a size of PriceGridSize(1) * PriceGridSize(2) * TimeGridSize. The price for t = 0 is contained in PriceGrid(:, :, 1).

AssetPrice1 — Prices for first asset defined by StockSpec1
vector

Prices for first asset defined by StockSpec1, corresponding to the first dimension of PriceGrid, returned as a vector.

AssetPrice2 — Prices for second asset defined by StockSpec2
vector

Prices for second asset defined by StockSpec2, corresponding to the second dimension of PriceGrid, returned as a vector.

`Times` — Times corresponding to third dimension of `PriceGrid`
vector

Times corresponding to third dimension of `PriceGrid`, returned as a vector.

More About

Spread Option

A spread option is an option written on the difference of two underlying assets.

For example, a European call on the difference of two assets *X1* and *X2* would have the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

where:

K is the strike price.

For more information, see “Spread Option” on page 3-30.

Version History

Introduced in R2013b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `spreadsensbyfd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Carmona, R., Durrleman, V. “Pricing and Hedging Spread Options.” *SIAM Review.* Vol. 45, No. 4, pp. 627–685, Society for Industrial and Applied Mathematics, 2003.

[2] Villeneuve, S., Zanette, A. “Parabolic ADI Methods for Pricing American Options on Two Stocks.” *Mathematics of Operations Research.* Vol. 27, No. 1, pp. 121–149, INFORMS, 2002.

[3] Ikonen, S., Toivanen, J. *Efficient Numerical Methods for Pricing American Options Under Stochastic Volatility.* Wiley InterScience, 2007.

See Also

`spreadbyfd` | `spreadbybjs` | `spreadbykirk` | `spreadbyls`

Topics

“Pricing European and American Spread Options” on page 3-97
“Pricing Asian Options” on page 3-110
“Spread Option” on page 3-30
“Supported Energy Derivative Functions” on page 3-34

External Websites

Energy Trading & Risk Management with MATLAB (47 min 31 sec)

stockoptspec

Specify European stock option structure

Syntax

```
StockOptSpec = stockoptspec(OptPrice,Strike,Settle,Maturity,OptSpec,
InterpMethod)
StockOptSpec = stockoptspec( ___ ,InterpMethod)
```

Description

`StockOptSpec = stockoptspec(OptPrice,Strike,Settle,Maturity,OptSpec, InterpMethod)` creates a structure encapsulating the properties of a stock option structure.

`StockOptSpec = stockoptspec( ___ ,InterpMethod)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Specify a European Stock Option Structure

This example shows how to specify a European stock option structure using the following data quoted from liquid options in the market with varying strikes and maturity.

```
Settle =   datetime(2006,1,1);

Maturity = [datetime(2006,7,1) ; datetime(2006,7,1) ; datetime(2006,7,1) ; datetime(2006,7,1) ; d
    datetime(2007,7,1) ; datetime(2007,7,1) ; datetime(2007,7,1) ; datetime(2008,1,1) ; datetime

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

Input Arguments

`OptPrice` — European option prices

vector

European option prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Strike` — Strike prices

vector

Strike prices, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `stockoptspec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity dates
datetime array | string array | date character vector

Maturity dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `stockoptspec` also accepts serial date numbers as inputs, but they are not recommended.

`OptSpec` — Option type
cell array of character vectors with a value of `'call'` or `'put'`

Option type, specified as an `NINST`-by-`1` cell array of character vectors with a value of `'call'` or `'put'`.

Data Types: `cell`

`InterpMethod` — Interpolation method for option prices
`'price'` (default) | scalar character vector with value of `'price'` or `'vol'`

(Optional) Interpolation method for option prices, specified as a scalar character vector with one of the following values:

- `'price'` indicates that prices are used for interpolation purposes.
- `'vol'` indicates that implied volatilities are used for interpolation purposes. The interpolated values are then used to calculate the implicit interpolated prices.

.

Data Types: `char`

Output Arguments

`StockOptSpec` — Structure encapsulating the properties of a stock options structure
structure

Structure encapsulating the properties of a stock options structure, returned as a structure.

Version History

Introduced in R2007a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `stockoptspec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`ittprice` | `itttree` | `stockspec`

Topics

“Building Implied Trinomial Trees” on page 3-6
“Examining Equity Trees” on page 3-14
“Understanding Equity Trees” on page 3-2
“Supported Equity Derivative Functions” on page 3-19

stockspec

Create stock structure

Syntax

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

Description

`StockSpec = stockspec(Sigma,AssetPrice)` creates a MATLAB structure containing the properties of a stock.

Note `StockSpec` handles other types of underliers when pricing instruments other than equities.

`StockSpec = stockspec( ___ ,DividendType,DividendAmounts,ExDividendDates)` adds optional arguments for `DividendType`, `DividendAmounts`, and `ExDividendDates`.

Examples

Create a StockSpec for Stocks With Cash Dividends

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 = [datetime(2008,1,3) , datetime(2008,4,1) , datetime(2008,7,5) , datetime(2008,1
```

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

```
StockSpec = struct with fields:
             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 = 1x1 cell array
    {'cash'}
```

The `StockSpec` structure encapsulates the information of the stock and its four cash dividends.

Create a StockSpec for Stocks With Cash and Continuous Dividends

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 = datetime(2008,3,1);
DividendDate2 = datetime(2008,6,1);

StockSpec = stockspec(Sigma, AssetPrice, DividendType, DividendAmount,...
{ DividendDate1, DividendDate2 ; NaN NaN})
```

```
StockSpec = struct with fields:
             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 = 2x1 cell
    {'cash'      }
    {'continuous'}
```

The `StockSpec` structure encapsulates the information of the two stocks and their dividends.

Input Arguments

`Sigma` — Annual price volatility of underlying security

decimal

Annual price volatility of underlying security, specified as a NINST-by-1 decimal.

Data Types: double

AssetPrice — Underlying asset price values at time 0
vector

Underlying asset price values at time 0, specified as a NINST-by-1 vector.

Data Types: double

DividendType — Stock dividend type
cell array of date character vectors

(Optional) Stock dividend type, specified as a NINST-by-1 cell array of character vectors.

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 types of dividends, shorter rows should be padded with the value NaN.

Data Types: char | cell

DividendAmounts — Dividend amounts
matrix | vector

(Optional) Dividend amounts, specified as a NINST-by-NDIV matrix of cash dividends or NINST-by-1 vector representing a constant or continuous annualized dividend yield.

Data Types: double

ExDividendDates — Ex-dividend dates
datetime array | string array | date character vector

(Optional) Ex-dividend dates, specified as a NINST-by-NDIV matrix of ex-dividend dates for a cash DividendType or a NINST-by-1 vector of ex-dividend dates for constant DividendType. For dates, use a datetime array, string array, or date character vectors. For a continuous DividendType, this argument should be ignored.

To support existing code, stockspec also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

StockSpec — Properties of stock structure
structure

Properties of stock structure, returned as a structure encapsulating the properties of a stock.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `stockspec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`crrprice` | `crrtree` | `intenvset` | `optstockbybjs` | `optstockbyblk` | `optstockbyls` | `optstockbybls` | `spreadbykirk` | `spreadbybjs` | `spreadbyfd` | `spreadbyls` | `optstockbyrgw`

Topics

“Portfolio Creation Using Functions” on page 1-6
“Supported Equity Derivative Functions” on page 3-19

sttprice

Price instruments using standard trinomial tree

Syntax

```
[Price,PriceTree] = sttprice(STTTree,InstSet)
[Price,PriceTree] = sttprice( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = sttprice(STTTree,InstSet)` prices instruments using a standard trinomial (STT) tree.

`[Price,PriceTree] = sttprice( ___ ,Name,Value)` prices instruments using a standard trinomial (STT) tree with an optional name-value pair argument for `Options`.

Examples

Price a sttree Instrument Set

Load the data into the MATLAB® workspace.

```
load deriv.mat
```

`STTTree` and `STTInstSet` are the input arguments required to call the function `sttprice`. Use the command `instdisp` to examine the set of instruments contained in the variable `STTInstSet`.

```
instdisp(STTInstSet)
```

```
Index Type     OptSpec Strike Settle         ExerciseDates  AmericanOpt Name  Quantity
1     OptStock call    100    01-Jan-2009    01-Jan-2011    1           Call1 10
2     OptStock put      80    01-Jan-2009    01-Jan-2012    0           Put1   5

Index Type    OptSpec Strike Settle         ExerciseDates  AmericanOpt BarrierSpec Barrier Rebate
3     Barrier call    105    01-Jan-2009    01-Jan-2012    1           ui          115     0

Index Type     UOptSpec UStrike USettle        UExerciseDates UAmericanOpt COptSpec CStrike CSett
4     Compound call     95      01-Jan-2009    01-Jan-2012    1            put      5       01-Ja

Index Type     OptSpec Strike Settle         ExerciseDates  AmericanOpt Name      Quantity
5     Lookback call    90     01-Jan-2009    01-Jan-2012    0           Lookback1 7
6     Lookback call    95     01-Jan-2009    01-Jan-2013    0           Lookback2 9

Index Type  OptSpec Strike Settle         ExerciseDates  AmericanOpt AvgType    AvgPrice AvgDate
7     Asian call    100    01-Jan-2009    01-Jan-2012    0           arithmetic NaN      NaN
8     Asian call    100    01-Jan-2009    01-Jan-2013    0           arithmetic NaN      NaN
```

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`)

Use `sttprice` to calculate the price of each instrument in the instrument set.

```
Price = sttprice(STTTree, STTInstSet)
```

```
Price = 8×1

    4.5025
    3.0603
    3.7977
    1.7090
   11.7296
   12.9120
    1.6905
    2.6203
```

Input Arguments

`STTTree` — Stock tree structure for standard trinomial tree

structure

Stock tree structure for a standard trinomial tree, specified by using `stttree`.

Data Types: `struct`

`InstSet` — Variable containing a collection instruments

structure

Variable containing a collection of `NINST` instruments, specified as a structure. Instruments are broken down by type and each type can have different data fields.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = sttprice(STTTree,InstSet,'Options',deriv)`

`Options` — Derivatives pricing options

structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices for each instrument at time `0`
matrix

Expected prices for each instrument at time `0`, returned as a `NINST`-by-`1` vector. The prices are computed by backward dynamic programming on the standard trinomial (STT) stock tree. If an instrument cannot be priced, a `NaN` is returned in that entry.

`PriceTree` — Structure with vector of instrument prices at each node
tree structure

Structure with a vector of instrument prices at each node, returned as a tree structure.

`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.

Version History

Introduced in R2015b

See Also

`stttree` | `sttsens` | `stttimespec`

Topics

“Convertible Bond” on page 2-4

sttsens

Instrument sensitivities and prices using standard trinomial tree

Syntax

```
[Delta,Gamma,Vega,Price] = sttsens(STTTree,InstSet)
[Delta,Gamma,Vega,Price] = sttsens(___,Name,Value)
```

Description

`[Delta,Gamma,Vega,Price] = sttsens(STTTree,InstSet)` to generate instrument sensitivities and prices using a standard trinomial (STT) tree.

`[Delta,Gamma,Vega,Price] = sttsens(___,Name,Value)` to generate instrument sensitivities and prices using a standard trinomial (STT) tree with an optional name-value pair argument for `Options`.

Examples

Determine the Price and Sensitivities for a `stttree` Instrument Set

Load the data into the MATLAB® workspace.

```
load deriv.mat
```

`STTTree` and `STTInstSet` are the input arguments required to call the function `sttprice`. Use the command `instdisp` to examine the set of instruments contained in the variable `STTInstSet`.

```
instdisp(STTInstSet)
```

```
Index Type     OptSpec Strike Settle         ExerciseDates  AmericanOpt Name  Quantity
1     OptStock call    100    01-Jan-2009    01-Jan-2011    1           Call1 10
2     OptStock put      80    01-Jan-2009    01-Jan-2012    0           Put1   5

Index Type    OptSpec Strike Settle         ExerciseDates  AmericanOpt BarrierSpec Barrier Rebate
3     Barrier call    105    01-Jan-2009    01-Jan-2012    1           ui          115     0

Index Type     UOptSpec UStrike USettle        UExerciseDates UAmericanOpt COptSpec CStrike CSett
4     Compound call     95      01-Jan-2009    01-Jan-2012    1            put      5       01-Ja

Index Type     OptSpec Strike Settle         ExerciseDates  AmericanOpt Name      Quantity
5     Lookback call    90     01-Jan-2009    01-Jan-2012    0           Lookback1 7
6     Lookback call    95     01-Jan-2009    01-Jan-2013    0           Lookback2 9

Index Type  OptSpec Strike Settle         ExerciseDates  AmericanOpt AvgType    AvgPrice AvgDate
7     Asian call    100    01-Jan-2009    01-Jan-2012    0           arithmetic NaN      NaN
8     Asian call    100    01-Jan-2009    01-Jan-2013    0           arithmetic NaN      NaN
```

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`)

Use `sttsens` to calculate the price and sensitivities for each instrument in the instrument set.

```
[Delta,Gamma,Vega,Price] = sttsens(STTTree, STTInstSet)
```

```
Delta = 8×1

    0.5267
   -0.0943
    0.4726
   -0.0624
    0.2313
    0.3266
    0.5706
    0.6646


Gamma = 8×1
10^5 ×

    0.0000
    0.0000
    0.0000
    0.0000
   -1.8650
   -1.9119
    1.8650
    1.9119


Vega = 8×1

   52.8980
   42.4369
   25.9792
   -9.5266
   70.3758
   92.9226
   25.8122
   37.8757


Price = 8×1

    4.5025
    3.0603
    3.7977
    1.7090
   11.7296
   12.9120
    1.6905
```

```
    2.6203
```

Determine Price and Sensitivities for Convertible Bond Instruments Using a stttree

Create a RateSpec.

```
StartDates = datetime(2015,1,1);
EndDates = datetime(2020,1,1);
Rates = 0.025;
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.8825
            Rates: 0.0250
         EndTimes: 5
       StartTimes: 0
         EndDates: 737791
       StartDates: 735965
    ValuationDate: 735965
            Basis: 1
     EndMonthRule: 1
```

Create a StockSpec.

```
AssetPrice = 80;
Sigma = 0.12;
StockSpec = stockspec(Sigma,AssetPrice)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.1200
         AssetPrice: 80
       DividendType: []
    DividendAmounts: 0
    ExDividendDates: []
```

Create a STTTree.

```
TimeSpec = stttimespec(StartDates, EndDates, 20);
STTTree = stttree(StockSpec, RateSpec, TimeSpec)
```

```
STTTree = struct with fields:
       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
         dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]
```

```
        STree: {1x21 cell}
        Probs: {1x20 cell}
```

Define the convertible bond. The convertible bond can be called starting on Jan 1, 2016 with a strike price of 95.

```
CouponRate = 0.03;
Settle = datetime(2015,1,1);
Maturity = datetime(2018,4,1);
Period = 1;
CallStrike = 95;
CallExDates = [datetime(2016,1,1) datetime(2018,4,1)];
ConvRatio = 1;
Spread = 0.025;
```

Price the convertible bond using the standard trinomial tree model.

```
[Price,PriceTree,EqtTre,DbtTree] = cbondbystt(STTTree,CouponRate,Settle,Maturity,ConvRatio,...
'Period',Period,'Spread',Spread,'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall'
```

```
Price = 90.2511
```

```
PriceTree = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {1x21 cell}
      tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
      dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]


EqtTre = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {1x21 cell}
      tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
      dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]


DbtTree = struct with fields:
    FinObj: 'TrinPriceTree'
     PTree: {1x21 cell}
      tObs: [0 0.2500 0.5000 0.7500 1 1.2500 1.5000 1.7500 2 2.2500 ... ]
      dObs: [735965 736056 736147 736238 736330 736421 736512 736604 ... ]
```

Compute the delta and gamma of the convertible bond.

```
InstSet= instcbond(CouponRate,Settle,Maturity,ConvRatio,'Spread',Spread,...
'CallExDates',CallExDates,'CallStrike',CallStrike,'AmericanCall',1);
[Delta,Gamma] = sttsens(STTTree,InstSet)
```

```
Delta = 0.3945
```

```
Gamma = 0.0324
```

Input Arguments

STTTree — Stock tree structure for standard trinomial tree

structure

Stock tree structure for a standard trinomial tree, specified by using `stttree`.

Data Types: `struct`

`InstSet` — Variable containing a collection instruments
structure

Variable containing a collection of `NINST` instruments, specified as a structure. Instruments are broken down by type and each type can have different data fields.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Delta,Gamma,Vega,Price] = sttsens(STTTree,InstSet,'Options',deriv)`

`Options` — Derivatives pricing options
structure

Derivatives pricing options, specified as the comma-separated pair consisting of `'Options'` and a structure that is created with `derivset`.

Data Types: `struct`

Output Arguments

`Delta` — Rate of change of instruments prices with respect to changes in the stock price
vector of deltas

Rate of change of instrument prices with respect to changes in the stock price, returned as a `NINST`-by-`1` vector of deltas. For more information on the stock tree, see `stttree`.

`Gamma` — Rate of change of instrument deltas with respect to changes in the stock price
vector of gammas

Rate of change of instrument deltas with respect to changes in the stock price, returned as a `NINST`-by-`1` vector of gammas.

`Vega` — Rate of change of instrument prices with respect to changes in the volatility of the stock price
vector of vegas

Rate of change of instrument prices with respect to changes in the volatility of the stock price, returned as a `NINST`-by-`1` vector of vegas. For more information on the stock tree, see `stttree`.

`Price` — Expected prices for each instrument at time `0`
matrix

Expected prices for each instrument at time `0`, returned as a `NINST`-by-`1` vector. The prices are computed by backward dynamic programming on the standard trinomial (STT) stock tree. If an instrument cannot be priced, a `NaN` is returned in that entry.

Version History

Introduced in R2015b

See Also

`sttree` | `sttsens` | `stttimespec` | `derivset`

Topics

“Convertible Bond” on page 2-4

stttimespec

Specify time structure for standard trinomial tree

Syntax

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

Description

`TimeSpec = stttimespec(ValuationDate,Maturity,NumPeriods)` creates a time spec for a standard trinomial (STT) tree.

Examples

Create a stttimespec to Build a STTTree

Create a `RateSpec`.

```
StartDates = datetime(2014,1,1);
EndDates = datetime(2018,1,1);
Rates = 0.025;
Basis = 1;
Compounding = -1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9048
            Rates: 0.0250
         EndTimes: 4
       StartTimes: 0
         EndDates: 737061
       StartDates: 735600
    ValuationDate: 735600
            Basis: 1
     EndMonthRule: 1
```

Create a `StockSpec`.

```
AssetPrice = 110;
Sigma = 0.22;
Div = 0.02;
StockSpec = stockspec(Sigma, AssetPrice, 'continuous', Div)
```

```
StockSpec = struct with fields:
              FinObj: 'StockSpec'
               Sigma: 0.2200
```

```
         AssetPrice: 110
       DividendType: {'continuous'}
    DividendAmounts: 0.0200
    ExDividendDates: []
```

Create a `STTTimespec` and `STTTree`.

```
NumPeriods = length(cfdates(StartDates,EndDates,12));
TimeSpec = stttimespec(StartDates, EndDates, NumPeriods)
```

```
TimeSpec = struct with fields:
           FinObj: 'STTTimeSpec'
    ValuationDate: 735600
         Maturity: 737061
       NumPeriods: 48
            Basis: 0
     EndMonthRule: 1
             tObs: [0 0.0833 0.1667 0.2500 0.3333 0.4167 0.5000 0.5833 ... ]
             dObs: [735600 735630 735660 735691 735721 735752 735782 ... ]
```

```
STTT = stttree(StockSpec, RateSpec, TimeSpec)
```

```
STTT = struct with fields:
       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.0833 0.1667 0.2500 0.3333 0.4167 0.5000 0.5833 ... ]
         dObs: [735600 735630 735660 735691 735721 735752 735782 735813 ... ]
        STree: {1x49 cell}
        Probs: {1x48 cell}
```

Input Arguments

`ValuationDate` — Date marking the pricing date and first observation tree

datetime scalar | string scalar | date character vector

Date marking the pricing date and first observation in the tree, specified as a scalar datetime, string, or date character vector.

To support existing code, `stttimespec` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Date marking the depth of tree

datetime scalar | string scalar | date character vector

Date marking the depth of the tree, specified as a scalar using a scalar datetime, string, or date character vector.

To support existing code, `stttimespec` also accepts serial date numbers as inputs, but they are not recommended.

`NumPeriods` — Determines how many time steps are in tree
nonnegative integer

Determines how many time steps are in tree, specified as a scalar using a nonnegative integer value.

Data Types: `double`

Output Arguments

`TimeSpec` — Time layout for standard trinomial (STT) tree
structure

Time layout for standard trinomial (STT) tree, returned as a structure.

Version History

Introduced in R2015b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `stttimespec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`stttree`

sttree

Build standard trinomial tree

Syntax

```
STTTree = sttree(StockSpec,RateSpec,TimeSpec)
```

Description

`STTTree = sttree(StockSpec,RateSpec,TimeSpec)` builds a standard trinomial (STT) tree.

Examples

Build a STTTree

Create a `RateSpec`.

```
StartDates = 'Jan-1-2014';
EndDates = 'Jan-1-2018';
Rates = 0.025;
Basis = 1;
Compounding = -1;
RateSpec = intenvset('ValuationDate', StartDates, 'StartDates', StartDates,...
'EndDates', EndDates, 'Rates', Rates,'Compounding', Compounding, 'Basis', Basis)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: 0.9048
            Rates: 0.0250
         EndTimes: 4
       StartTimes: 0
         EndDates: 737061
       StartDates: 735600
    ValuationDate: 735600
            Basis: 1
     EndMonthRule: 1
```

Create a `StockSpec`.

```
AssetPrice = 55;
Sigma = 0.22;
Div = 0.02;
StockSpec = stockspec(Sigma, AssetPrice, 'continuous', Div)
```

```
StockSpec = struct with fields:
             FinObj: 'StockSpec'
              Sigma: 0.2200
         AssetPrice: 55
       DividendType: {'continuous'}
    DividendAmounts: 0.0200
```

```
    ExDividendDates: []
```

Create a Standard Trinomial Tree (STTTree).

```
NumSteps = 8;
TimeSpec = stttimespec(StartDates, EndDates, NumSteps);
STTT = stttree(StockSpec, RateSpec, TimeSpec)
```

```
STTT = struct with fields:
       FinObj: 'STStockTree'
    StockSpec: [1x1 struct]
     TimeSpec: [1x1 struct]
     RateSpec: [1x1 struct]
         tObs: [0 0.5000 1 1.5000 2 2.5000 3 3.5000 4]
         dObs: [735600 735782 735965 736147 736330 736513 736695 736878 ... ]
        STree: {1x9 cell}
        Probs: {1x8 cell}
```

Input Arguments

StockSpec — Stock specification for underlying asset

structure

Stock specification for underlying asset, specified using StockSpec obtained from stockspec. For information on the stock specification, see stockspec.

stockspec can handle other types of underlying assets. For example, stocks, stock indices, and commodities. If dividends are not specified in StockSpec, dividends are assumed to be 0.

Data Types: struct

RateSpec — Interest-rate term specification of initial risk-free rate curve

structure

Interest-rate term specification of initial risk-free rate curve, specified by the RateSpec obtained from intenvset. For information on the interest-rate specification, see intenvset.

Data Types: struct

TimeSpec — Tree time layout specification

structure

Tree time layout specification, specified using stttimespec to define the observation dates of the standard trinomial (STT) tree.

Data Types: struct

Output Arguments

STTTree — Tree specifying stock and time information for a standard trinomial (STT) tree

tree structure

Tree specifying stock and time information for a standard trinomial (STT) tree, returned as a tree structure.

Version History

Introduced in R2015b

See Also

`stttimespec`

Topics

“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194

supersharebybls

Determine price of supershare digital options using Black-Scholes model

Syntax

```
Price = supersharebybls(RateSpec,StockSpec,Settle,Maturity,StrikeLow,
StrikeHigh)
```

Description

`Price = supersharebybls(RateSpec,StockSpec,Settle,Maturity,StrikeLow, StrikeHigh)` computes supershare digital options using the Black-Scholes option pricing model.

Examples

Compute the Price of Supershare Digital Options Using Black-Scholes Model

This example shows how to compute the price of supershare digital options using Black-Scholes model. 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.

```
Settle = datetime(2008,11,1);
Maturity = datetime(2009,2,1);
Rates = 0.045;
Basis = 1;
Compounding = -1;

% create the RateSpec
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
```

Input Arguments

RateSpec — Interest-rate term structure
structure

Interest-rate term structure (annualized and continuously compounded), specified by the RateSpec obtained from intenvset. For information on the interest-rate specification, see intenvset.

Data Types: struct

StockSpec — Stock specification for underlying asset
structure

Stock specification for the underlying asset. For information on the stock specification, see stockspec.

stockspec handles several types of underlying assets. For example, for physical commodities the price is StockSpec.Asset, the volatility is StockSpec.Sigma, and the convenience yield is StockSpec.DividendAmounts.

Data Types: struct

Settle — Settlement or trade date
datetime array | string array | date character vector

Settlement or trade date for the basket option, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, supersharebybls also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date for the basket option, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, supersharebybls also accepts serial date numbers as inputs, but they are not recommended.

StrikeLow — Low strike price values
vector

Low strike price values, specified as an NINST-by-1 vector.

Data Types: double

StrikeHigh — High strike price values
vector

High strike price values, specified as an NINST-by-1 vector.

Data Types: double

Output Arguments

`Price` — Expected prices for supershare option

vector

Expected prices for supershare option, returned as a `NINST`-by-`1` vector.

More About

Supershare Option

A supershare option 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.

For more information, see “Digital Option” on page 3-26.

Version History

Introduced in R2009a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `supersharebybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`assetbybls` | `cashbybls` | `gapbybls` | `supersharesensbybls`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Pricing Using the Black-Scholes Model” on page 3-82
“Digital Option” on page 3-26
“Supported Equity Derivative Functions” on page 3-19

supersharesensbybls

Determine price or sensitivities of supershare digital options using Black-Scholes model

Syntax

```
PriceSens = supersharesensbybls(RateSpec,StockSpec,Settle,Maturity,StrikeLow,
StrikeHigh)
PriceSens = supersharesensbybls( ___ ,Name,Value)
```

Description

`PriceSens = supersharesensbybls(RateSpec,StockSpec,Settle,Maturity,StrikeLow, StrikeHigh)` computes price or sensitivities of supershare digital options using the Black-Scholes option pricing model.

`PriceSens = supersharesensbybls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute Price and Sensitivities of Supershare Digital Options Using Black-Scholes Model

This example shows how to compute price and sensitivities of supershare digital options using a Black-Scholes model. 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.

```
Settle = datetime(2008,11,1);
Maturity = datetime(2009,2,1);
Rates = 0.045;
Basis = 1;
Compounding = -1;

% define the RateSpec
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
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `supersharesensbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date for the basket option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `supersharesensbybls` also accepts serial date numbers as inputs, but they are not recommended.

`StrikeLow` — Low strike price values

vector

Low strike price values, specified as an `NINST`-by-`1` vector.

Data Types: double

StrikeHigh — High strike price values
vector

High strike price values, specified as an NINST-by-1 vector.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Gamma,Theta,Price] = supersharesensbybls(RateSpec,StockSpec,Settle,Maturity,StrikeLow,StrikeHigh,'OutSpec',{'gamma';'theta';'price'})

OutSpec — Define outputs
{'Price'} (default) | character vector with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All' | cell array of character vectors with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and a NOUT- by-1 or a 1-by-NOUT cell array of character vectors with possible values of 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'.

OutSpec = {'All'} specifies that the output is Delta, Gamma, Vega, Lambda, Rho, Theta, and Price, in that order. This is the same as specifying OutSpec to include each sensitivity.

Example: OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}

Data Types: char | cell

Output Arguments

PriceSens — Expected prices or sensitivities for supershare option
vector

Expected prices or sensitivities for supershare option, returned as a NINST-by-1 vector.

More About

Supershare Option

A supershare option 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.

For more information, see “Digital Option” on page 3-26.

Version History

Introduced in R2009a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `supersharesensbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`supersharebybls`

Topics

“Equity Derivatives Using Closed-Form Solutions” on page 3-79
“Pricing European Call Options Using Different Equity Models” on page 3-88
“Pricing Using the Black-Scholes Model” on page 3-82
“Digital Option” on page 3-26
“Supported Equity Derivative Functions” on page 3-19

swapbybdt

Price swap instrument from Black-Derman-Toy interest-rate tree

Syntax

```
[Price,PriceTree,CFTree,SwapRate] = swapbybdt(BDTTree,LegRate,Settle,
Maturity)
[Price,PriceTree,CFTree,SwapRate] = swapbybdt( ___ ,Name,Value)
```

Description

`[Price,PriceTree,CFTree,SwapRate] = swapbybdt(BDTTree,LegRate,Settle, Maturity)` prices a swap instrument from a Black-Derman-Toy interest-rate tree. `swapbybdt` computes prices of vanilla swaps, amortizing swaps and forward swaps.

Note Alternatively, you can use the `Swap` object to price swap instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree,CFTree,SwapRate] = swapbybdt( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price an Interest-Rate Swap

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 = datetime(2000,1,1);
Maturity = datetime(2003,1,1);
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
```

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-14

PriceTree = struct with fields:
    FinObj: 'BDTPriceTree'
      tObs: [0 1 2 3 4]
     PTree: {1x5 cell}


CFTree = struct with fields:
    FinObj: 'BDTCFTree'
      tObs: [0 1 2 3 4]
    CFTree: {[NaN]  [NaN NaN]  [NaN NaN NaN]  [NaN NaN NaN NaN]  [NaN NaN NaN NaN]}


SwapRate = 0.1205
```

Price an Amortizing Swap

Price an amortizing swap using the `Principal` input argument to define the amortization schedule.

Create the `RateSpec`.

```
Rates = 0.035;
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = datetime(2017,1,1);
Compounding = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.8135
            Rates: 0.0350
         EndTimes: 6
       StartTimes: 0
         EndDates: 736696
```

```
       StartDates: 734504
    ValuationDate: 734504
            Basis: 0
     EndMonthRule: 1
```

Create the swap instrument using the following data:

```
Settle = datetime(2011,1,1);
Maturity = datetime(2017,1,1);
Period = 1;
LegRate = [0.04 10];
```

Define the swap amortizing schedule.

```
Principal ={{datetime(2013,1,1) 100;datetime(2014,1,1) 80;datetime(2015,1,1) 60;datetime(2016,1,1
```

Build the BDT tree and assume volatility is 10%.

```
MatDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; d
BDTTimeSpec = bdttimespec(ValuationDate, MatDates);
Volatility = 0.10;
BDTVolSpec = bdtvolspec(ValuationDate, MatDates, Volatility*ones(1,length(MatDates))');
BDTT = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Compute the price of the amortizing swap.

```
Price = swapbybdt(BDTT, LegRate, Settle, Maturity, 'Principal' , Principal)
```

```
Price = 1.4574
```

Price a Forward Swap

Price a forward swap using the `StartDate` input argument to define the future starting date of the swap.

Create the `RateSpec`.

```
Rates = 0.0325;
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = datetime(2018,1,1);
Compounding = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.8254
            Rates: 0.0325
         EndTimes: 6
       StartTimes: 0
         EndDates: 737061
       StartDates: 734869
```

```
    ValuationDate: 734869
            Basis: 0
     EndMonthRule: 1
```

Build the tree with a volatility of 10%.

```
MatDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1) ; d
BDTTimeSpec = bdttimespec(ValuationDate, MatDates);
Volatility = 0.10;
BDTVolSpec = bdtvolspec(ValuationDate, MatDates, Volatility*ones(1,length(MatDates))');
BDTT = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Compute the price of a forward swap that starts in two years (Jan 1, 2014) and matures in three years with a forward swap rate of 3.85%.

```
Settle = datetime(2012,1,1);
Maturity = datetime(2017,1,1);
StartDate = datetime(2014,1,1);
LegRate = [0.0385 10];

Price = swapbybdt(BDTT, LegRate, Settle, Maturity, 'StartDate', StartDate)
```

```
Price = 1.3203
```

Using the previous data, compute the forward swap rate, the coupon rate for the fixed leg, such that the forward swap price at time = 0 is zero.

```
LegRate = [NaN 10];
[Price, ~,~, SwapRate] = swapbybdt(BDTT, LegRate, Settle, Maturity, 'StartDate', StartDate)
```

```
Price = -4.9738e-12

SwapRate = 0.0335
```

Input Arguments

BDTTree — Interest-rate structure

structure

Interest-rate tree structure, created by `bdttree`

Data Types: `struct`

LegRate — Leg rate

matrix

Leg rate, specified as a `NINST`-by-2 matrix, with each row defined as one of the following:

- `[CouponRate Spread]` (fixed-float)
- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

`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.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swapbybdt` also accepts serial date numbers as inputs, but they are not recommended.

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

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each swap.

To support existing code, `swapbybdt` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree,CFTree,SwapRate] = swapbybdt(BDTTree,LegRate,Settle,Maturity,LegReset,Basis,Principal,LegType)`

`LegReset` — Reset frequency per year for each swap
`[1 1]` (default) | vector

Reset frequency per year for each swap, specified as the comma-separated pair consisting of `'LegReset'` and a `NINST`-by-`2` vector.

Data Types: `double`

`Basis` — Day-count basis representing the basis for each leg
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis for each leg, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` array (or `NINST`-by-`2` if `Basis` is different for each leg).

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360

- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts or principal value schedules, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array (or `NINST`-by-`2` if `Principal` is different for each leg) of the notional principal amounts or principal value schedules. For schedules, each element of the cell array is a `NumDates`-by-`2` array where the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`LegType` — Leg type
`[1 0]` for each instrument (default) | matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float)

Leg type, specified as the comma-separated pair consisting of `'LegType'` and a `NINST`-by-`2` matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float). 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`. `LegType` allows `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float) swaps

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` (or `NINST`-by-`2` if `EndMonthRule` is different for each leg).

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` (or `NINST`-by-`2` if `AdjustCashFlowsBasis` is different for each leg) of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-`2` if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`StartDate` — Date swap actually starts
`Settle` date (default) | datetime array | string array | date character vector

Date swap actually starts, specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swapbybdt` also accepts serial date numbers as inputs, but they are not recommended.

Use this argument to price forward swaps, that is, swaps that start in a future date

Output Arguments

`Price` — Expected swap prices at time 0

vector

Expected swap prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as 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.

`CFTree` — Swap cash flows

structure

Swap cash flows, returned as 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` — Rates applicable to fixed leg

matrix

Rates applicable to the fixed leg, returned as 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`.

More About

Amortizing Swap

In an amortizing swap, the notional principal decreases periodically because it is tied to an underlying financial instrument with a declining (amortizing) principal balance, such as a mortgage.

Forward Swap

Agreement to enter into an interest-rate swap arrangement on a fixed date in future.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swapbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdttree` | `capbybdt` | `cfbybdt` | `floorbybdt` | `Swap`

Topics

“Computing Instrument Prices” on page 2-81
“Pricing a Portfolio Using the Black-Derman-Toy Model” on page 1-10
“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Swap” on page 2-13
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

swapbybk

Price swap instrument from Black-Karasinski interest-rate tree

Syntax

```
[Price,PriceTree,CFTree,SwapRate] = swapbybk(BKTree,LegRate,Settle,Maturity)
[Price,PriceTree,CFTree,SwapRate] = swapbybk( ___ ,Name,Value)
```

Description

`[Price,PriceTree,CFTree,SwapRate] = swapbybk(BKTree,LegRate,Settle,Maturity)` prices a swap instrument from a Black-Karasinski interest-rate tree. `swapbybk` computes prices of vanilla swaps, amortizing swaps and forward swaps.

Note Alternatively, you can use the `Swap` object to price swap instruments. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`[Price,PriceTree,CFTree,SwapRate] = swapbybk( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price an Interest-Rate Swap

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, 2004
- Swap maturity date: Jan. 01, 2006

Based on the information above, set the required arguments and build the `LegRate`, `LegType`, and `LegReset` matrices:

```
Settle = datetime(2004,1,1);
Maturity = datetime(2006,1,1);
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 `BKTree` included in the MAT-file `deriv.mat`. `BKTree` contains the time and forward-rate information needed to price the instrument.

```
load deriv.mat;
```

Use swapbybk to price of the swap.

```
Price = swapbybk(BKTree, LegRate,...
Settle, Maturity, LegReset, Basis, Principal, LegType)
```

```
Price = 5.0425
```

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 = -2.8422e-14

PriceTree = struct with fields:
     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 = 0.0336
```

Price an Amortizing Swap

Price an amortizing swap using the Principal input argument to define the amortization schedule.

Create the RateSpec.

```
Rates = 0.035;
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = datetime(2017,1,1);
Compounding = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.8135
            Rates: 0.0350
         EndTimes: 6
       StartTimes: 0
         EndDates: 736696
       StartDates: 734504
    ValuationDate: 734504
            Basis: 0
     EndMonthRule: 1
```

Create the swap instrument using the following data:

```
Settle ='1-Jan-2011';
Maturity = '1-Jan-2017';
Period = 1;
LegRate = [0.04 10];
```

Define the swap amortizing schedule.

```
Principal ={{datetime(2013,1,1) 100;datetime(2014,1,1) 80;datetime(2015,1,1) 60;datetime(2016,1,1
```

Build the BK tree and assume volatility is 10%.

```
MatDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; d
BKTimeSpec = bktimespec(ValuationDate, MatDates);
Volatility = 0.10;
AlphaDates = datetime(2017,1,1);
AlphaCurve = 0.1;
BKVolSpec = bkvolspec(ValuationDate, MatDates, Volatility*ones(1,length(MatDates))',...
AlphaDates, AlphaCurve);
BKT = bktree(BKVolSpec, RateSpec, BKTimeSpec);
```

Compute the price of the amortizing swap.

```
Price = swapbybk(BKT, LegRate, Settle, Maturity, 'Principal' , Principal)
```

```
Price = 1.4574
```

Price a Forward Swap

Price a forward swap using the `StartDate` input argument to define the future starting date of the swap.

Create the `RateSpec`.

```
Rates = 0.0374;
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = datetime(2018,1,1);
Compounding = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.8023
            Rates: 0.0374
         EndTimes: 6
       StartTimes: 0
         EndDates: 737061
       StartDates: 734869
    ValuationDate: 734869
            Basis: 0
```

```
    EndMonthRule: 1
```

Build a BK tree.

```
VolDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1) ; d
VolCurve = 0.1;
AlphaDates = datetime(2018,1,1);
AlphaCurve = 0.1;

BKVolSpec = bkvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
BKTimeSpec = bktimespec(RateSpec.ValuationDate, VolDates, Compounding);
BKT = bktree(BKVolSpec, RateSpec, BKTimeSpec);
```

Compute the price of a forward swap that starts in a year (Jan 1, 2013) and matures in four years with a forward swap rate of 4.25%.

```
Settle = datetime(2012,1,1);
Maturity = datetime(2017,1,1);
StartDate = datetime(2013,1,1);
LegRate = [0.0425 10];

Price = swapbybk(BKT, LegRate, Settle, Maturity, 'StartDate', StartDate)
```

```
Price = 1.4434
```

Using the previous data, compute the forward swap rate, the coupon rate for the fixed leg, such that the forward swap price at time = 0 is zero.

```
LegRate = [NaN 10];
[Price, ~,SwapRate] = swapbybk(BKT, LegRate, Settle, Maturity, 'StartDate', StartDate)
```

```
Price = 1.4211e-14
```

```
SwapRate = 0.0384
```

Input Arguments

BKTree — Interest-rate structure

structure

Interest-rate tree structure, created by `bktree`

Data Types: `struct`

LegRate — Leg rate

matrix

Leg rate, specified as a `NINST`-by-2 matrix, with each row defined as one of the following:

- `[CouponRate Spread]` (fixed-float)
- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

`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.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swapbybk` also accepts serial date numbers as inputs, but they are not recommended.

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

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors representing the maturity date for each swap.

To support existing code, `swapbybk` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree,CFTree,SwapRate] = swapbybk(BKTree,LegRate,Settle,Maturity,LegReset,Basis,Principal,LegType)`

`LegReset` — Reset frequency per year for each swap
`[1 1]` (default) | vector

Reset frequency per year for each swap, specified as the comma-separated pair consisting of `'LegReset'` and a `NINST`-by-`2` vector.

Data Types: `double`

`Basis` — Day-count basis representing the basis for each leg
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis for each leg, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` array (or `NINST`-by-`2` if `Basis` is different for each leg).

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360

- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts or principal value schedules, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array (or `NINST`-by-`2` if `Principal` is different for each leg) of the notional principal amounts or principal value schedules. For schedules, each element of the cell array is a `NumDates`-by-`2` array where the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`LegType` — Leg type
`[1 0]` for each instrument (default) | matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float)

Leg type, specified as the comma-separated pair consisting of `'LegType'` and a `NINST`-by-`2` matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float). 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`. `LegType` allows `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float) swaps

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` (or `NINST`-by-`2` if `EndMonthRule` is different for each leg).

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` (or `NINST`-by-`2` if `AdjustCashFlowsBasis` is different for each leg) of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-`2` if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`StartDate` — Date swap actually starts
`Settle` date (default) | datetime array | string array | date character vector

Date swap actually starts, specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swapbybk` also accepts serial date numbers as inputs, but they are not recommended.

Use this argument to price forward swaps, that is, swaps that start in a future date

Output Arguments

`Price` — Expected swap prices at time 0

vector

Expected swap prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

`CFTree` — Swap cash flows

structure

Swap cash flows, returned as 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` — Rates applicable to fixed leg

matrix

Rates applicable to the fixed leg, returned as 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`.

More About

Amortizing Swap

In an amortizing swap, the notional principal decreases periodically because it is tied to an underlying financial instrument with a declining (amortizing) principal balance, such as a mortgage.

Forward Swap

Agreement to enter into an interest-rate swap arrangement on a fixed date in future.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swapbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bktree` | `bondbybk` | `capbybk` | `fixedbybk` | `floorbybk` | `Swap`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Swap” on page 2-13
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

swapbycir

Price swap instrument from Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree,SwapRate] = swapbycir(CIRTree,LegRate,Settle,Maturity)
[Price,PriceTree,SwapRate] = swapbycir( ___ ,Name,Value)
```

Description

`[Price,PriceTree,SwapRate] = swapbycir(CIRTree,LegRate,Settle,Maturity)` prices a swap instrument from a Cox-Ingersoll-Ross (CIR) interest-rate tree. `swapbycir` computes prices of vanilla swaps, amortizing swaps, and forward swaps using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree,SwapRate] = swapbycir( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price an Interest-Rate Swap Using a CIR Interest-Rate Tree

Define 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.

```
Basis = 0;
Principal = 100;
LegRate = [0.06 20]; % [CouponRate Spread]
LegType = [1 0]; % [Fixed Float]
LegReset = [1 1]; % Payments once per year
```

Create a `RateSpec` using the `intenvset` function.

```
Rates = [0.035; 0.042147; 0.047345; 0.052707];
Dates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; date
ValuationDate = datetime(2017,1,1);
EndDates = Dates(2:end)';
Compounding = 1;
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',End
```

Create a `CIR` tree.

```
NumPeriods = 5;
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Settle = datetime(2017,1,1);
Maturity = datetime(2022,1,1);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);
```

```
CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)

CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 4]
        dObs: [736696 737061 737426 737791 738156]
     FwdTree: {1x5 cell}
     Connect: {[3x1 double]  [3x3 double]  [3x5 double]  [3x7 double]}
       Probs: {[3x1 double]  [3x3 double]  [3x5 double]  [3x7 double]}
```

Price the interest-rate swap.

```
[Price,PriceTree] = swapbycir(CIRT,LegRate,Settle,Maturity,'LegReset',LegReset,'Basis',3,'Princi

Price = 2.5522

PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
       tObs: [0 1 2 3 4 5]
      PTree: {1x6 cell}
    Connect: {[3x1 double]  [3x3 double]  [3x5 double]  [3x7 double]}
```

Input Arguments

`CIRTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `cirtree`

Data Types: `struct`

`LegRate` — Leg rate

matrix

Leg rate, specified as a `NINST`-by-2 matrix, with each row defined as one of the following:

- `[CouponRate Spread]` (fixed-float)
- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

`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.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `swapbycir` also accepts serial date numbers as inputs, but they are not recommended.

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

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each swap.

To support existing code, `swapbycir` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree,SwapRate] = swapbycir(CIRTree,LegRate,Settle,Maturity,LegReset,Basis,Principal,LegType)`

`LegReset` — Reset frequency per year for each swap
`[1 1]` (default) | vector

Reset frequency per year for each swap, specified as the comma-separated pair consisting of `'LegReset'` and a `NINST`-by-2 vector.

Data Types: `double`

`Basis` — Day-count basis representing the basis for each leg
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis for each leg, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-1 array (or `NINST`-by-2 if `Basis` is different for each leg).

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts or principal value schedules, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array (or `NINST`-by-`2` if `Principal` is different for each leg) of the notional principal amounts or principal value schedules. For schedules, each element of the cell array is a `NumDates`-by-`2` array where the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`LegType` — Leg type
`[1 0]` for each instrument (default) | matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float)

Leg type, specified as the comma-separated pair consisting of `'LegType'` and a `NINST`-by-`2` matrix with values:

- `[1 1]` (fixed-fixed) swap
- `[1 0]` (fixed-float) swap
- `[0 1]` (float-fixed) swap
- `[0 0]` (float-float) swap

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`.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` (or `NINST`-by-`2` if `EndMonthRule` is different for each leg).

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

AdjustCashFlowsBasis — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` (or `NINST`-by-`2` if `AdjustCashFlowsBasis` is different for each leg) of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

BusinessDayConvention — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-`2` if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

Holidays — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

StartDate — Date swap actually starts
`Settle` date (default) | datetime array | string array | date character vector

Date swap actually starts, specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector of dates using a datetime array, string array, or date character vectors.

To support existing code, `swapbycir` also accepts serial date numbers as inputs, but they are not recommended.

Use this argument to price forward swaps, that is, swaps that start in a future date

Output Arguments

`Price` — Expected swap prices at time 0

vector

Expected swap prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as a MATLAB structure of trees containing vectors of swaption instrument prices and a vector of observation times for each node. Within `PriceTree`:

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

`SwapRate` — Rates applicable to fixed leg

matrix

Rates applicable to the fixed leg, returned as 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`.

More About

Amortizing Swap

In an amortizing swap, the notional principal decreases periodically because it is tied to an underlying financial instrument with a declining (amortizing) principal balance, such as a mortgage.

Forward Swap

Agreement to enter into an interest-rate swap arrangement on a fixed date in future.

Version History

Introduced in R2018a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swapbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J., and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swaptionbycir` | `instswap`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Swap” on page 2-13
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

swapbyhjm

Price swap instrument from Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree,CFTree,SwapRate] = swapbyhjm(HJMTree,LegRate,Settle,
Maturity)
[Price,PriceTree,CFTree,SwapRate] = swapbyhjm( ___ ,Name,Value)
```

Description

`[Price,PriceTree,CFTree,SwapRate] = swapbyhjm(HJMTree,LegRate,Settle, Maturity)` prices a swap instrument from a Heath-Jarrow-Morton interest-rate tree. `swapbyhjm` computes prices of vanilla swaps, amortizing swaps and forward swaps.

`[Price,PriceTree,CFTree,SwapRate] = swapbyhjm( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price an Interest-Rate Swap

This example shows how to 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 = datetime(2000,1,1);
Maturity = datetime(2003,1,1);
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.

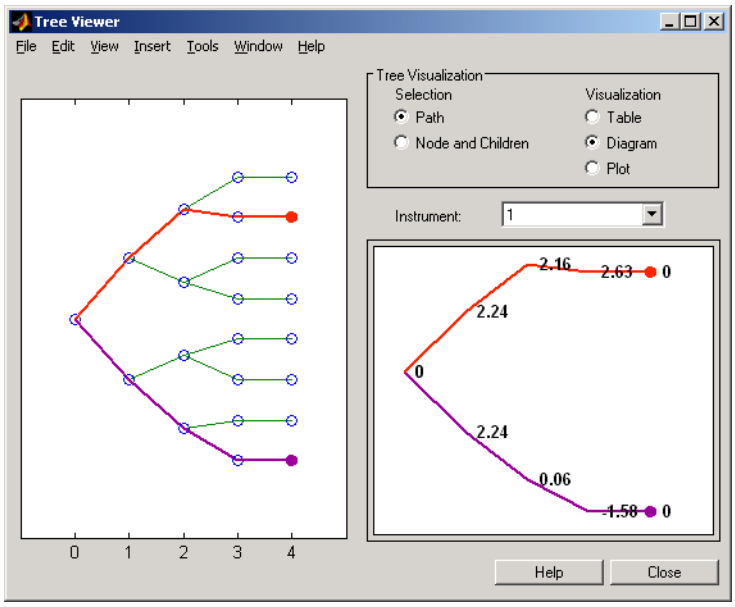
```
load deriv.mat;
```

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]}
```

Use `treeviewer` to 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)
```



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 receive this amount. But if interest rates go up, your cash flow is negative (-$1.58), meaning that you owe this amount.

`treeviewer` price tree diagrams follow the convention that increasing prices appear on the upper branch of a tree and, so, 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).

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
```

Price an Amortizing Swap

Price an amortizing swap using the `Principal` input argument to define the amortization schedule.

Create the `RateSpec`.

```
Rates = 0.035;
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = '1-Jan-2017';
Compounding = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.8135
            Rates: 0.0350
          EndTimes: 6
        StartTimes: 0
          EndDates: 736696
        StartDates: 734504
    ValuationDate: 734504
            Basis: 0
     EndMonthRule: 1
```

Create the swap instrument using the following data:

```
Settle = datetime(2011,1,1);
Maturity = datetime(2017,1,1);
```

```
Period = 1;
LegRate = [0.04 10];
```

Define the swap amortizing schedule.

```
Principal ={{datetime(2013,1,1) 100;datetime(2014,1,1) 80;datetime(2015,1,1) 60;datetime(2016,1,1
```

Build the HJM tree using the following data:

```
MatDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; d
HJMTimeSpec = hjmtimespec(RateSpec.ValuationDate, MatDates);
Volatility = [.10; .08; .06; .04];
CurveTerm = [ 1; 2; 3; 4];
HJMVolSpec = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
HJMT = hjmtree(HJMVolSpec,RateSpec,HJMTimeSpec);
```

Compute the price of the amortizing swap.

```
Price = swapbyhjm(HJMT, LegRate, Settle, Maturity, 'Principal', Principal)
```

```
Price = 1.4574
```

Price a Forward Swap

Price a forward swap using the `StartDate` input argument to define the future starting date of the swap.

Create the `RateSpec`.

```
Rates = 0.0374;
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = datetime(2018,1,1);
Compounding = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.8023
            Rates: 0.0374
         EndTimes: 6
       StartTimes: 0
         EndDates: 737061
       StartDates: 734869
    ValuationDate: 734869
            Basis: 0
     EndMonthRule: 1
```

Build an HJM tree.

```
MatDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1) ; d
HJMTimeSpec = hjmtimespec(RateSpec.ValuationDate, MatDates);
```

```
Volatility = [.10; .08; .06; .04];
CurveTerm = [ 1; 2; 3; 4];
HJMVolSpec = hjmvolspec('Proportional', Volatility, CurveTerm, 1e6);
HJMT = hjmtree(HJMVolSpec,RateSpec,HJMTimeSpec);
```

Compute the price of a forward swap that starts in a year (Jan 1, 2013) and matures in four years with a forward swap rate of 4.25%.

```
Settle = datetime(2012,1,1);
Maturity = datetime(2017,1,1);
StartDate = datetime(2013,1,1);
LegRate = [0.0425 10];

Price = swapbyhjm(HJMT, LegRate, Settle, Maturity, 'StartDate', StartDate)
```

```
Price = 1.4434
```

Using the previous data, compute the forward swap rate, the coupon rate for the fixed leg, such that the forward swap price at time = 0 is zero.

```
LegRate = [NaN 10];
[Price, ~,~, SwapRate] = swapbyhjm(HJMT, LegRate, Settle, Maturity, 'StartDate', StartDate)
```

```
Price = 0

SwapRate = 0.0384
```

Input Arguments

`HJMTree` — Interest-rate structure

structure

Interest-rate tree structure, created by `hjmtree`

Data Types: `struct`

`LegRate` — Leg rate

matrix

Leg rate, specified as a `NINST`-by-2 matrix, with each row defined as one of the following:

- `[CouponRate Spread]` (fixed-float)
- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

`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.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `swapbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

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

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each swap.

To support existing code, `swapbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree,CFTree,SwapRate] = swapbyhjm(HJMTree,LegRate,Settle,Maturity,LegReset,Basis,Principal,LegType)`

`LegReset` — Reset frequency per year for each swap
`[1 1]` (default) | vector

Reset frequency per year for each swap, specified as the comma-separated pair consisting of `'LegReset'` and a `NINST`-by-2 vector.

Data Types: `double`

`Basis` — Day-count basis representing the basis for each leg
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis for each leg, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-1 array (or `NINST`-by-2 if `Basis` is different for each leg).

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules
`100` (default) | vector or cell array

Notional principal amounts or principal value schedules, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-`1` vector or `NINST`-by-`1` cell array (or `NINST`-by-`2` if `Principal` is different for each leg) of the notional principal amounts or principal value schedules. For schedules, each element of the cell array is a `NumDates`-by-`2` array where the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`LegType` — Leg type
`[1 0]` for each instrument (default) | matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float)

Leg type, specified as the comma-separated pair consisting of `'LegType'` and a `NINST`-by-`2` matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float). 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`. `LegType` allows `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float) swaps

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` (or `NINST`-by-`2` if `EndMonthRule` is different for each leg).

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.

- 1 = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: logical

AdjustCashFlowsBasis — Flag to adjust cash flows based on actual period day count
false (default) | value of 0 (false) or 1 (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of 'AdjustCashFlowsBasis' and a NINST-by-1 (or NINST-by-2 if AdjustCashFlowsBasis is different for each leg) of logicals with values of 0 (false) or 1 (true).

Data Types: logical

BusinessDayConvention — Business day conventions
actual (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of 'BusinessDayConvention' and a character vector or a N-by-1 (or NINST-by-2 if BusinessDayConvention is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- actual — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- follow — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- modifiedfollow — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- previous — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- modifiedprevious — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: char | cell

Holidays — Holidays used in computing business days
if not specified, the default is to use holidays.m (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of 'Holidays' and MATLAB dates using a NHolidays-by-1 vector.

Data Types: datetime

StartDate — Date swap actually starts
Settle date (default) | datetime array | string array | date character vector

Date swap actually starts, specified as the comma-separated pair consisting of 'StartDate' and a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, swapbyhjm also accepts serial date numbers as inputs, but they are not recommended.

Use this argument to price forward swaps, that is, swaps that start in a future date

Output Arguments

`Price` — Expected swap prices at time 0

vector

Expected swap prices at time 0, returned as a `NINST`-by-1 vector.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as a MATLAB structure of trees containing vectors of swaption instrument prices and a vector of observation times for each node. Within `PriceTree`:

- `PriceTree.tObs` contains the observation times.
- `PriceTree.PBush` contains the clean prices.

`CFTree` — Swap cash flows

structure

Swap cash flows, returned as 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` — Rates applicable to fixed leg

matrix

Rates applicable to the fixed leg, returned as 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`.

More About

Amortizing Swap

In an amortizing swap, the notional principal decreases periodically because it is tied to an underlying financial instrument with a declining (amortizing) principal balance, such as a mortgage.

Forward Swap

Agreement to enter into an interest-rate swap arrangement on a fixed date in future.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swapbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`capbyhjm` | `cfbyhjm` | `floorbyhjm` | `hjmtree` | `treeviewer`

Topics

“Computing Instrument Prices” on page 2-81
“Swap” on page 2-13
“Understanding Interest-Rate Tree Models” on page 2-66
“Pricing Options Structure” on page A-2
“Supported Interest-Rate Instrument Functions” on page 2-3

swapbyhw

Price swap instrument from Hull-White interest-rate tree

Syntax

```
[Price,PriceTree,CFTree,SwapRate] = swapbyhw(HWTree,LegRate,Settle,Maturity)
[Price,PriceTree,CFTree,SwapRate] = swapbyhw( ___ ,Name,Value)
```

Description

`[Price,PriceTree,CFTree,SwapRate] = swapbyhw(HWTree,LegRate,Settle,Maturity)` prices a swap instrument from a Hull-White interest-rate tree. `swapbyhw` computes prices of vanilla swaps, amortizing swaps and forward swaps.

Note Alternatively, you can use the `Swap` object to price swap instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree,CFTree,SwapRate] = swapbyhw( ___ ,Name,Value)` adds additional name-value pair arguments.

Examples

Price an Interest-Rate Swap

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 = datetime(2005,1,1);
Maturity = datetime(2008,1,1);
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)
```

```
Warning: Swaps are valued at Tree ValuationDate rather than Settle

Price = 5.9109

PriceTree = struct with fields:
     FinObj: 'HWPriceTree'
      PTree: {[5.9109]  [-1.2692 3.0317 7.5253]  [-5.1049 -2.1588 0.8799 4.0142 7.2471]  [-3.3462
       tObs: [0 1 2 3 4]
    Connect: {[2]  [2 3 4]  [2 2 3 4 4]}
      Probs: {[3×1 double]  [3×3 double]  [3×5 double]}


SwapRate = NaN
```

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)
```

```
Warning: Swaps are valued at Tree ValuationDate rather than Settle

Price = 1.4211e-14

PriceTree = struct with fields:
     FinObj: 'HWPriceTree'
      PTree: {[1.4211e-14]  [-5.5941 -1.4265 2.9289]  [-7.9659 -5.0883 -2.1199 0.9423 4.1011]  [
       tObs: [0 1 2 3 4]
    Connect: {[2]  [2 3 4]  [2 2 3 4 4]}
      Probs: {[3×1 double]  [3×3 double]  [3×5 double]}


SwapRate = 0.0438
```

Price an Amortizing Swap

Price an amortizing swap using the Principal input argument to define the amortization schedule.

Create the RateSpec.

```
Rates = 0.035;
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = datetime(2017,1,1);
Compounding = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.8135
            Rates: 0.0350
         EndTimes: 6
       StartTimes: 0
         EndDates: 736696
       StartDates: 734504
    ValuationDate: 734504
            Basis: 0
     EndMonthRule: 1
```

Create the swap instrument using the following data:

```
Settle = datetime(2011,1,1);
Maturity = datetime(2017,1,1);
Period = 1;
LegRate = [0.04 10];
```

Define the swap amortizing schedule.

```
Principal ={{datetime(2013,1,1) 100;datetime(2014,1,1) 80;datetime(2015,1,1) 60;datetime(2016,1,1
```

Build the HW tree using the following data:

```
VolDates = [datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; d
VolCurve = 0.1;
AlphaDates = datetime(2017,1,1);
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RateSpec, HWTimeSpec);
```

Compute the price of the amortizing swap.

```
Price = swapbyhw(HWT, LegRate, Settle, Maturity, 'Principal', Principal)
```

```
Price = 1.4574
```

Price a Forward Swap

Price a forward swap using the `StartDate` input argument to define the future starting date of the swap.

Create the `RateSpec`.

```
Rates = 0.0374;
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = datetime(2018,1,1);
Compounding = 1;
```

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.8023
            Rates: 0.0374
         EndTimes: 6
       StartTimes: 0
         EndDates: 737061
       StartDates: 734869
    ValuationDate: 734869
            Basis: 0
     EndMonthRule: 1
```

Build an HW tree.

```
VolDates = [datetime(2013,1,1) ; datetime(2014,1,1) ; datetime(2015,1,1) ; datetime(2016,1,1) ; d
VolCurve = 0.1;
AlphaDates = datetime(2018,1,1);
AlphaCurve = 0.1;

HWVolSpec = hwvolspec(RateSpec.ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);
HWTimeSpec = hwtimespec(RateSpec.ValuationDate, VolDates, Compounding);
HWT = hwtree(HWVolSpec, RateSpec, HWTimeSpec);
```

Compute the price of a forward swap that starts in a year (Jan 1, 2013) and matures in four years with a forward swap rate of 4.25%.

```
Settle = datetime(2012,1,1);
Maturity = datetime(2017,1,1);
StartDate = datetime(2013,1,1);
LegRate = [0.0425 10];

Price = swapbyhw(HWT, LegRate, Settle, Maturity, 'StartDate', StartDate)
```

```
Price = 1.4434
```

Using the previous data, compute the forward swap rate, the coupon rate for the fixed leg, such that the forward swap price at time = 0 is zero.

```
LegRate = [NaN 10];
[Price, ~,SwapRate] = swapbyhw(HWT, LegRate, Settle, Maturity, 'StartDate', StartDate)
```

```
Price = 4.2633e-14
```

```
SwapRate = 0.0384
```

Input Arguments

HWTree — Interest-rate structure

structure

Interest-rate tree structure, created by `hwtree`

Data Types: `struct`

`LegRate` — Leg rate
matrix

Leg rate, specified as a `NINST`-by-2 matrix, with each row defined as one of the following:

- `[CouponRate Spread]` (fixed-float)
- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

`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.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `swapbyhw` also accepts serial date numbers as inputs, but they are not recommended.

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

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each swap.

To support existing code, `swapbyhw` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree,CFTree,SwapRate] = swapbyhw(HWTree,LegRate,Settle,Maturity,LegReset,Basis,Principal,LegType)`

`LegReset` — Reset frequency per year for each swap
`[1 1]` (default) | vector

Reset frequency per year for each swap, specified as the comma-separated pair consisting of `'LegReset'` and a `NINST`-by-2 vector.

Data Types: double

Basis — Day-count basis representing the basis for each leg
0 (actual/actual) (default) | integer from 0 to 13

Day-count basis representing the basis for each leg, specified as the comma-separated pair consisting of 'Basis' and a NINST-by-1 array (or NINST-by-2 if Basis is different for each leg).

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

For more information, see “Basis” on page 2-228.

Data Types: double

Principal — Notional principal amounts or principal value schedules
100 (default) | vector or cell array

Notional principal amounts or principal value schedules, specified as the comma-separated pair consisting of 'Principal' and a vector or cell array.

Principal accepts a NINST-by-1 vector or NINST-by-1 cell array (or NINST-by-2 if Principal is different for each leg) of the notional principal amounts or principal value schedules. For schedules, each element of the cell array is a NumDates-by-2 array where the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: cell | double

LegType — Leg type
[1 0] for each instrument (default) | matrix with values [1 1] (fixed-fixed), [1 0] (fixed-float), [0 1] (float-fixed), or [0 0] (float-float)

Leg type, specified as the comma-separated pair consisting of 'LegType' and a NINST-by-2 matrix with values [1 1] (fixed-fixed), [1 0] (fixed-float), [0 1] (float-fixed), or [0 0] (float-float). 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. LegType allows [1 1] (fixed-fixed), [1 0] (fixed-float), [0 1] (float-fixed), or [0 0] (float-float) swaps

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` (or `NINST`-by-`2` if `EndMonthRule` is different for each leg).

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` (or `NINST`-by-`2` if `AdjustCashFlowsBasis` is different for each leg) of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-`2` if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.

- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`StartDate` — Date swap actually starts
`Settle` date (default) | datetime array | string array | date character vector

Date swap actually starts, specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swapbyhw` also accepts serial date numbers as inputs, but they are not recommended.

Use this argument to price forward swaps, that is, swaps that start in a future date

Output Arguments

`Price` — Expected swap prices at time 0
vector

Expected swap prices at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

`CFTree` — Swap cash flows
structure

Swap cash flows, returned as 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 — Rates applicable to fixed leg

matrix

Rates applicable to the fixed leg, returned as 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.

More About

Amortizing Swap

In an amortizing swap, the notional principal decreases periodically because it is tied to an underlying financial instrument with a declining (amortizing) principal balance, such as a mortgage.

Forward Swap

Agreement to enter into an interest-rate swap arrangement on a fixed date in future.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although swapbyhw supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

bondbyhw | capbyhw | cfbyhw | floorbyhw | fixedbyhw | hwtree | Swap

Topics

"Pricing Using Interest-Rate Tree Models" on page 2-81
"Calibrating Hull-White Model Using Market Data" on page 2-92
"Price a Swaption Using SABR Model and Analytic Pricer" on page 2-185
"Swap" on page 2-13
"Understanding Interest-Rate Tree Models" on page 2-66
"Pricing Options Structure" on page A-2
"Supported Interest-Rate Instrument Functions" on page 2-3

“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

swapbyzero

Price swap instrument from set of zero curves and price cross-currency swaps

Syntax

```
[Price,SwapRate,AI,RecCF,RecCFDates,PayCF,PayCFDates] = swapbyzero(RateSpec,
LegRate,Settle,Maturity)
[Price,SwapRate,AI,RecCF,RecCFDates,PayCF,PayCFDates] = swapbyzero(RateSpec,
LegRate,Settle,Maturity,Name,Value)
```

Description

`[Price,SwapRate,AI,RecCF,RecCFDates,PayCF,PayCFDates] = swapbyzero(RateSpec, LegRate,Settle,Maturity)` prices a swap instrument. You can use `swapbyzero` to compute prices of vanilla swaps, amortizing swaps, and forward swaps. All inputs are either scalars or `NINST`-by-`1` vectors unless otherwise specified. Any date can be a date character vector. An optional argument can be passed as an empty matrix `[]`.

Note Alternatively, you can use the `Swap` object to price swap instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,SwapRate,AI,RecCF,RecCFDates,PayCF,PayCFDates] = swapbyzero(RateSpec, LegRate,Settle,Maturity,Name,Value)` prices a swap instrument with additional options specified by one or more `Name,Value` pair arguments. You can use `swapbyzero` to compute prices of vanilla swaps, amortizing swaps, forward swaps, and cross-currency swaps. For more information on the name-value pairs for vanilla swaps, amortizing swaps, and forward swaps, see Vanilla Swaps, Amortizing Swaps, Forward Swaps on page 11-0 .

Specifically, you can use name-value pairs for `FXRate`, `ExchangeInitialPrincipal`, and `ExchangeMaturityPrincipal` to compute the price for cross-currency swaps. For more information on the name-value pairs for cross-currency swaps, see Cross-Currency Swaps on page 11-0 .

Examples

Price an Interest-Rate Swap

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 = datetime(2000,1,1);
Maturity = datetime(2003,1,1);
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
```

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
```

In swapbyzero , if Settle is not on a reset date (and 'StartDate' is not specified), the effective date is assumed to be the previous reset date before Settle in order to compute the accrued interest and dirty price. In this example, the effective date is ('15-Sep-2009'), which is the previous reset date before the ('08-Jun-2010') Settle date.

Use swapbyzero with name-value pair arguments for LegRate, LegType, LatestFloatingRate, AdjustCashFlowsBasis, and BusinessDayConvention to calculate output for Price, SwapRate, AI, RecCF, RecCFDates, PayCF, and PayCFDates:

```
Settle = datetime(2008,6,1);
RateSpec = intenvset('Rates', [.005 .0075 .01 .014 .02 .025 .03]',...
'StartDates',Settle, 'EndDates',[datetime(2010,12,8) , datetime(2011,6,8) , datetime(2012,6,8) ,
Maturity = datetime(2020,9,15);
LegRate = [.025 50];
LegType = [1 0]; % fixed/floating
LatestFloatingRate = .005;

[Price, SwapRate, AI, RecCF, RecCFDates, PayCF,PayCFDates] = ...
swapbyzero(RateSpec, LegRate, Settle, Maturity,'LegType',LegType,...
'LatestFloatingRate',LatestFloatingRate,'AdjustCashFlowsBasis',true,...
'BusinessDayConvention','modifiedfollow')
```

```
Price = -7.7485
```

```
SwapRate = NaN

AI = 1.4098

RecCF = 1×14

   -1.7623    2.4863    2.5000    2.5000    2.5000    2.5137    2.4932    2.4932    2.5000    2.5

RecCFDates = 1×14

      733560      733666      734031      734396      734761      735129      735493      735857

PayCF = 1×14

   -0.3525    0.4973    1.0006    1.0006    2.0510    2.5944    3.6040    3.8939    4.4152    4.6

PayCFDates = 1×14

      733560      733666      734031      734396      734761      735129      735493      735857
```

Price Swaps By Specifying Multiple Term Structures Using RateSpec

Price three swaps using two interest-rate curves. First, define the data for the interest-rate term structure:

```
StartDates = datetime(2012,5,1);
EndDates = [datetime(2013,5,1) ; datetime(2014,5,1) ; datetime(2015,5,1) ; datetime(2016,5,1)];
Rates = [[0.0356;0.041185;0.04489;0.047741],[0.0366;0.04218;0.04589;0.04974]];
```

Create the `RateSpec` using `intenvset`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
            Disc: [4x2 double]
           Rates: [4x2 double]
         EndTimes: [4x1 double]
       StartTimes: [4x1 double]
         EndDates: [4x1 double]
       StartDates: 734990
    ValuationDate: 734990
            Basis: 0
     EndMonthRule: 1
```

Look at the `Rates` for the two interest-rate curves.

```
RateSpec.Rates
```

```
ans = 4×2
```

```
    0.0356    0.0366
    0.0412    0.0422
    0.0449    0.0459
    0.0477    0.0497
```

Define the swap instruments.

```
Settle = datetime(2012,5,1);
Maturity = datetime(2015,5,1);
LegRate = [0.06 10];
Principal = [100;50;100];  % Three notional amounts
```

Price three swaps using two curves.

```
Price = swapbyzero(RateSpec, LegRate, Settle, Maturity, 'Principal', Principal)
```

```
Price = 3×2

    3.9688    3.6869
    1.9844    1.8434
    3.9688    3.6869
```

Price Swap By Specifying Multiple Term Structures Using a 1-by-2 RateSpec

Price a swap using two interest-rate curves. First, define data for the two interest-rate term structures:

```
StartDates = datetime(2012,5,1);
EndDates = [datetime(2013,5,1) ; datetime(2014,5,1) ; datetime(2015,5,1) ; datetime(2016,5,1)];
Rates1 = [0.0356;0.041185;0.04489;0.047741];
Rates2 = [0.0366;0.04218;0.04589;0.04974];
```

Create the `RateSpec` using `intenvset`.

```
RateSpecReceiving = intenvset('Rates', Rates1, 'StartDates',StartDates,...
'EndDates', EndDates, 'Compounding', 1);
RateSpecPaying= intenvset('Rates', Rates2, 'StartDates',StartDates,...
'EndDates', EndDates, 'Compounding', 1);
RateSpec=[RateSpecReceiving RateSpecPaying]
```

```
RateSpec=1×2 struct array with fields:
    FinObj
    Compounding
    Disc
    Rates
    EndTimes
    StartTimes
    EndDates
    StartDates
    ValuationDate
    Basis
    EndMonthRule
```

Define the swap instruments.

```
Settle = datetime(2012,5,1);
Maturity = datetime(2015,5,1);
LegRate = [0.06 10];
Principal = [100;50;100];
```

Price three swaps using the two curves.

```
Price = swapbyzero(RateSpec, LegRate, Settle, Maturity, 'Principal', Principal)
```

```
Price = 3×1

    3.9693
    1.9846
    3.9693
```

Compute a Forward Par Swap Rate

To compute a forward par swap rate, set the `StartDate` parameter to a future date and set the fixed coupon rate in the `LegRate` input to `NaN`.

Define the zero curve data and build a zero curve using `IRDataCurve`.

```
ZeroRates = [2.09 2.47 2.71 3.12 3.43 3.85 4.57]'/100;
Settle = datetime(2012,1,1);
EndDates = datemnth(Settle,12*[1 2 3 5 7 10 20]');
Compounding = 1;

ZeroCurve = IRDataCurve('Zero',Settle,EndDates,ZeroRates,'Compounding',Compounding)
```

```
ZeroCurve =
             Type: Zero
           Settle: 734869 (01-Jan-2012)
      Compounding: 1
            Basis: 0 (actual/actual)
     InterpMethod: linear
            Dates: [7x1 double]
             Data: [7x1 double]
```

Create a `RateSpec` structure using the `toRateSpec` method.

```
RateSpec = ZeroCurve.toRateSpec(EndDates)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [7x1 double]
            Rates: [7x1 double]
         EndTimes: [7x1 double]
       StartTimes: [7x1 double]
         EndDates: [7x1 double]
       StartDates: 734869
    ValuationDate: 734869
```

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

Compute the forward swap rate (the coupon rate for the fixed leg), such that the forward swap price at time = 0 is zero. The forward swap starts in a month (1-Feb-2012) and matures in 10 years (1-Feb-2022).

```
StartDate = datetime(2012,2,1);
Maturity = datetime(2022,2,1);
LegRate = [NaN 0];

[Price, SwapRate] = swapbyzero(RateSpec, LegRate, Settle, Maturity,...
'StartDate', StartDate)
```

```
Price = 0

SwapRate = 0.0378
```

Compute a Forward Swap Rate Using the Optional Input BusinessDayConvention

The swapbyzero function generates the cash flow dates based on the Settle and Maturity dates, while using the Maturity date as the "anchor" date from which to count backwards in regular intervals. By default, swapbyzero does not distinguish non-business days from business days. To make swapbyzero move non-business days to the following business days, you can you can set the optional name-value input argument BusinessDayConvention with a value of follow.

Define the zero curve data and build a zero curve using IRDataCurve.

```
ZeroRates = [2.09 2.47 2.71 3.12 3.43 3.85 4.57]'/100;
Settle = datetime(2012,1,5);
EndDates = datemnth(Settle,12*[1 2 3 5 7 10 20]');
Compounding = 1;
ZeroCurve = IRDataCurve('Zero',Settle,EndDates,ZeroRates,'Compounding',Compounding);
RateSpec = ZeroCurve.toRateSpec(EndDates);
StartDate = datetime(2012,2,5);
Maturity = datetime(2022,2,5);
LegRate = [NaN 0];
```

To demonstrate the optional input BusinessDayConvention, swapbyzero is first used without and then with the optional name-value input argument BusinessDayConvention. Notice that when using BusinessDayConvention, all days are business days.

```
[Price1,SwapRate1,~,~,RecCFDates1,~,PayCFDates1] = swapbyzero(RateSpec,LegRate,Settle,Maturity,.
    'StartDate',StartDate);
datestr(RecCFDates1)
```

```
ans = 11x11 char array
    '05-Jan-2012'
    '05-Feb-2013'
    '05-Feb-2014'
    '05-Feb-2015'
    '05-Feb-2016'
    '05-Feb-2017'
    '05-Feb-2018'
```

```
    '05-Feb-2019'
    '05-Feb-2020'
    '05-Feb-2021'
    '05-Feb-2022'
```

```
isbusday(RecCFDates1)
```

```
ans = 11x1 logical array

   1
   1
   1
   1
   1
   0
   1
   1
   1
   1
      ⋮
```

```
[Price2,SwapRate2,~,~,RecCFDates2,~,PayCFDates2] = swapbyzero(RateSpec,LegRate,Settle,Maturity,.
    'StartDate',StartDate,'BusinessDayConvention','follow');
datestr(RecCFDates2)
```

```
ans = 12x11 char array
    '05-Jan-2012'
    '06-Feb-2012'
    '05-Feb-2013'
    '05-Feb-2014'
    '05-Feb-2015'
    '05-Feb-2016'
    '06-Feb-2017'
    '05-Feb-2018'
    '05-Feb-2019'
    '05-Feb-2020'
    '05-Feb-2021'
    '07-Feb-2022'
```

```
isbusday(RecCFDates2)
```

```
ans = 12x1 logical array

   1
   1
   1
   1
   1
   1
   1
   1
   1
   1
      ⋮
```

Price an Amortizing Swap

Price an amortizing swap using the `Principal` input argument to define the amortization schedule.

Create the `RateSpec`.

```
Rates = 0.035;
ValuationDate = datetime(2011,1,1);
StartDates = ValuationDate;
EndDates = datetime(2017,1,1);
Compounding = 1;

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

Create the swap instrument using the following data:

```
Settle = datetime(2011,1,1);
Maturity = datetime(2017,1,1);
LegRate = [0.04 10];
```

Define the swap amortizing schedule.

```
Principal ={{datetime(2013,1,1) 100;datetime(2014,1,1) 80;datetime(2015,1,1) 60;datetime(2016,1,1
```

Compute the price of the amortizing swap.

```
Price = swapbyzero(RateSpec, LegRate, Settle, Maturity, 'Principal' , Principal)
```

```
Price = 1.4574
```

Price a Forward Swap

Price a forward swap using the `StartDate` input argument to define the future starting date of the swap.

Create the `RateSpec`.

```
Rates = 0.0325;
ValuationDate = datetime(2012,1,1);
StartDates = ValuationDate;
EndDates = datetime(2018,1,1);
Compounding = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: 0.8254
            Rates: 0.0325
         EndTimes: 6
```

```
       StartTimes: 0
         EndDates: 737061
       StartDates: 734869
    ValuationDate: 734869
            Basis: 0
     EndMonthRule: 1
```

Compute the price of a forward swap that starts in a year (Jan 1, 2013) and matures in three years with a forward swap rate of 4.27%.

```
Settle = datetime(2012,1,1);
StartDate = datetime(2013,1,1);
Maturity = datetime(2016,1,1);
LegRate = [0.0427 10];

Price = swapbyzero(RateSpec, LegRate, Settle, Maturity, 'StartDate' , StartDate)

Price = 2.5083
```

Using the previous data, compute the forward swap rate, the coupon rate for the fixed leg, such that the forward swap price at time = 0 is zero.

```
LegRate = [NaN 10];
[Price, SwapRate] = swapbyzero(RateSpec, LegRate, Settle, Maturity,...
'StartDate' , StartDate)

Price = 0

SwapRate = 0.0335
```

Specify the Rate at the Instrument's Starting Date When It Cannot Be Obtained from the RateSpec

If `Settle` is not on a reset date of a floating-rate note, `swapbyzero` attempts to obtain the latest floating rate before `Settle` from `RateSpec` or the `LatestFloatingRate` parameter. When the reset date for this rate is out of the range of `RateSpec` (and `LatestFloatingRate` is not specified), `swapbyzero` fails to obtain the rate for that date and generates an error. This example shows how to use the `LatestFloatingRate` input parameter to avoid the error.

Create the error condition when a swap instrument's `StartDate` cannot be determined from the `RateSpec`.

```
Settle = datetime(2000,1,1);
Maturity = datetime(2003,12,1);
Basis = 0;
Principal = 100;
LegRate = [0.06 20]; % [CouponRate Spread]
LegType = [1 0]; % [Fixed Float]
LegReset = [1 1]; % Payments once per year

load deriv.mat;

Price = swapbyzero(ZeroRateSpec,LegRate,Settle,Maturity,...
'LegReset',LegReset,'Basis',Basis,'Principal',Principal, ...
'LegType',LegType)

Error using floatbyzero (line 256)
The rate at the instrument starting date cannot be obtained from RateSpec.
```

```
 Its reset date (01-Dec-1999) is out of the range of dates contained in RateSpec.
 This rate is required to calculate cash flows at the instrument starting date.
 Consider specifying this rate with the 'LatestFloatingRate' input parameter.

Error in swapbyzero (line 289)
[FloatFullPrice, FloatPrice,FloatCF,FloatCFDates] = floatbyzero(FloatRateSpec, Spreads, Settle,...
```

Here, the reset date for the rate at `Settle` was `01-Dec-1999`, which was earlier than the valuation date of `ZeroRateSpec` (`01-Jan-2000`). This error can be avoided by specifying the rate at the swap instrument's starting date using the `LatestFloatingRate` input parameter.

Define `LatestFloatingRate` and calculate the floating-rate price.

```
Price = swapbyzero(ZeroRateSpec,LegRate,Settle,Maturity,...
'LegReset',LegReset,'Basis',Basis,'Principal',Principal, ...
'LegType',LegType,'LatestFloatingRate',0.03)
```

```
Price =

    4.7594
```

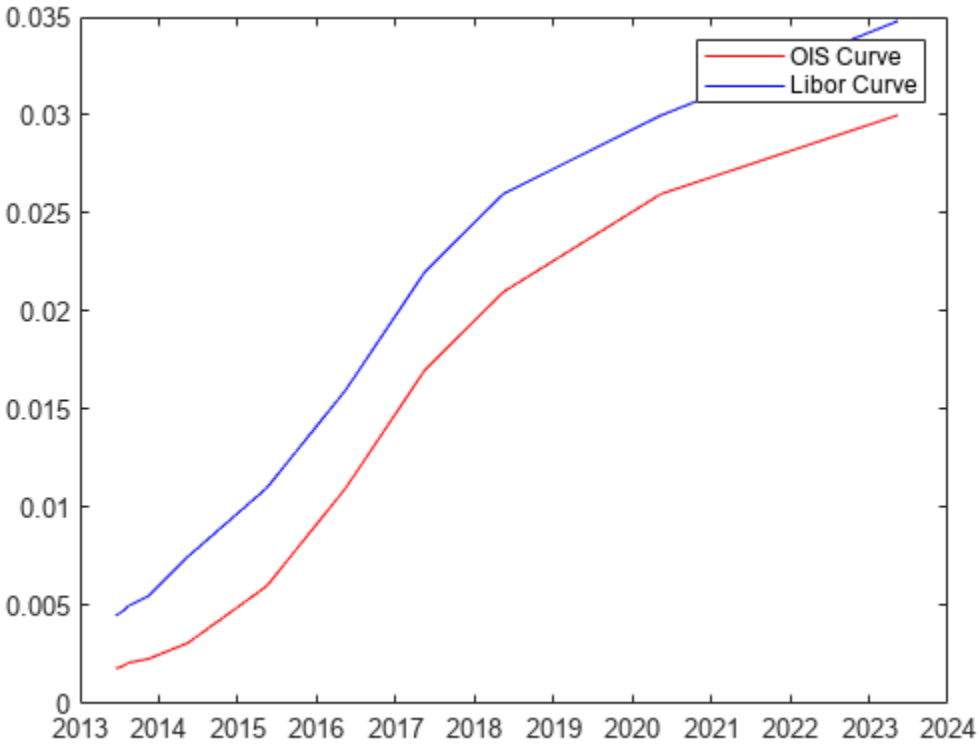
Price a Swap Using a Different Curve to Generate the Cash Flows of the Floating Leg

Define the OIS and Libor rates.

```
Settle = datetime(2013,5,15);
CurveDates = daysadd(Settle,360*[1/12 2/12 3/12 6/12 1 2 3 4 5 7 10],1);
OISRates = [.0018 .0019 .0021 .0023 .0031 .006  .011 .017 .021 .026 .03]';
LiborRates = [.0045 .0047 .005 .0055 .0075 .011 .016 .022 .026 .030 .0348]';
```

Plot the dual curves.

```
figure,plot(CurveDates,OISRates,'r');hold on;plot(CurveDates,LiborRates,'b')
datetick
legend({'OIS Curve', 'Libor Curve'})
```



Create an associated `RateSpec` for the OIS and Libor curves.

```
OISCurve = intenvset('Rates',OISRates,'StartDate',Settle,'EndDates',CurveDates);
LiborCurve = intenvset('Rates',LiborRates,'StartDate',Settle,'EndDates',CurveDates);
```

Define the swap.

```
Maturity = datetime(2018,5,15); % Five year swap
FloatSpread = 0;
FixedRate = .025;
LegRate = [FixedRate FloatSpread];
```

Compute the price of the swap instrument. The `LiborCurve` term structure will be used to generate the cash flows of the floating leg. The `OISCurve` term structure will be used for discounting the cash flows.

```
Price = swapbyzero(OISCurve, LegRate, Settle,...
Maturity,'ProjectionCurve',LiborCurve)
```

```
Price = -0.3697
```

Compare results when the term structure `OISCurve` is used both for discounting and also generating the cash flows of the floating leg.

```
PriceSwap = swapbyzero(OISCurve, LegRate, Settle, Maturity)
```

```
PriceSwap = 2.0517
```

Price a Fixed-Fixed Currency Swap

Price an existing cross currency swap that receives a fixed rate of JPY and pays a fixed rate of USD at an annual frequency.

```
Settle = datetime(2015,8,15);
Maturity = datetime(2018,8,15);
Reset = 1;
LegType = [1 1]; % Fixed-Fixed

r_USD = .09;
r_JPY = .04;

FixedRate_USD = .08;
FixedRate_JPY = .05;

Principal_USD = 10000000;
Principal_JPY = 1200000000;

S = 1/110;

RateSpec_USD = intenvset('StartDate',Settle,'EndDate', Maturity,'Rates',r_USD,'Compounding',-1);
RateSpec_JPY = intenvset('StartDate',Settle,'EndDate', Maturity,'Rates', r_JPY,'Compounding',-1)

Price = swapbyzero([RateSpec_JPY RateSpec_USD], [FixedRate_JPY FixedRate_USD],...
Settle, Maturity,'Principal',[Principal_JPY Principal_USD],'FXRate',[S 1], 'LegType',LegType)
```

```
Price = 1.5430e+06
```

Price a Float-Float Currency Swap

Price a new swap where you pay a EUR float and receive a USD float.

```
Settle = datetime(2015,12,22);
Maturity = datetime(2018,8,15);
LegRate = [0 -50/10000];
LegType = [0 0]; % Float Float
LegReset = [4 4];
FXRate = 1.1;
Notional = [10000000 8000000];

USD_Dates = datemnth(Settle,[1 3 6 12*[1 2 3 5 7 10 20 30]]');
USD_Zero = [0.03 0.06 0.08 0.13 0.36 0.76 1.63 2.29 2.88 3.64 3.89]'/100;
Curve_USD = intenvset('StartDate',Settle,'EndDates',USD_Dates,'Rates',USD_Zero);

EUR_Dates = datemnth(Settle,[3 6 12*[1 2 3 5 7 10 20 30]]');
EUR_Zero = [0.017 0.033 0.088 .27 .512 1.056 1.573 2.183 2.898 2.797]'/100;
Curve_EUR = intenvset('StartDate',Settle,'EndDates',EUR_Dates,'Rates',EUR_Zero);

Price = swapbyzero([Curve_USD Curve_EUR], ...
    LegRate, Settle, Maturity,'LegType',LegType,'LegReset',LegReset,'Principal',Notional,...
    'FXRate',[1 FXRate],'ExchangeInitialPrincipal',false)
```

```
Price = 1.2002e+06
```

Input Arguments

`RateSpec` — Interest-rate structure

structure

Interest-rate structure, specified using `intenvset` to create a `RateSpec`.

`RateSpec` can also be a 1-by-2 input variable of `RateSpecs`, with the second `RateSpec` structure containing one or more discount curves for the paying leg. If only one `RateSpec` structure is specified, then this `RateSpec` is used to discount both legs.

Data Types: `struct`

`LegRate` — Leg rate

matrix

Leg rate, specified as a `NINST`-by-2 matrix, with each row defined as one of the following:

- `[CouponRate Spread]` (fixed-float)
- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

`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.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified either as a scalar or `NINST`-by-1 vector using a datetime array, string array, or date character vectors with the same value which represents the settlement date for each swap. `Settle` must be earlier than `Maturity`.

To support existing code, `swapbyzero` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors representing the maturity date for each swap.

To support existing code, `swapbyzero` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,SwapRate,AI,RecCF,RecCFDates,PayCF,PayCFDates] = swapbyzero(RateSpec,LegRate,Settle, Maturity,'LegType',LegType,'LatestFloatingRate',LatestFloatingRate,'AdjustCashFlowsBasis',true, 'BusinessDayConvention','modifiedfollow')`

Vanilla Swaps, Amortizing Swaps, Forward Swaps

`LegReset` — Reset frequency per year for each swap

`[1 1]` (default) | vector

Reset frequency per year for each swap, specified as the comma-separated pair consisting of `'LegReset'` and a `NINST`-by-2 vector.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis for each leg, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-1 array (or `NINST`-by-2 if `Basis` is different for each leg).

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amounts or principal value schedules

`100` (default) | vector or cell array

Notional principal amounts or principal value schedules, specified as the comma-separated pair consisting of `'Principal'` and a vector or cell array.

`Principal` accepts a `NINST`-by-1 vector or `NINST`-by-1 cell array (or `NINST`-by-2 if `Principal` is different for each leg) of the notional principal amounts or principal value schedules. For schedules,

each element of the cell array is a `NumDates`-by-2 array where the first column is dates and the second column is its associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `cell` | `double`

`LegType` — Leg type
`[1 0]` for each instrument (default) | matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float)

Leg type, specified as the comma-separated pair consisting of `'LegType'` and a `NINST`-by-2 matrix with values `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float). 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`. `LegType` allows `[1 1]` (fixed-fixed), `[1 0]` (fixed-float), `[0 1]` (float-fixed), or `[0 0]` (float-float) swaps

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [`0`, `1`] using a `NINST`-by-`1` (or `NINST`-by-2 if `EndMonthRule` is different for each leg).

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Flag to adjust cash flows based on actual period day count
`false` (default) | value of `0` (false) or `1` (true)

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'AdjustCashFlowsBasis'` and a `NINST`-by-`1` (or `NINST`-by-2 if `AdjustCashFlowsBasis` is different for each leg) of logicals with values of `0` (false) or `1` (true).

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` (or `NINST`-by-2 if `BusinessDayConvention` is different for each leg) cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.

- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Holidays` — Holidays used in computing business days
if not specified, the default is to use `holidays.m` (default) | MATLAB dates

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and MATLAB dates using a `NHolidays`-by-`1` vector.

Data Types: `datetime`

`StartDate` — Dates when swaps actually start
If not specified, date is `Settle` (default) | character vector | cell array of character vectors

Dates when the swaps actually start, specified as the comma-separated pair consisting of `'StartDate'` and a `NINST`-by-`1` vector of character vectors or cell array of character vectors.

To support existing code, `swapbyzero` also accepts serial date numbers as inputs, but they are not recommended.

`LatestFloatingRate` — Rate for the next floating payment
If not specified, then `RateSpec` must contain this information (default) | scalar numeric

Rate for the next floating payment, set at the last reset date, specified as the comma-separated pair consisting of `'LatestFloatingRate'` and a scalar numeric value.

`LatestFloatingRate` accepts a Rate for the next floating payment, set at the last reset date. `LatestFloatingRate` is a `NINST`-by-`1` (or `NINST`-by-`2` if `LatestFloatingRate` is different for each leg).

Data Types: `double`

`ProjectionCurve` — Rate curve used in generating cash flows for the floating leg of the swap
if `ProjectionCurve` is not specified, then `RateSpec` is used both for discounting and generating cash flows for the floating leg (default) | `RateSpec` or vector

Rate curve used in generating cash flows for the floating leg of the swap, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a `RateSpec`.

If specifying a fixed-float or a float-fixed swap, the `ProjectionCurve` rate curve is used in generating cash flows for the floating leg of the swap. This structure must be created using `intenvset`.

If specifying a fixed-fixed or a float-float swap, then `ProjectionCurve` is `NINST`-by-`2` vector because each floating leg could have a different projection curve.

Data Types: `struct`

Cross-Currency Swaps

`FXRate` — Foreign exchange (FX) rate applied to cash flows
if not specified, both legs of `swapbyzero` are in same currency (default) | array

Foreign exchange (FX) rate applied to cash flows, specified as the comma-separated pair consisting of `'FXRate'` and a `NINST`-by-2 array of doubles. Since the foreign exchange rate could be applied to either the payer or receiver leg, there are 2 columns in the input array and you must specify which leg has the foreign currency.

Data Types: `double`

`ExchangeInitialPrincipal` — Flag to indicate if initial `Principal` is exchanged
`0` (false) (default) | array

Flag to indicate if initial `Principal` is exchanged, specified as the comma-separated pair consisting of `'ExchangeInitialPrincipal'` and a `NINST`-by-`1` array of logicals.

Data Types: `logical`

`ExchangeMaturityPrincipal` — Flag to indicate if `Principal` exchanged at `Maturity`
`1` (true) (default) | array

Flag to indicate if `Principal` is exchanged at `Maturity`, specified as the comma-separated pair consisting of `'ExchangeMaturityPrincipal'` and a `NINST`-by-`1` array of logicals. While in practice most single currency swaps do not exchange principal at maturity, the default is true to maintain backward compatibility.

Data Types: `logical`

Output Arguments

`Price` — Swap prices
matrix

Swap prices, returned as the number of instruments (`NINST`) by number of curves (`NUMCURVES`) matrix. Each column arises from one of the zero curves. `Price` output is the dirty price. To compute the clean price, subtract the accrued interest (`AI`) from the dirty price.

`SwapRate` — Rates applicable to fixed leg
matrix

Rates applicable to the fixed leg, returned as a `NINST`-by-`NUMCURVES` matrix 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`.

`AI` — Accrued interest
matrix

Accrued interest, returned as a `NINST`-by-`NUMCURVES` matrix.

`RecCF` — Cash flows for receiving leg
matrix

Cash flows for the receiving leg, returned as a `NINST`-by-`NUMCURVES` matrix.

Note If there is more than one curve specified in the `RateSpec` input, then the first `NCURVES` row corresponds to the first swap, the second `NCURVES` row correspond to the second swap, and so on.

`RecCFDates` — Payment dates for receiving leg
matrix

Payment dates for the receiving leg, returned as an `NINST`-by-`NUMCURVES` matrix.

`PayCF` — Cash flows for paying leg
matrix

Cash flows for the paying leg, returned as an `NINST`-by-`NUMCURVES` matrix.

`PayCFDates` — Payment dates for paying leg
matrix

Payment dates for the paying leg, returned as an `NINST`-by-`NUMCURVES` matrix.

More About

Amortizing Swap

In an amortizing swap, the notional principal decreases periodically because it is tied to an underlying financial instrument with a declining (amortizing) principal balance, such as a mortgage.

Forward Swap

Agreement to enter into an interest-rate swap arrangement on a fixed date in future.

Cross-currency Swap

Swaps where the payment legs of the swap are denominated in different currencies.

One difference between cross-currency swaps and standard swaps is that an exchange of principal may occur at the beginning and/or end of the swap. The exchange of initial principal will only come into play in pricing a cross-currency swap at inception (in other words, pricing an existing cross-currency swap will occur after this cash flow has happened). Furthermore, these exchanges of principal typically do not affect the value of the swap (since the principal values of the two legs are chosen based on the currency exchange rate) but affect the cash flows for each leg.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swapbyzero` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, J. *Options, Futures and Other Derivatives* Fourth Edition. Prentice Hall, 2000.

See Also

`bondbyzero` | `intenvset` | `cfbyzero` | `fixedbyzero` | `floatbyzero` | `Swap`

Topics

“Pricing Using Interest-Rate Term Structure” on page 2-61
“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Swap” on page 2-13
“Understanding the Interest-Rate Term Structure” on page 2-48
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

swaptionbybdt

Price swaption from Black-Derman-Toy interest-rate tree

Syntax

```
[Price,PriceTree] = swaptionbybdt(BDTTree,OptSpec,Strike,ExerciseDates,
Spread,Settle,Maturity)
[Price,PriceTree] = swaptionbybdt( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = swaptionbybdt(BDTTree,OptSpec,Strike,ExerciseDates, Spread,Settle,Maturity)` prices swaption using a Black-Derman-Toy tree.

Note Alternatively, you can use the `Swaption` object to price swaption instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = swaptionbybdt( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a 5-Year Call Swaption Using a BDT Interest-Rate Tree

This example shows how to 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 = [datetime(2007,1,1) ; datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1)

EndDates = [datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1) ; datetime(2011,1,1) ; d

ValuationDate = datetime(2007,1,1);
Compounding = 1;

% define the RateSpec
RateSpec = intenvset('Rates', Rates, 'StartDates', StartDates, 'EndDates', EndDates, ...
'Compounding', Compounding);

% use VolSpec to compute 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 a BDT tree
TimeSpec = bdttimespec(ValuationDate, EndDates, Compounding);
```

```
% build the BDT tree
BDTTree = bdttree(VolSpec, RateSpec, TimeSpec);

% use the following swaption arguments
ExerciseDates = datetime(2012,1,1);
SwapSettlement = ExerciseDates;
SwapMaturity   = datetime(2015,1,1);
Spread = 0;
SwapReset = 1;
Principal = 100;
OptSpec = 'call';
Strike =.062;
Basis = 1;

% price the swaption
[Price, PriceTree] = swaptionbybdt(BDTTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity, 'SwapReset', SwapReset, ...
'Basis', Basis, 'Principal', Principal)

Price = 2.0592

PriceTree = struct with fields:
    FinObj: 'BDTPriceTree'
      tObs: [0 1 2 3 4 5 6 7 8 9 10]
     PTree: {1x11 cell}
```

Price a 5-Year Call Swaption with Receiving and Paying Legs Using a BDT Interest-Rate Tree

This example shows how to price a 5-year call swaption with receiving and paying legs 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 = [datetime(2007,1,1) ; datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1)
EndDates = [datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1) ; datetime(2011,1,1) ; d
ValuationDate = datetime(2007,1,1);
Compounding = 1;
```

Define the RateSpec.

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

RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [10x1 double]
            Rates: [10x1 double]
         EndTimes: [10x1 double]
       StartTimes: [10x1 double]
         EndDates: [10x1 double]
       StartDates: [10x1 double]
    ValuationDate: 733043
            Basis: 0
```

```
     EndMonthRule: 1
```

Use `VolSpec` to compute 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 a BDT tree.

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

Build the BDT tree.

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

```
BDTTree = struct with fields:
      FinObj: 'BDTFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
        tObs: [0 1 2 3 4 5 6 7 8 9]
        dObs: [733043 733408 733774 734139 734504 734869 735235 735600 ... ]
        TFwd: {1x10 cell}
      CFlowT: {1x10 cell}
     FwdTree: {1x10 cell}
```

Define the swaption arguments.

```
ExerciseDates = datetime(2012,1,1);
SwapSettlement = ExerciseDates;
SwapMaturity   = datetime(2015,1,1);
Spread = 0;
SwapReset = [1 1]; % 1st column represents receiving leg, 2nd column represents paying leg
Principal = 100;
OptSpec = 'call';
Strike=.062;
Basis= [2 4]; % 1st column represents receiving leg, 2nd column represents paying leg
```

Price the swaption.

```
[Price, PriceTree] = swaptionbybdt(BDTTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity, 'SwapReset', SwapReset, ...
'Basis', Basis, 'Principal', Principal)
```

```
Price = 2.0592
```

```
PriceTree = struct with fields:
    FinObj: 'BDTPriceTree'
      tObs: [0 1 2 3 4 5 6 7 8 9 10]
     PTree: {1x11 cell}
```

Input Arguments

`BDTTree` — Interest-rate tree structure
structure

Interest-rate tree structure, specified by using `bdttree`.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vector with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors. For more information, see “More About” on page 11-1884.

Data Types: `char` | `cell`

`Strike` — Strike swap rate values
decimal

Strike swap rate values, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`ExerciseDates` — Exercise dates for swaption
datetime array | string array | date character vector

Exercise dates for the swaption, specified as a `NINST`-by-`1` vector or a `NINST`-by-`2` vector using a datetime array, string array, or date character vectors, depending on the option type.

- For a European option, `ExerciseDates` are a `NINST`-by-`1` vector of exercise dates. Each row is the schedule for one option. When using a European option, there is only one `ExerciseDate` on the option expiry date.
- For an American option, `ExerciseDates` are a `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 `ValuationDate` of the tree and the single listed `ExerciseDate`.

To support existing code, `swaptionbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`Spread` — Number of basis points over reference rate
numeric

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date (representing the settle date for each swap), specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every swaption is set to the `ValuationDate` of the BDT tree. The swap argument `Settle` is ignored. The underlying swap starts at the maturity of the swaption.

To support existing code, `swaptionbybdt` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for swap
datetime array | string array | date character vector

Maturity date for each swap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swaptionbybdt` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = swaptionbybdt(BDTTree,OptSpec, ExerciseDates,Spread,Settle,Maturity,'SwapReset',4,'Basis',5,'Principal',10000)`

`AmericanOpt` — Option type
`0` (European) (default) | integer with values `0` or `1`

(Optional) Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer flags with values:

- `0` — European
- `1` — American

Data Types: `double`

`SwapReset` — Reset frequency per year for underlying swap
`1` (default) | numeric

Reset frequency per year for the underlying swap, specified as the comma-separated pair consisting of `'SwapReset'` and a `NINST`-by-`1` vector or `NINST`-by-`2` matrix representing the reset frequency per year for each leg. If `SwapReset` is `NINST`-by-`2`, the first column represents the receiving leg, while the second column represents the paying leg.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward rate tree for each instrument, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector or `NINST`-by-`2` matrix representing the basis for each leg. If `Basis` is `NINST`-by-`2`, the first column represents the receiving leg, while the second column represents the paying leg.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of swaptions at time 0
vector

Expected prices of the swaptions at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.

More About

Call 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.

Put Swaption

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.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swaptionbybdt` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bdttree` | `instswaption` | `swapbybdt` | `Swaption`

Topics

“Computing Instrument Prices” on page 2-81
“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Swaption” on page 2-14
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

swaptionbybk

Price swaption from Black-Karasinski interest-rate tree

Syntax

```
[Price,PriceTree] = swaptionbybk(BKTree,OptSpec,Strike,ExerciseDates,Spread,
Settle,Maturity)
[Price,PriceTree] = swaptionbybk( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = swaptionbybk(BKTree,OptSpec,Strike,ExerciseDates,Spread, Settle,Maturity)` prices swaption using a Black-Karasinski tree.

Note Alternatively, you can use the `Swaption` object to price swaption instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = swaptionbybk( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a 4-Year Call and Put Swaption Using a BK Interest-Rate Tree

This example shows how to price a 4-year call and put swaption using a BK interest-rate tree, assuming the interest rate is fixed at 7% annually.

```
Rates =0.07 * ones (10,1);
Compounding = 2;
StartDates = [datetime(2007,1,1) ; datetime(2007,7,1) ; datetime(2008,1,1) ; datetime(2008,7,1)
EndDates = [datetime(2007,7,1) ; datetime(2008,1,1) ; datetime(2008,7,1) ; datetime(2009,1,1) ; d
ValuationDate = datetime(2007,1,1);

% define the RateSpec

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 1-year swap and 4-year swaption
ExerciseDates = datetime(2011,1,1);
SwapSettlement = ExerciseDates;
SwapMaturity   = datetime(2012,1,1);
Spread = 0;
SwapReset = 2 ;
Principal = 100;
OptSpec = {'call' ;'put'};
Strike= [ 0.07 ; 0.0725];
Basis=1;

% price the swaption
PriceSwaption = swaptionbybk(BKTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity, 'SwapReset', SwapReset, 'Basis', Basis, ...
'Principal', Principal)
```

```
PriceSwaption = 2×1

    0.3634
    0.4798
```

Price a 4-Year Call and Put Swaption with Receiving and Paying Legs Using a BK Interest-Rate Tree

This example shows how to price a 4-year call and put swaption with receiving and paying legs using a BK interest-rate tree, assuming the interest rate is fixed at 7% annually. Build a tree with the following data.

```
Rates =0.07 * ones (10,1);
Compounding = 2;
StartDates = [datetime(2007,1,1) ; datetime(2007,7,1) ; datetime(2008,1,1) ; datetime(2008,7,1)
EndDates = [datetime(2007,7,1) ; datetime(2008,1,1) ; datetime(2008,7,1) ; datetime(2009,1,1) ; d
ValuationDate = datetime(2007,1,1);
```

Define the `RateSpec`.

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [10x1 double]
            Rates: [10x1 double]
         EndTimes: [10x1 double]
       StartTimes: [10x1 double]
         EndDates: [10x1 double]
       StartDates: [10x1 double]
    ValuationDate: 733043
            Basis: 0
     EndMonthRule: 1
```

Use BKVolSpec to compute 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 a BK tree.

```
BKTimeSpec = bktimespec(ValuationDate, EndDates, Compounding);
```

Build the BK tree.

```
BKTree = bktree(BKVolSpec, RateSpec, BKTimeSpec);
```

Define the arguments for a 1-year swap and 4-year swaption.

```
ExerciseDates = datetime(2011,1,1);
SwapSettlement = ExerciseDates;
SwapMaturity   = datetime(2012,1,1);
Spread = 0;
SwapReset = [2 2]; % 1st column represents swaption receiving leg, 2nd column represents swaptio
Principal = 100;
OptSpec = {'call' ;'put'};
Strike= [ 0.07 ; 0.0725];
Basis= [1 3]; % 1st column represents swaption receiving leg, 2nd column represents swaption pay
```

Price the swaption.

```
PriceSwaption = swaptionbybk(BKTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity, 'SwapReset', SwapReset, 'Basis', Basis, ...
'Principal', Principal)
```

```
PriceSwaption = 2×1

    0.3634
    0.4798
```

Input Arguments

BKTree — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using bktree.

Data Types: struct

OptSpec — Definition of option

character vector with values 'call' or 'put' | cell array of character vector with values 'call' or 'put'

Definition of the option as 'call' or 'put', specified as a NINST-by-1 cell array of character vectors. For more information, see “More About” on page 11-1890.

Data Types: char | cell

`Strike` — Strike swap rate values
decimal

Strike swap rate values, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`ExerciseDates` — Exercise dates for swaption
datetime array | string array | date character vector

Exercise dates for the swaption, specified as a `NINST`-by-`1` vector or a `NINST`-by-`2` vector using a datetime array, string array, or date character vectors, depending on the option type.

- For a European option, `ExerciseDates` are a `NINST`-by-`1` vector of exercise dates. Each row is the schedule for one option. When using a European option, there is only one `ExerciseDate` on the option expiry date.
- For an American option, `ExerciseDates` are a `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 `ValuationDate` of the tree and the single listed `ExerciseDate`.

To support existing code, `swaptionbybk` also accepts serial date numbers as inputs, but they are not recommended.

`Spread` — Number of basis points over reference rate
numeric

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date (representing the settle date for each swap), specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors,. The `Settle` date for every swaption is set to the `ValuationDate` of the BK tree. The swap argument `Settle` is ignored. The underlying swap starts at the maturity of the swaption.

To support existing code, `swaptionbybk` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for swap
datetime array | string array | date character vector

Maturity date for each swap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swaptionbybk` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = swaptionbykbk(BKTree,OptSpec, ExerciseDates,Spread,Settle,Maturity,'SwapReset',4,'Basis',5,'Principal',10000)`

`AmericanOpt` — Option type

`0` (European) (default) | integer with values `0` or `1`

(Optional) Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer flags with values:

- `0` — European
- `1` — American

Data Types: `double`

`SwapReset` — Reset frequency per year for underlying swap

`1` (default) | numeric

Reset frequency per year for the underlying swap, specified as the comma-separated pair consisting of `'SwapReset'` and a `NINST`-by-`1` vector or `NINST`-by-`2` matrix representing the reset frequency per year for each leg. If `SwapReset` is `NINST`-by-`2`, the first column represents the receiving leg, while the second column represents the paying leg.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward rate tree for each instrument, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector or `NINST`-by-`2` matrix representing the basis for each leg. If `Basis` is `NINST`-by-`2`, the first column represents the receiving leg, while the second column represents the paying leg.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of swaptions at time 0
vector

Expected prices of the swaptions at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.

More About

Call 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.

Put Swaption

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.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swaptionbybk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bktree` | `instswaption` | `swapbybk` | `Swaption`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Swaption” on page 2-14
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

swaptionbycir

Price swaption from Cox-Ingersoll-Ross interest-rate tree

Syntax

```
[Price,PriceTree] = swaptionbycir(CIRTree,OptSpec,Strike,ExerciseDates,
Spread,Settle,Maturity)
[Price,PriceTree] = swaptionbycir( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = swaptionbycir(CIRTree,OptSpec,Strike,ExerciseDates, Spread,Settle,Maturity)` prices swaption with a Cox-Ingersoll-Ross (CIR) tree using a CIR++ model with the Nawalka-Beliaeva (NB) approach.

`[Price,PriceTree] = swaptionbycir( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Swaption Using a CIR Interest-Rate Tree

Define a 3-year put swaption.

```
Rates =0.075 * ones (10,1);
Compounding = 2;
StartDates = [datetime(2017,1,1);datetime(2017,7,1);datetime(2018,1,1);datetime(2018,7,1);dateti
EndDates = [datetime(2017,7,1);datetime(2018,1,1);datetime(2018,7,1);datetime(2019,1,1);datetime

ValuationDate = datetime(2017,1,1);
```

Create a `RateSpec` using the `intenvset` function.

```
RateSpec = intenvset('ValuationDate', ValuationDate, 'StartDates', ValuationDate, 'EndDates',End
```

Create a `CIR` tree.

```
NumPeriods = length(EndDates);
Alpha = 0.03;
Theta = 0.02;
Sigma = 0.1;
Maturity = datetime(2023,1,1);
CIRTimeSpec = cirtimespec(ValuationDate, Maturity, NumPeriods);
CIRVolSpec = cirvolspec(Sigma, Alpha, Theta);

CIRT = cirtree(CIRVolSpec, RateSpec, CIRTimeSpec)
```

```
CIRT = struct with fields:
      FinObj: 'CIRFwdTree'
     VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
```

```
    RateSpec: [1x1 struct]
        tObs: [0 0.6000 1.2000 1.8000 2.4000 3 3.6000 4.2000 4.8000 5.4000]
        dObs: [736696 736915 737134 737353 737572 737791 738010 738229 ... ]
     FwdTree: {1x10 cell}
     Connect: {1x9 cell}
       Probs: {1x9 cell}
```

Use the following arguments for a 1-year swap and a 3-year swaption.

```
ExerciseDates = datetime(2020,1,1);
SwapSettlement = ExerciseDates;
SwapMaturity   = datetime(2022,1,1);
Spread = 0;
SwapReset = 2 ;
Principal = 100;
OptSpec = 'put';
Strike= 0.04;
Basis=1;
```

Price the swaption.

```
[Price,PriceTree] = swaptionbycir(CIRT,OptSpec,Strike,ExerciseDates,Spread,SwapSettlement,SwapMa
'Basis',Basis,'Principal',Principal)
```

```
Price = 3.1537
```

```
PriceTree = struct with fields:
     FinObj: 'CIRPriceTree'
      PTree: {1x11 cell}
       tObs: [0 0.6000 1.2000 1.8000 2.4000 3 3.6000 4.2000 4.8000 5.4000 6]
    Connect: {1x9 cell}
      Probs: {1x9 cell}
```

Input Arguments

`CIRTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `cirtree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vector with values `'call'` or `'put'` | string array with values `"call"` or `"put"`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-`1` cell array of character vectors or string arrays. For more information, see “More About” on page 11-1890.

Data Types: `char` | `cell` | `string`

`Strike` — Strike swap rate values

decimal

Strike swap rate values, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`ExerciseDates` — Exercise dates for swaption
datetime array | string array | date character vector

Exercise dates for the swaption, specified as a `NINST`-by-`1` vector or a `NINST`-by-`2` vector using a datetime array, string array, or date character vectors, depending on the option type.

- For a European option, `ExerciseDates` are a `NINST`-by-`1` vector of exercise dates. Each row is the schedule for one option. When using a European option, there is only one `ExerciseDate` on the option expiry date.
- For an American option, `ExerciseDates` are a `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 `ValuationDate` of the tree and the single listed `ExerciseDate`.

To support existing code, `swaptionbycir` also accepts serial date numbers as inputs, but they are not recommended.

`Spread` — Number of basis points over reference rate
numeric

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date (representing the settle date for each swap), specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every swaption is set to the `ValuationDate` of the CIR tree. The swap argument `Settle` is ignored. The underlying swap starts at the maturity of the swaption.

To support existing code, `swaptionbycir` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for swap
datetime array | string array | date character vector

Maturity date for each swap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swaptionbycir` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = swaptionbycir(CIRTree,OptSpec, ExerciseDates,Spread,Settle,Maturity,'SwapReset',4,'Basis',5,'Principal',1000 0)`

`AmericanOpt` — Option type
`0` (European) (default) | integer with values `0` or `1`

Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer flags with values:

- `0` — European
- `1` — American

Data Types: `double`

`SwapReset` — Reset frequency per year for underlying swap
`1` (default) | numeric

Reset frequency per year for the underlying swap, specified as the comma-separated pair consisting of `'SwapReset'` and a `NINST`-by-`1` vector or `NINST`-by-`2` matrix representing the reset frequency per year for each leg. If `SwapReset` is `NINST`-by-`2`, the first column represents the receiving leg, and the second column represents the paying leg.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward-rate tree for each instrument, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector or `NINST`-by-`2` matrix representing the basis for each leg. If `Basis` is `NINST`-by-`2`, the first column represents the receiving leg, while the second column represents the paying leg.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

Output Arguments

`Price` — Expected prices of swaptions at time 0
vector

Expected prices of the swaptions at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, returned as 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.
- `PriceTree.Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicated where the down branch connects to.
- `PriceTree.Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

More About

Call 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.

Put Swaption

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.

Version History

Introduced in R2018a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `swaptionbycir` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Cox, J., Ingersoll, J., and S. Ross. "A Theory of the Term Structure of Interest Rates." *Econometrica*. Vol. 53, 1985.

[2] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[3] Hirsa, A. *Computational Methods in Finance*. CRC Press, 2012.

[4] Nawalka, S., Soto, G., and N. Beliaeva. *Dynamic Term Structure Modeling*. Wiley, 2007.

[5] Nelson, D. and K. Ramaswamy. "Simple Binomial Processes as Diffusion Approximations in Financial Models." *The Review of Financial Studies*. Vol 3. 1990, pp. 393–430.

See Also

`bondbycir` | `capbycir` | `cfbycir` | `fixedbycir` | `floatbycir` | `floorbycir` | `oasbycir` | `optbndbycir` | `optfloatbycir` | `optembndbycir` | `optemfloatbycir` | `rangefloatbycir` | `swapbycir` | `instswaption`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Swaption” on page 2-14
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

swaptionbyblk

Price European swaption instrument using Black model

Syntax

```
Price = swaptionbyblk(RateSpec,OptSpec,Strike,Settle,ExerciseDates,Maturity,
Volatility)
Price = swaptionbyblk( ___ ,Name,Value)
```

Description

`Price = swaptionbyblk(RateSpec,OptSpec,Strike,Settle,ExerciseDates,Maturity, Volatility)` prices swaptions using the Black option pricing model.

Note Alternatively, you can use the `Swaption` object to price swaption instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = swaptionbyblk( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a European Swaption Using the Black Model Where the Yield Curve is Flat at 6%

Price a European swaption that gives the holder the right to enter in five years into a three-year paying swap where a fixed-rate of 6.2% is paid and floating is received. Assume that the yield curve is flat at 6% per annum with continuous compounding, the volatility of the swap rate is 20%, the principal is $100, and payments are exchanged semiannually.

Create the `RateSpec`.

```
Rate = 0.06;
Compounding  = -1;
ValuationDate = datetime(2010,1,1);
EndDates =   datetime(2020,1,1);
Basis = 1;

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

Price the swaption using the Black model.

```
Settle = datetime(2011,1,1);
ExerciseDates = datetime(2016,1,1);
Maturity = datetime(2019,1,1);
Reset = 2;
Principal = 100;
Strike = 0.062;
```

```
Volatility = 0.2;
OptSpec = 'call';

Price= swaptionbyblk(RateSpec, OptSpec, Strike, Settle, ExerciseDates, Maturity, ...
Volatility, 'Reset', Reset, 'Principal', Principal, 'Basis', Basis)
```

```
Price = 2.0710
```

Price a European Swaption with Receiving and Paying Legs Using the Black Model Where the Yield Curve is 6%

This example shows Price a European swaption with receiving and paying legs that gives the holder the right to enter in five years into a three-year paying swap where a fixed-rate of 6.2% is paid and floating is received. Assume that the yield curve is flat at 6% per annum with continuous compounding, the volatility of the swap rate is 20%, the principal is $100, and payments are exchanged semiannually.

```
Rate = 0.06;
Compounding  = -1;
ValuationDate = datetime(2010,1,1);
EndDates =   datetime(2020,1,1);
Basis = 1;
```

Define the RateSpec.

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

Define the swaption arguments.

```
Settle = datetime(2011,1,1);
ExerciseDates = datetime(2016,1,1);
Maturity = datetime(2019,1,1);
Reset = [2 4]; % 1st column represents receiving leg, 2nd column represents paying leg
Principal = 100;
Strike = 0.062;
Volatility = 0.2;
OptSpec = 'call';
Basis = [1 3]; % 1st column represents receiving leg, 2nd column represents paying leg
```

Price the swaption.

```
Price= swaptionbyblk(RateSpec,OptSpec,Strike,Settle,ExerciseDates,Maturity,Volatility, ...
'Reset',Reset,'Principal',Principal,'Basis',Basis)
```

```
Price = 1.6494
```

Price a European Swaption Using the Black Model Where the Yield Curve Is Incrementally Increasing

Price a European swaption that gives the holder the right to enter into a 5-year receiving swap in a year, where a fixed rate of 3% is received and floating is paid. Assume that the 1-year, 2-year, 3-year, 4-year and 5- year zero rates are 3%, 3.4%, 3.7%, 3.9% and 4% with continuous compounding. The swap rate volatility is 21%, the principal is $1000, and payments are exchanged semiannually.

Create the RateSpec.

```
ValuationDate = datetime(2010,1,1);
EndDates = [datetime(2011,1,1) ; datetime(2012,1,1) ; datetime(2013,1,1) ; datetime(2014,1,1) ; d
Rates = [0.03; 0.034 ; 0.037; 0.039; 0.04;];
Compounding = -1;
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: -1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: [5x1 double]
       StartDates: 734139
    ValuationDate: 734139
            Basis: 1
     EndMonthRule: 1
```

Price the swaption using the Black model.

```
Settle = datetime(2011,1,1);
ExerciseDates = datetime(2012,1,1);
Maturity = datetime(2017,1,1);
Strike = 0.03;
Volatility = 0.21;
Principal =1000;
Reset = 2;
OptSpec = 'put';

Price = swaptionbyblk(RateSpec, OptSpec, Strike, Settle,  ExerciseDates, ...
Maturity, Volatility,'Basis', Basis, 'Reset', Reset,'Principal', Principal)
```

```
Price = 0.5771
```

Price a Swaption Using a Different Curve to Generate the Future Forward Rates

Define the OIS and Libor curves.

```
Settle = datetime(2013,3,15);
CurveDates = daysadd(Settle,360*[1/12 2/12 3/12 6/12 1 2 3 4 5 7 10],1);
OISRates = [.0018 .0019 .0021 .0023 .0031 .006  .011 .017 .021 .026 .03]';
LiborRates = [.0045 .0047 .005 .0055 .0075 .0109  .0162 .0216 .0262 .0309 .0348]';
```

Create an associated RateSpec for the OIS and Libor curves.

```
OISCurve = intenvset('Rates',OISRates,'StartDate',Settle,'EndDates',CurveDates,'Compounding',2,'
LiborCurve = intenvset('Rates',LiborRates,'StartDate',Settle,'EndDates',CurveDates,'Compounding'
```

Define the swaption instruments.

```
ExerciseDate = datetime(2018,3,15);
Maturity = [datetime(2020,3,15) ; datetime(2023,3,15)];
OptSpec = 'call';
Strike = 0.04;
BlackVol = 0.2;
```

Price the swaption instruments using the term structure `OISCurve` both for discounting the cash flows and generating the future forward rates.

```
Price = swaptionbyblk(OISCurve, OptSpec, Strike, Settle, ExerciseDate, Maturity, BlackVol,'Reset
```

```
Price = 2×1

    1.0956
    2.6944
```

Price the swaption instruments using the term structure `LiborCurve` to generate the future forward rates. The term structure `OISCurve` is used for discounting the cash flows.

```
PriceLC = swaptionbyblk(OISCurve, OptSpec, Strike, Settle, ExerciseDate, Maturity, BlackVol,'Pro
```

```
PriceLC = 2×1

    1.5346
    3.8142
```

Price a Swaption Using the Shifted Black Model

Create the `RateSpec`.

```
ValuationDate = datetime(2016,1,1);
EndDates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; d
Rates = [-0.02; 0.024 ; 0.047; 0.090; 0.12;]/100;
Compounding = 1;
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: [5x1 double]
       StartDates: 736330
    ValuationDate: 736330
            Basis: 1
     EndMonthRule: 1
```

Price the swaption with a negative strike using the Shifted Black model.

```
Settle = datetime(2016,1,1);
ExerciseDates = datetime(2017,1,1);
Maturity = 'Jan-1-2020';
Strike = -0.003; % Set -0.3 percent strike.
ShiftedBlackVolatility = 0.31;
Principal = 1000;
Reset = 1;
OptSpec = 'call';
Shift = 0.008; % Set 0.8 percent shift.

Price = swaptionbyblk(RateSpec,OptSpec,Strike,Settle,ExerciseDates, ...
Maturity,ShiftedBlackVolatility,'Basis',Basis,'Reset',Reset,...
'Principal',Principal,'Shift',Shift)
```

```
Price = 12.8301
```

Price Swaptions Using the Shifted Black Model with a Vector of Shifts

Create the RateSpec.

```
ValuationDate = datetime(2016,1,1);
EndDates = [datetime(2017,1,1) ; datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1) ; d
Rates = [-0.02; 0.024 ; 0.047; 0.090; 0.12;]/100;
Compounding = 1;
Basis = 1;

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

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 1
             Disc: [5x1 double]
            Rates: [5x1 double]
         EndTimes: [5x1 double]
       StartTimes: [5x1 double]
         EndDates: [5x1 double]
       StartDates: 736330
    ValuationDate: 736330
            Basis: 1
     EndMonthRule: 1
```

Price the swaptions with using the Shifted Black model.

```
Settle = datetime(2016,1,1);
ExerciseDates = datetime(2017,1,1);
Maturities = [datetime(2018,1,1) ; datetime(2019,1,1) ; datetime(2020,1,1)];
Strikes = [-0.0034;-0.0032;-0.003];
ShiftedBlackVolatilities = [0.33;0.32;0.31]; % A vector of volatilities.
Principal = 1000;
Reset = 1;
OptSpec = 'call';
Shifts = [0.0085;0.0082;0.008]; % A vector of shifts.

Prices = swaptionbyblk(RateSpec,OptSpec,Strikes,Settle,ExerciseDates, ...
```

```
Maturities,ShiftedBlackVolatilities,'Basis',Basis,'Reset',Reset, ...
'Principal',Principal,'Shift',Shifts)

Prices = 3×1

    4.1117
    8.0577
   12.8301
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

If the paying leg is different than the receiving leg, the `RateSpec` can be a `NINST`-by-2 input variable of `RateSpecs`, with the second input being the discount curve for the paying leg. If only one curve is specified, then it is used to discount both legs.

Data Types: `struct`

OptSpec — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vector with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-1 cell array of character vectors.

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.

Data Types: `char` | `cell`

Strike — Strike swap rate values

decimal

Strike swap rate values, specified as a `NINST`-by-1 vector of decimal values.

Data Types: `double`

Settle — Settlement date

datetime array | string array | date character vector

Settlement date (representing the settle date for each swaption), specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors. `Settle` must not be later than `ExerciseDates`.

To support existing code, `swaptionbyblk` also accepts serial date numbers as inputs, but they are not recommended.

The Settle date input for swaptionbyblk is the valuation date on which the swaption (an option to enter into a swap) is priced. The swaption buyer pays this price on this date to hold the swaption.

ExerciseDates — Dates on which swaption expires and underlying swap starts
datetime array | string array | date character vector

Dates, specified as a vector using a datetime array, string array, or date character vectors on which the swaption expires and the underlying swap starts. The swaption holder can choose to enter into the swap on this date if the situation is favorable.

To support existing code, swaptionbyblk also accepts serial date numbers as inputs, but they are not recommended.

For a European option, ExerciseDates are a NINST-by-1 vector of exercise dates. Each row is the schedule for one option. When using a European option, there is only one ExerciseDate on the option expiry date.

Maturity — Maturity date for each forward swap
datetime array | string array | date character vector

Maturity date for each forward swap, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, swaptionbyblk also accepts serial date numbers as inputs, but they are not recommended.

Volatility — Annual volatilities values
numeric

Annual volatilities values, specified as a NINST-by-1 vector of numeric values.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: Price = swaptionbyblk(OISCurve,OptSpec,Strike,Settle,ExerciseDate,Maturity,BlackVol,'Reset',1,'Shift',.5)

Basis — Day-count basis of instrument
0 (actual/actual) (default) | integer from 0 to 13

Day-count basis of the instrument, specified as the comma-separated pair consisting of 'Basis' and a NINST-by-1 vector or NINST-by-2 matrix representing the basis for each leg. If Basis is NINST-by-2, the first column represents the receiving leg, while the second column represents the paying leg. Default is 0 (actual/actual).

- 0 = actual/actual
- 1 = 30/360 (SIA)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Reset` — Reset frequency per year for underlying forward swap
`1` (default) | numeric

Reset frequency per year for the underlying forward swap, specified as the comma-separated pair consisting of `'Reset'` and a `NINST`-by-`1` vector or `NINST`-by-`2` matrix representing the reset frequency per year for each leg. If `Reset` is `NINST`-by-`2`, the first column represents the receiving leg, while the second column represents the paying leg.

Data Types: `double`

`ProjectionCurve` — Rate curve used in generating future forward rates
if `ProjectionCurve` is not specified, then `RateSpec` is used both for discounting cash flows and projecting future forward rates (default) | structure

The rate curve to be used in generating the future forward rates, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a structure created using `intenvset`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `struct`

`Shift` — Shift in decimals for shifted Black model
`0` (no shift) (default) | positive decimal

Shift in decimals for the shifted Black model, specified as the comma-separated pair consisting of `'Shift'` and a scalar or `NINST`-by-`1` vector of rate shifts in positive decimals. Set this parameter to a positive rate shift in decimals to add a positive shift to the forward swap rate and strike, which effectively sets a negative lower bound for the forward swap rate and strike. For example, a `Shift` of `0.01` is equal to a 1% shift.

Data Types: `double`

Output Arguments

`Price` — Prices for swaptions at time 0

vector

Prices for the swaptions at time 0, returned as a `NINST`-by-`1` vector of prices.

More About

Forward Swap

A forward swap is a swap that starts at a future date.

Shifted Black

The Shifted Black model is essentially the same as the Black's model, except that it models the movements of ($F + Shift$) as the underlying asset, instead of F (which is the forward swap rate in the case of swaptions).

This model allows negative rates, with a fixed negative lower bound defined by the amount of shift; that is, the zero lower bound of Black's model has been shifted.

Algorithms

Black Model

$$dF = \sigma_{Black} F dw$$

$$call = e^{-\gamma T}[FN(d_1) - KN(d_2)]$$

$$put = e^{-\gamma T}[KN(-d_2) - FN(-d_1)]$$

$$d_1 = \frac{\ln\left(\frac{F}{K}\right) + \left(\frac{\sigma_B{}^2}{2}\right)T}{\sigma_B\sqrt{T}}, \quad d_2 = d_1 - \sigma_B\sqrt{T}$$

$$\sigma_B = \sigma_{Black}$$

Where F is the forward value and K is the strike.

Shifted Black Model

$$dF = \sigma_{Shifted\_Black}(F + Shift)dw$$

$$call = e^{-\gamma T}[(F + Shift)N(d_{s1}) - (K + Shift)N(d_{s2})]$$

$$put = e^{-\gamma T}[(K + Shift)N(-d_{s2}) - (F + Shift)N(-d_{s1})]$$

$$d_{s1} = \frac{\ln\left(\frac{F + Shift}{K + Shift}\right) + \left(\frac{\sigma_{sB}{}^2}{2}\right)T}{\sigma_{sB}\sqrt{T}}, \quad d_{s2} = d_{s1} - \sigma_{sB}\sqrt{T}$$

$$\sigma_{sB} = \sigma_{Shifted\_Black}$$

Where *F+Shift* is the forward value and *K+Shift* is the strike for the shifted version.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swaptionbyblk` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bondbyzero` | `cfbyzero` | `fixedbyzero` | `floatbyzero` | `blackvolbysabr` | `intenvset` | `swaptionbynormal` | `capbyblk` | `floorbyblk` | `Swaption`

Topics

“Calibrate the SABR Model” on page 2-33
“Price a Swaption Using the SABR Model” on page 2-38
“Price Swaptions with Negative Strikes Using the Shifted SABR Model” on page 2-26
“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Swaption” on page 2-14
“Work with Negative Interest Rates Using Functions” on page 2-18
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

External Websites

How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

swaptionbyhjm

Price swaption from Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price,PriceTree] = swaptionbyhjm(HJMTree,OptSpec,Strike,ExerciseDates,
Spread,Settle,Maturity)
[Price,PriceTree] = swaptionbyhjm( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = swaptionbyhjm(HJMTree,OptSpec,Strike,ExerciseDates, Spread,Settle,Maturity)` prices swaption using a Heath-Jarrow-Morton tree.

`[Price,PriceTree] = swaptionbyhjm( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a 1-Year Call Swaption Using an HJM Interest-Rate Tree

This example shows how to 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.

```
Rates = [ 0.05;0.05;0.05;0.05];
StartDates = datetime(2007,1,1);
EndDates =[datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1) ; datetime(2011,1,1)];
ValuationDate = StartDates;
Compounding = 1;

% define the RateSpec
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
ExerciseDates = datetime(2008,1,1);
SwapSettlement = ExerciseDates;
SwapMaturity   = datetime(2010,1,1);
Spread = [0];
SwapReset = 1;
```

```
Basis  = 1;
Principal = 100;
OptSpec = 'call';
Strike=0.05;

% price the swaption

[Price, PriceTree] = swaptionbyhjm(HJMTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity,'SwapReset', SwapReset, ...
'Basis', Basis, 'Principal', Principal)

Price = 0.9296

PriceTree = struct with fields:
    FinObj: 'HJMPriceTree'
      tObs: [5x1 double]
     PBush: {[0.9296]  [1x1x2 double]  [1x2x2 double]  [1x4x2 double]  [0 ... ]}
```

Price a 1-Year Call Swaption with Receiving and Paying Legs Using an HJM Interest-Rate Tree

This example shows how to price a 1-year call swaption with receiving and paying legs 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.

```
Rates = [ 0.05;0.05;0.05;0.05];
StartDates = datetime(2007,1,1);
EndDates =[datetime(2008,1,1) ; datetime(2009,1,1) ; datetime(2010,1,1) ; datetime(2011,1,1)];
ValuationDate = StartDates;
Compounding = 1;
```

Define the `RateSpec`.

```
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

```
ExerciseDates = datetime(2008,1,1);
SwapSettlement = ExerciseDates;
SwapMaturity   = datetime(2010,1,1);
Spread = [0];
SwapReset = [1 1];  % 1st column represents receiving leg, 2nd column represents paying leg
```

```
Basis  = [1 3];     % 1st column represents receiving leg, 2nd column represents paying leg
Principal = 100;
OptSpec = 'call';
Strike=0.05;
```

Price the swaption.

```
[Price, PriceTree] = swaptionbyhjm(HJMTree,OptSpec,Strike,ExerciseDates, ...
Spread,SwapSettlement,SwapMaturity,'SwapReset',SwapReset, ...
'Basis',Basis,'Principal',Principal)
```

```
Price = 0.9296
```

```
PriceTree = struct with fields:
    FinObj: 'HJMPriceTree'
      tObs: [5x1 double]
     PBush: {[0.9296]  [1x1x2 double]  [1x2x2 double]  [1x4x2 double]  [0 ... ]}
```

Input Arguments

`HJMTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hjmtree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vector with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-1 cell array of character vectors. For more information, see “More About” on page 11-1913.

Data Types: `char` | `cell`

`Strike` — Strike swap rate values

decimal

Strike swap rate values, specified as a `NINST`-by-1 vector.

Data Types: `double`

`ExerciseDates` — Exercise dates for swaption

datetime array | string array | date character vector

Exercise dates for the swaption, specified as a `NINST`-by-1 vector or a `NINST`-by-2 vector using a datetime array, string array, or date character vectors, depending on the option type.

- For a European option, `ExerciseDates` are a `NINST`-by-1 vector of exercise dates. Each row is the schedule for one option. When using a European option, there is only one `ExerciseDate` on the option expiry date.
- For an American option, `ExerciseDates` are a `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 `ValuationDate` of the tree and the single listed `ExerciseDate`.

To support existing code, `swaptionbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`Spread` — Number of basis points over reference rate
numeric

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date (representing the settle date for each swap), specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every swaption is set to the `ValuationDate` of the HJM tree. The swap argument `Settle` is ignored. The underlying swap starts at the maturity of the swaption.

To support existing code, `swaptionbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for swap
datetime array | string array | date character vector

Maturity date for each swap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swaptionbyhjm` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = swaptionbyhjm(HJMTree,OptSpec, ExerciseDates,Spread,Settle,Maturity,'SwapReset',4,'Basis',5,'Principal',10000)`

`AmericanOpt` — Option type
`0` (European) (default) | integer with values `0` or `1`

(Optional) Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer flags with values:

- `0` — European
- `1` — American

Data Types: `double`

SwapReset — Reset frequency per year for underlying swap
1 (default) | numeric

Reset frequency per year for the underlying swap, specified as the comma-separated pair consisting of 'SwapReset' and a NINST-by-1 vector or NINST-by-2 matrix representing the reset frequency per year for each leg. If SwapReset is NINST-by-2, the first column represents the receiving leg, while the second column represents the paying leg.

Data Types: double

Basis — Day-count basis of instrument
0 (actual/actual) (default) | integer from 0 to 13

Day-count basis representing the basis used when annualizing the input forward rate tree for each instrument, specified as the comma-separated pair consisting of 'Basis' and a NINST-by-1 vector or NINST-by-2 matrix representing the basis for each leg. If Basis is NINST-by-2, the first column represents the receiving leg, while the second column represents the paying leg.

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

For more information, see “Basis” on page 2-228.

Data Types: double

Principal — Notional principal amount
100 (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of 'Principal' and a NINST-by-1 vector.

Data Types: double

Options — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of 'Options' and a structure obtained from using derivset.

Data Types: struct

Output Arguments

`Price` — Expected prices of swaptions at time 0

vector

Expected prices of the swaptions at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as 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.

More About

Call 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.

Put Swaption

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.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swaptionbyhjm` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hjmtree` | `instswaption` | `swapbyhjm`

Topics

“Computing Instrument Prices” on page 2-81
“Swaption” on page 2-14
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3

swaptionbyhw

Price swaption from Hull-White interest-rate tree

Syntax

```
[Price,PriceTree] = swaptionbyhw(HWTree,OptSpec,Strike,ExerciseDates,Spread,
Settle,Maturity)
[Price,PriceTree] = swaptionbyhw( ___ ,Name,Value)
```

Description

`[Price,PriceTree] = swaptionbyhw(HWTree,OptSpec,Strike,ExerciseDates,Spread, Settle,Maturity)` prices swaption using a Hull-White tree.

Note Alternatively, you can use the `Swaption` object to price swaption instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Price,PriceTree] = swaptionbyhw( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a 3-Year Put Swaption Using an HW Interest-Rate Tree

This example shows how to price a 3-year put swaption using an HW interest-rate tree with the following data.

```
Rates =0.075 * ones (10,1);
Compounding = 2;
StartDates = [datetime(2007,1,1);datetime(2007,7,1);datetime(2008,1,1);datetime(2008,7,1);dateti
EndDates = [datetime(2007,7,1);datetime(2008,1,1);datetime(2008,7,1);datetime(2009,1,1);datetime
ValuationDate = datetime(2007,1,1);

% Define the RatesSpec
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 1-year swap and 3-year swaption
ExerciseDates = datetime(2010,1,1);
SwapSettlement = ExerciseDates;
SwapMaturity   = datetime(2012,1,1);
Spread = 0;
SwapReset = 2 ;
Principal = 100;
OptSpec = 'put';
Strike= 0.04;
Basis=1;

% Price the swaption
PriceSwaption = swaptionbyhw(HWTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity,'SwapReset', SwapReset, ...
'Basis', Basis,'Principal', Principal)

PriceSwaption = 2.9201
```

Price a 3-Year Put Swaption with Receiving and Paying Legs Using an HW Interest-Rate Tree

This example shows how to price a 3-year put swaption with receiving and paying legs using an HW interest-rate tree with the following data.

```
Rates =0.075 * ones (10,1);
Compounding = 2;
StartDates = [datetime(2007,1,1);datetime(2007,7,1);datetime(2008,1,1);datetime(2008,7,1);dateti
EndDates = [datetime(2007,7,1);datetime(2008,1,1);datetime(2008,7,1);datetime(2009,1,1);datetime
ValuationDate = datetime(2007,1,1);
```

Define the `RatesSpec`.

```
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 1-year swap and 3-year swaption

```
ExerciseDates = datetime(2010,1,1);
SwapSettlement = ExerciseDates;
SwapMaturity   = datetime(2012,1,1);
```

```
Spread = 0;
SwapReset = [2 2]; % 1st column represents receiving leg, 2nd column represents paying leg
Principal = 100;
OptSpec = 'put';
Strike= 0.04;
Basis= [1 3];    % 1st column represents receiving leg, 2nd column represents paying leg
```

Price the swaption.

```
PriceSwaption = swaptionbyhw(HWTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity,'SwapReset', SwapReset, ...
'Basis', Basis,'Principal', Principal)
```

```
PriceSwaption = 2.9201
```

Input Arguments

`HWTree` — Interest-rate tree structure

structure

Interest-rate tree structure, specified by using `hwtree`.

Data Types: `struct`

`OptSpec` — Definition of option

character vector with values `'call'` or `'put'` | cell array of character vector with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-1 cell array of character vectors. For more information, see “More About” on page 11-1920.

Data Types: `char` | `cell`

`Strike` — Strike swap rate values

decimal

Strike swap rate values, specified as a `NINST`-by-1 vector.

Data Types: `double`

`ExerciseDates` — Exercise dates for swaption

datetime array | string array | date character vector

Exercise dates for the swaption, specified as a `NINST`-by-1 vector or a `NINST`-by-2 vector using a datetime array, string array, or date character vectors, depending on the option type.

- For a European option, `ExerciseDates` are a `NINST`-by-1 vector of exercise dates. Each row is the schedule for one option. When using a European option, there is only one `ExerciseDate` on the option expiry date.
- For an American option, `ExerciseDates` are a `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 `ValuationDate` of the tree and the single listed `ExerciseDate`.

To support existing code, `swaptionbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Spread` — Number of basis points over reference rate
numeric

Number of basis points over the reference rate, specified as a `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date (representing the settle date for each swap), specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date for every swaption is set to the `ValuationDate` of the HW tree. The swap argument `Settle` is ignored. The underlying swap starts at the maturity of the swaption.

To support existing code, `swaptionbyhw` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for swap
datetime array | string array | date character vector

Maturity date for each swap, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swaptionbyhw` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Price,PriceTree] = swaptionbyhw(HWTree,OptSpec, ExerciseDates,Spread,Settle,Maturity,'SwapReset',4,'Basis',5,'Principal',10000)`

`AmericanOpt` — Option type
`0` (European) (default) | integer with values `0` or `1`

(Optional) Option type, specified as the comma-separated pair consisting of `'AmericanOpt'` and `NINST`-by-`1` positive integer flags with values:

- `0` — European
- `1` — American

Data Types: `double`

`SwapReset` — Reset frequency per year for underlying swap
`1` (default) | numeric

Reset frequency per year for the underlying swap, specified as the comma-separated pair consisting of `'SwapReset'` and a `NINST`-by-`1` vector or `NINST`-by-`2` matrix representing the reset frequency per year for each leg. If `SwapReset` is `NINST`-by-`2`, the first column represents the receiving leg, while the second column represents the paying leg.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis representing the basis used when annualizing the input forward rate tree for each instrument, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` vector or `NINST`-by-`2` matrix representing the basis for each leg. If `Basis` is `NINST`-by-`2`, the first column represents the receiving leg, while the second column represents the paying leg.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

`Options` — Derivatives pricing options structure
structure

Derivatives pricing options structure, specified as the comma-separated pair consisting of `'Options'` and a structure obtained from using `derivset`.

Data Types: `struct`

Output Arguments

`Price` — Expected prices of swaptions at time 0

vector

Expected prices of the swaptions at time 0, returned as a `NINST`-by-`1` vector.

`PriceTree` — Tree structure of instrument prices

structure

Tree structure of instrument prices, returned as 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.

More About

Call 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.

Put Swaption

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.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swaptionbyhw` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hwtree` | `instswaption` | `swapbyhw` | `Swaption`

Topics

“Pricing Using Interest-Rate Tree Models” on page 2-81
“Calibrating Hull-White Model Using Market Data” on page 2-92
“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Swaption” on page 2-14
“Pricing Options Structure” on page A-2
“Understanding Interest-Rate Tree Models” on page 2-66
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

swaptionbylg2f

Price European swaption using Linear Gaussian two-factor model

Syntax

```
Price = swaptionbylg2f(ZeroCurve,a,b,sigma,eta,rho,Strike,ExerciseDate,
Maturity)
Price = swaptionbylg2f( ___ ,Name,Value)
```

Description

`Price = swaptionbylg2f(ZeroCurve,a,b,sigma,eta,rho,Strike,ExerciseDate, Maturity)` returns the European swaption price for a two-factor additive Gaussian interest-rate model.

Note Alternatively, you can use the `Swaption` object to price swaption instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = swaptionbylg2f( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a European Swaption Using a Linear Gaussian Two-Factor Model

Define the `ZeroCurve`, `a`, `b`, `sigma`, `eta`, and `rho` parameters to compute the price of the swaption.

```
Settle = datetime(2007,12,15);

ZeroTimes = [3/12 6/12 1 5 7 10 20 30]';
ZeroRates = [0.033 0.034 0.035 0.040 0.042 0.044 0.048 0.0475]';
CurveDates = daysadd(Settle,360*ZeroTimes,1);

irdc = IRDataCurve('Zero',Settle,CurveDates,ZeroRates);

a = .07;
b = .5;
sigma = .01;
eta = .006;
rho = -.7;

Reset = 1;
ExerciseDate = daysadd(Settle,360*5,1);
Maturity = daysadd(ExerciseDate,360*[3;4],1);
Strike = .05;

Price = swaptionbylg2f(irdc,a,b,sigma,eta,rho,Strike,ExerciseDate,Maturity,'Reset',Reset)
```

```
Price = 2×1
```

```
    1.1869
    1.5590
```

Input Arguments

`ZeroCurve` — Zero-curve for Linear Gaussian two-factor model

structure

Zero-curve for the Linear Gaussian two-factor model, specified using `IRDataCurve` or `RateSpec`.

Data Types: `struct`

`a` — Mean reversion for first factor for Linear Gaussian two-factor model

scalar

Mean reversion for first factor for the Linear Gaussian two-factor model, specified as a scalar.

Data Types: `single` | `double`

`b` — Mean reversion for second factor for Linear Gaussian two-factor model

scalar

Mean reversion for second factor for the Linear Gaussian two-factor model, specified as a scalar.

Data Types: `single` | `double`

`sigma` — Volatility for first factor for Linear Gaussian two-factor model

scalar

Volatility for first factor for the Linear Gaussian two-factor model, specified as a scalar.

Data Types: `single` | `double`

`eta` — Volatility for second factor for Linear Gaussian two-factor model

scalar

Volatility for second factor for the Linear Gaussian two-factor model, specified as a scalar.

Data Types: `single` | `double`

`rho` — Scalar correlation of the factors

scalar

Scalar correlation of the factors, specified as a scalar.

Data Types: `single` | `double`

`Strike` — Swaption strike price

nonnegative integer | vector of nonnegative integers

Swaption strike price, specified as a nonnegative integer using a `NumSwaptions`-by-`1` vector.

Data Types: `single` | `double`

`ExerciseDate` — Swaption exercise dates

datetime array | string array | date character vector

Swaption exercise dates, specified as a `NumSwaptions`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swaptionbylg2f` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Underlying swap maturity date
datetime array | string array | date character vector

Underlying swap maturity date, specified using a `NumSwaptions`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `swaptionbylg2f` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = swaptionbylg2f(irdc,a,b,sigma,eta,rho,Strike,ExerciseDate,Maturity,'Reset',1,'Notional',100,'OptSpec','call')`

`Reset` — Frequency of swaption payments per year
`2` (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Frequency of swaption payments per year, specified as the comma-separated pair consisting of `'Reset'` and positive integers for the values `1,2,4,6,12` in a `NumSwaptions`-by-`1` vector.

Data Types: `single` | `double`

`Notional` — Notional value of swaption
`100` (default) | nonnegative integer | vector of nonnegative integers

Notional value of swaption, specified as the comma-separated pair consisting of `'Notional'` and a nonnegative integer using a `NumSwaptions`-by-`1` vector of notional amounts.

Data Types: `single` | `double`

`OptSpec` — Option specification for the swaption
`'call'` (default) | character vector with value of `'call'` or `'put'` | cell array of character vectors with values of `'call'` or `'put'`

Option specification for the swaption, specified as the comma-separated pair consisting of `'OptSpec'` and a character vector or a `NumSwaptions`-by-`1` cell array of character vectors with a value of `'call'` or `'put'`.

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.

Data Types: `char` | `cell`

Output Arguments

`Price` — Swaption price

scalar | vector

Swaption price, returned as a scalar or an `NumSwaptions`-by-`1` vector.

More About

Call 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.

Put Swaption

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.

Algorithms

The following defines the swaption price for a two-factor additive Gaussian interest-rate model, given the `ZeroCurve`, `a`, `b`, `sigma`, `eta`, and `rho` parameters:

$$r(t) = x(t) + y(t) + \phi(t)$$

$$dx(t) = -ax(t)dt + \sigma dW_1(t), \quad x(0) = 0$$

$$dy(t) = -by(t)dt + \eta dW_2(t), \quad y(0) = 0$$

where $dW_1(t)dW_2(t) = \rho dt$ is a two-dimensional Brownian motion with correlation ρ and ϕ is a function chosen to match the initial zero curve.

Version History

Introduced in R2013a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swaptionbylg2f` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

See Also

`capbylg2f` | `floorbylg2f` | `LinearGaussian2F` | `Swaption`

Topics

“Price Swaptions with Interest-Rate Models Using Simulation” on page 2-100
“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Swaption” on page 2-14
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

swaptionbynormal

Price swaptions using Normal or Bachelier option pricing model

Syntax

```
Price = swaptionbynormal(RateSpec,OptSpec,Strike,Settle,ExerciseDates,
Maturity,Volatility)
Price = swaptionbynormal( ___ ,Name,Value)
```

Description

`Price = swaptionbynormal(RateSpec,OptSpec,Strike,Settle,ExerciseDates, Maturity,Volatility)` prices swaptions using the Normal or Bachelier option pricing model.

Note Alternatively, you can use the `Swaption` object to price swaption instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`Price = swaptionbynormal( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Swaption Using the Normal Model

Define the zero curve, and create a `RateSpec`.

```
Settle = datetime(2016,1,20);
ZeroTimes = [.5 1 2 3 4 5 7 10 20 30]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = datemnth(Settle,12*ZeroTimes);
RateSpec = intenvset('StartDate',Settle,'EndDates',ZeroDates,'Rates',ZeroRates)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [10x1 double]
            Rates: [10x1 double]
         EndTimes: [10x1 double]
       StartTimes: [10x1 double]
         EndDates: [10x1 double]
       StartDates: 736349
    ValuationDate: 736349
            Basis: 0
     EndMonthRule: 1
```

Define the swaption.

```
ExerciseDate = datetime(2021,1,20);
Maturity = datetime(2026,1,20);
```

```
OptSpec = 'call';
LegReset = [1 1];
```

Compute the par swap rate.

```
[~,ParSwapRate] = swapbyzero(RateSpec,[NaN 0],Settle,Maturity,'LegReset',LegReset)
```

```
ParSwapRate = 0.0216
```

```
Strike = ParSwapRate;
BlackVol = .3;
NormalVol = BlackVol*ParSwapRate;
```

Price with Black volatility.

```
Price = swaptionbyblk(RateSpec,OptSpec,Strike,Settle,ExerciseDate,Maturity,BlackVol)
```

```
Price = 5.9756
```

Price with Normal volatility.

```
Price_Normal = swaptionbynormal(RateSpec,OptSpec,Strike,Settle,ExerciseDate,Maturity,NormalVol)
```

```
Price_Normal = 5.5537
```

Price a Swaption with a Receiving and Paying Leg Using the Normal Model

Create a RateSpec.

```
Rate = 0.06;
Compounding  = -1;
ValuationDate = datetime(2010,1,1);
EndDates = datetime(2020,1,1);
Basis = 1;
RateSpec = intenvset('ValuationDate', ValuationDate,'StartDates', ValuationDate, ...
'EndDates', EndDates, 'Rates', Rate, 'Compounding', Compounding, 'Basis', Basis);
```

Define the swaption.

```
ExerciseDate = datetime(2021,1,20);
Maturity = datetime(2026,1,20);
Settle = datetime(2010,1,1);
OptSpec = 'call';
Strike = .09;
NormalVol = .03;
Reset = [1 4];  % 1st column represents receiving leg, 2nd column represents paying leg
Basis = [1 7];  % 1st column represents receiving leg, 2nd column represents paying leg
```

Price with Normal volatility.

```
Price_Normal = swaptionbynormal(RateSpec,OptSpec,Strike,Settle,ExerciseDate,Maturity,NormalVol,'
```

```
Price_Normal = 5.9084
```

Price a Swaption Using swaptionbynormal and Compare to swaptionbyblk

Define the RateSpec.

```
Settle = datetime(2016,1,20);
ZeroTimes = [.5 1 2 3 4 5 7 10 20 30]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = datemnth(Settle,12*ZeroTimes);
RateSpec = intenvset('StartDate',Settle,'EndDates',ZeroDates,'Rates',ZeroRates)
```

```
RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [10x1 double]
            Rates: [10x1 double]
         EndTimes: [10x1 double]
       StartTimes: [10x1 double]
         EndDates: [10x1 double]
       StartDates: 736349
    ValuationDate: 736349
            Basis: 0
     EndMonthRule: 1
```

Define the swaption instrument and price with swaptionbyblk.

```
ExerciseDate = datetime(2021,1,20);
Maturity = datetime(2026,1,20);
OptSpec = 'call';

[~,ParSwapRate] = swapbyzero(RateSpec,[NaN 0],Settle,Maturity,'StartDate',ExerciseDate)
```

```
ParSwapRate = 0.0326
```

```
Strike = ParSwapRate;
BlackVol = .3;
NormalVol = BlackVol*ParSwapRate;

Price = swaptionbyblk(RateSpec,OptSpec,Strike,Settle,ExerciseDate,Maturity,BlackVol)
```

```
Price = 3.6908
```

Price the swaption instrument using swaptionbynormal.

```
Price_Normal = swaptionbynormal(RateSpec,OptSpec,Strike,Settle,ExerciseDate,Maturity,NormalVol)
```

```
Price_Normal = 3.7602
```

Price the swaption instrument using swaptionbynormal for a negative strike.

```
 Price_Normal = swaptionbynormal(RateSpec,OptSpec,-.005,Settle,ExerciseDate,Maturity,NormalVol)
```

```
Price_Normal = 16.3674
```

Input Arguments

RateSpec — Interest-rate term structure
structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

If the discount curve for the paying leg is different than the receiving leg, `RateSpec` can be a `NINST`-by-2 input variable of `RateSpecs`, with the second input being the discount curve for the paying leg. If only one curve is specified, then it is used to discount both legs.

Data Types: `struct`

`OptSpec` — Definition of option
character vector with values `'call'` or `'put'` | cell array of character vector with values `'call'` or `'put'`

Definition of the option as `'call'` or `'put'`, specified as a `NINST`-by-1 cell array of character vectors.

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.

Data Types: `char` | `cell`

`Strike` — Strike swap rate values
decimal

Strike swap rate values, specified as a `NINST`-by-1 vector of decimal values.

Data Types: `double`

`Settle` — Settlement date
datetime array | string array | date character vector

Settlement date (representing the settle date for each swaption), specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors. `Settle` must not be later than `ExerciseDates`.

To support existing code, `swaptionbynormal` also accepts serial date numbers as inputs, but they are not recommended.

The `Settle` date input for `swaptionbynormal` is the valuation date on which the swaption (an option to enter into a swap) is priced. The swaption buyer pays this price on this date to hold the swaption.

`ExerciseDates` — Dates on which swaption expires and underlying swap starts
datetime array | string array | date character vector

Dates on which the swaption expires and the underlying swap starts, specified as a `NINST`-by-1 vector using a datetime array, string array, or date character vectors. There is only one `ExerciseDate` on the option expiry date. This is also the `StartDate` of the underlying forward swap.

To support existing code, `swaptionbynormal` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date for each forward swap
datetime array | string array | date character vector

Maturity date for each forward swap, specified as a NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, swaptionbynormal also accepts serial date numbers as inputs, but they are not recommended.

Volatility — Volatilities values
numeric

Volatilities values (for normal volatility), specified as a NINST-by-1 vector of numeric values.

For more information on the Normal model, see “Work with Negative Interest Rates Using Functions” on page 2-18.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: Price = swaptionbynormal(OISCurve,OptSpec,Strike,Settle,ExerciseDate,Maturity,NormalVol,'Reset',4)

Reset — Reset frequency per year for underlying forward swap
1 (default) | numeric

Reset frequency per year for the underlying forward swap, specified as the comma-separated pair consisting of 'Reset' and a NINST-by-1 vector or NINST-by-2 matrix representing the reset frequency per year for each leg. If Reset is NINST-by-2, the first column represents the receiving leg, while the second column represents the paying leg.

Data Types: double

Basis — Day-count basis of instrument
0 (actual/actual) (default) | integer from 0 to 13

Day-count basis of the instrument representing the basis used when annualizing the input term structure, specified as the comma-separated pair consisting of 'Basis' and a NINST-by-1 vector or NINST-by-2 matrix representing the basis for each leg. If Basis is NINST-by-2, the first column represents the receiving leg, while the second column represents the paying leg.

Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)

- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount
`100` (default) | numeric

Notional principal amount, specified as the comma-separated pair consisting of `'Principal'` and a `NINST`-by-`1` vector.

Data Types: `double`

`ProjectionCurve` — Rate curve used in projecting future cash flows
if `ProjectionCurve` is not specified, then `RateSpec` is used both for discounting cash flows and projecting future cash flows (default) | structure

The rate curve to be used in projecting the future cash flows, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a rate curve structure. This structure must be created using `intenvset`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `struct`

Output Arguments

`Price` — Prices for swaptions at time 0
vector

Prices for the swaptions at time 0, returned as a `NINST`-by-`1` vector of prices.

More About

Call 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.

Put Swaption

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.

Version History

Introduced in R2017a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `swaptionbynormal` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`swaptionbyblk` | `capbynormal` | `floorbynormal` | `intenvset` | `Swaption`

Topics

“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Swaption” on page 2-14
“Work with Negative Interest Rates Using Functions” on page 2-18
“Supported Interest-Rate Instrument Functions” on page 2-3
“Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73

External Websites

How to Price Interest Rate Options with Negative Interest Rates (3 min 05 sec)

time2date

Dates from time and frequency

Syntax

```
Maturity = time2date(Settle,Times)
Maturity = time2date( ___ ,Compounding,Basis,EndMonthRule)
```

Description

`Maturity = time2date(Settle,Times)` computes dates corresponding to compounded rate quotes between `Settle` and time factors.

`Maturity = time2date( ___ ,Compounding,Basis,EndMonthRule)` add additional optional arguments.

Examples

Dates From Time and Frequency

This example shows how to compute dates from time and frequency.

```
Settle = datetime(2002,9,1);
Maturity = [datetime(2005,8,31) ; datetime(2006,2,28) ; datetime(2006,6,15) ; datetime(2006,12,31
Compounding = 2;
Basis = 0;
EndMonthRule = 1;
Times = date2time(Settle, Maturity, Compounding, Basis, EndMonthRule)
```

```
Times = 4×1

    5.9945
    6.9945
    7.5738
    8.6576
```

```
Dates_calc = time2date(Settle, Times, Compounding, Basis, EndMonthRule)
```

```
Dates_calc = 4x1 datetime
   31-Aug-2005
   28-Feb-2006
   15-Jun-2006
   31-Dec-2006
```

Input Arguments

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `time2date` also accepts serial date numbers as inputs, but they are not recommended.

`Times` — Time factors corresponding to `Compounding`
vector

Time factors corresponding to `Compounding`, specified as an `N`-by-`1` vector.

Data Types: `double`

`Compounding` — Rate at which the input zero rates were compounded when annualized
`2` (default) | integer with value of `1`, `2`, `3`, `4`, `6`, `12`, `365`, or `-1`

(Optional) Rate at which the input zero rates were compounded when annualized, specified as a scalar integer value.

- If `Compounding` = `1`, `2`, `3`, `4`, `6`, `12`:

  `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units; for example, `T = F` is one year.
- If `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.
- If `Compounding` = `−1`:

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

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis, specified as a scalar or an `N`-by-`1` vector using the following values:

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

- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `[0,1]`

(Optional) End-of-month rule flag, specified as a scalar or an `N`-by-`1` vector of end-of-month rules.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`Maturity` — Maturity dates corresponding to compounded rate quotes between `Settle` and time factors
serial date number

Maturity dates corresponding to compounded rate quotes between `Settle` and time factors, returned as a scalar or an `N`-by-`1` vector.

The `time2date` function is the inverse of `date2time`.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `time2date` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cftimes` | `date2time` | `cfamounts` | `disc2rate` | `rate2disc`

Topics

“Modeling the Interest-Rate Term Structure” on page 2-57
“Interest-Rate Term Conversions” on page 2-53
“Interest Rates Versus Discount Factors” on page 2-48
“Understanding the Interest-Rate Term Structure” on page 2-48

treepath

Entries from node of recombining binomial tree

Syntax

```
Values = treepath(Tree,BranchList)
```

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.

Examples

Extract Entries of a Node of a Recombining Binomial Tree

Create a BDT tree by loading the example file.

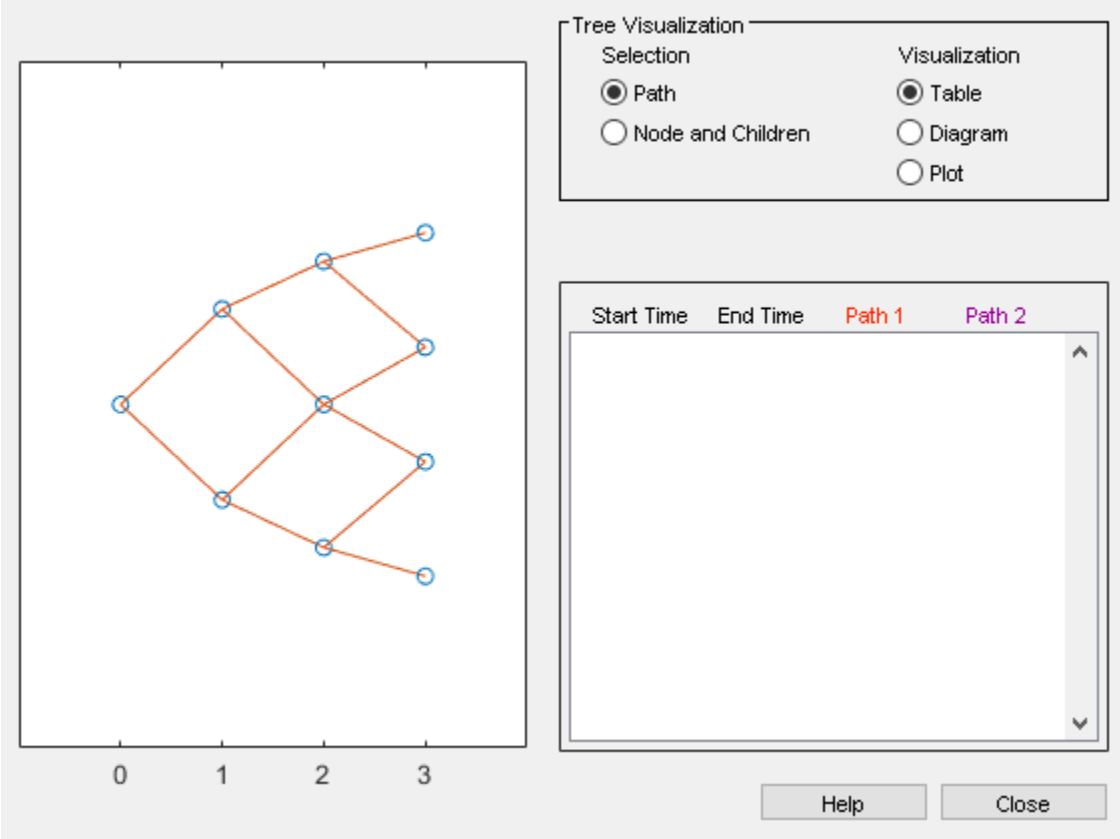
```
load deriv.mat;
FwdRates = treepath(BDTTree.FwdTree, [1 2 1])
```

```
FwdRates = 4×1

    1.1000
    1.0979
    1.1377
    1.1183
```

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

You can visualize this with the `treeviewer` function.

```
treeviewer(BDTTree)
```



Input Arguments

Tree — Recombining binomial tree or trinomial tree

struct

Recombining binomial tree or trinomial tree specified as a struct that is created using one of the following functions:

- `hjmtree`
- `bdttree`
- `hwtree`
- `bktree`
- `crrtree`
- `eqptree`
- `lrtree`
- `cirtree`
- `stttree`
- `itttree`

Data Types: `struct`

`BranchList` — Number of paths by path length
matrix

Number of paths (`NUMPATHS`) by path length (`PATHLENGTH`), specified as a matrix containing the sequence of branchings.

Data Types: `double`

Output Arguments

`Values` — Retrieved entries of a recombining tree
matrix

Retrieved entries of a recombining tree, returned a number of values (`NUMVALS`)-by-`NUMPATHS` matrix.

Version History

Introduced before R2006a

See Also

`mktree` | `treeshape`

Topics

“Graphical Representation of Trees” on page 2-219
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Overview of Interest-Rate Tree Models” on page 2-44

treeshape

Shape of recombining binomial tree

Syntax

```
[NumLevels,NumPos,IsPriceTree] = treeshape(Tree)
```

Description

`[NumLevels,NumPos,IsPriceTree] = treeshape(Tree)` returns information on a recombining binomial tree's shape.

Examples

Determine Shape of Recombining Binomial Tree

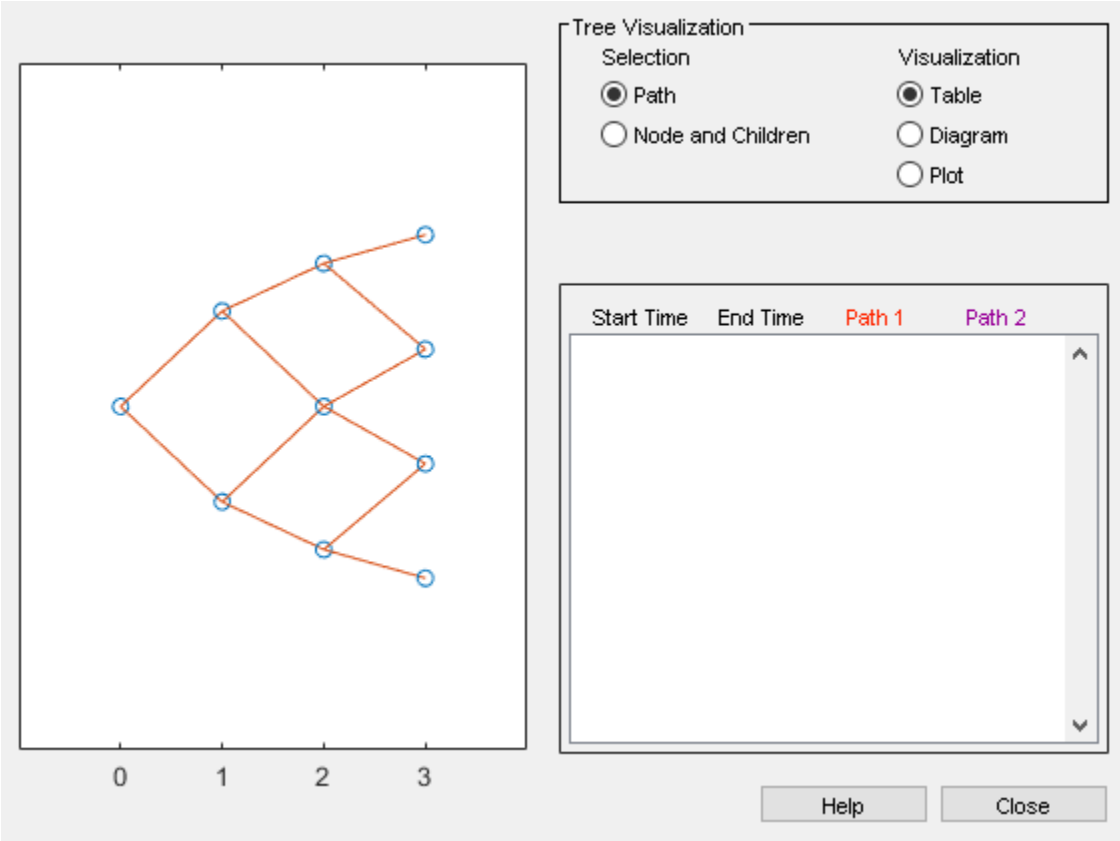
This example shows how to obtain information on a recombining binomial tree's shape.

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)
```



Use `treeshape` to display the shape of the binomial tree.

```
[NumLevels, NumPos, IsPriceTree] = treeshape(BDTTree.FwdTree)
```

```
NumLevels = 4
```

```
NumPos = 1×4

     1     1     1     1
```

```
IsPriceTree = logical
   0
```

Input Arguments

Tree — Recombining binomial tree

struct

Recombining binomial tree, specified as a struct that is created using one of the following functions:

- `hjmtree`
- `bdttree`
- `hwtree`

- `bktree`
- `cirtree`

Data Types: `struct`

Output Arguments

`NumLevels` — Number of time levels of tree
numeric

Number of time levels of tree, returned as a numeric.

`NumPos` — Length of the state vectors in each level
vector

Length of the state vectors in each level, returned as a `1`-by-`NUMLEVELS` vector.

`IsPriceTree` — Indicates if final horizontal branch is present in the tree
logical

Indicates if final horizontal branch is present in the tree, returned as a Boolean.

Version History

Introduced before R2006a

See Also

`mktree` | `treepath`

Topics

“Graphical Representation of Trees” on page 2-219
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Overview of Interest-Rate Tree Models” on page 2-44

treeviewer

Tree information

Syntax

```
treeviewer(Tree)
treeviewer(PriceTree,InstSet)
treeviewer(CFTree,InstSet)
```

Description

`treeviewer(Tree)` displays an interest rate tree, stock price tree, 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 on page 11-1947 for a description.) If you do not provide the optional `InstSet` input argument, the instruments are identified by their sequence number in the instrument set. (See Example 6 on page 11-1950 for a description.)

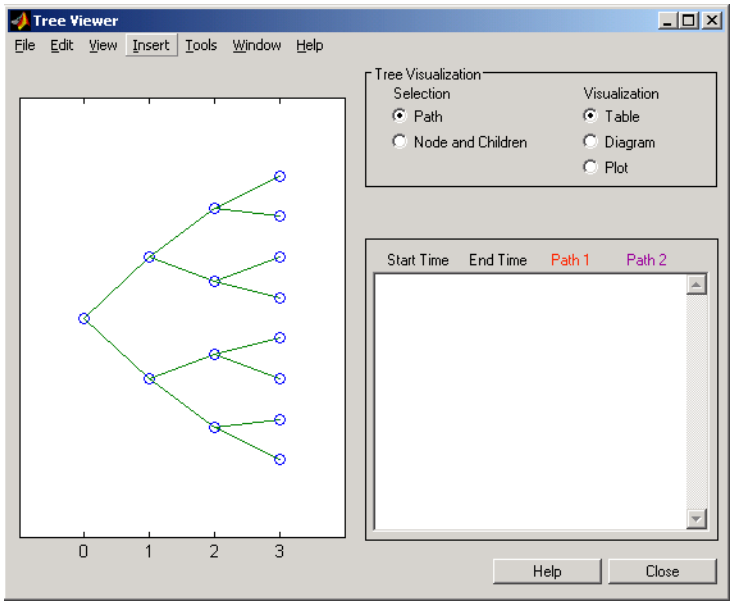
`treeviewer(CFTree,InstSet)` displays a cash flow tree that has been created with `swapbybdt` or `swapbyhjm`. If you provide the name of an instrument set (`InstSet`) containing cash flow names, the `treeviewer` display identifies the instrument being displayed with its name. (See Example 3 on page 11-1947 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 on page 11-1950 for a description.)

Examples

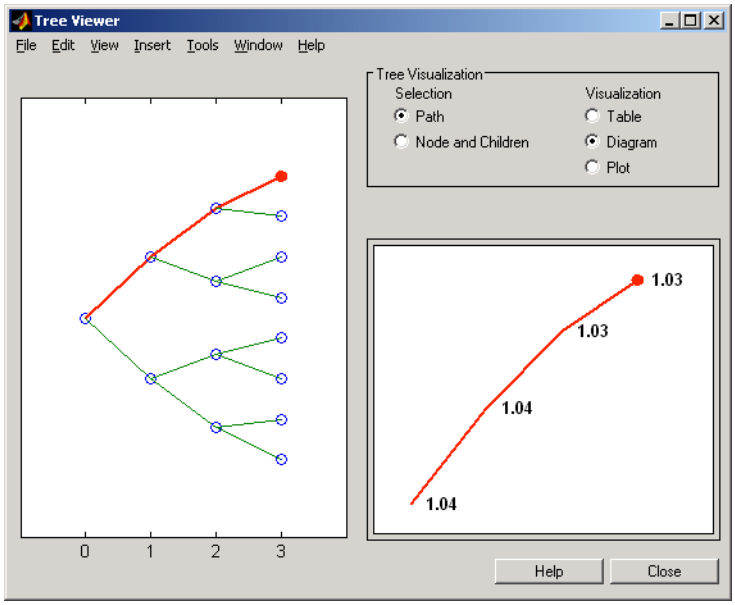
Display an HJM Interest-Rate Tree

```
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.

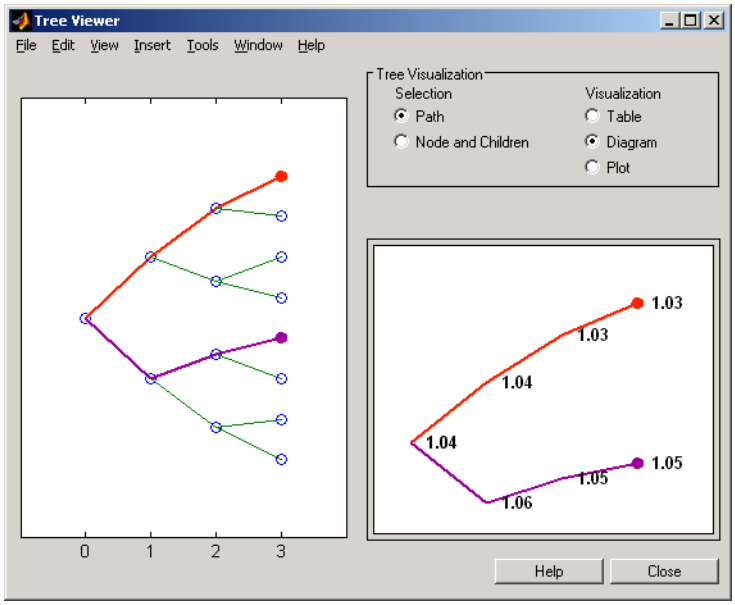


To visualize the actual interest-rate tree, go to the **Tree Visualization** pane and click**Path**(the default) and **Diagram**. Now, select the first path by clicking the last node (`t = 3`) of the upper branch.



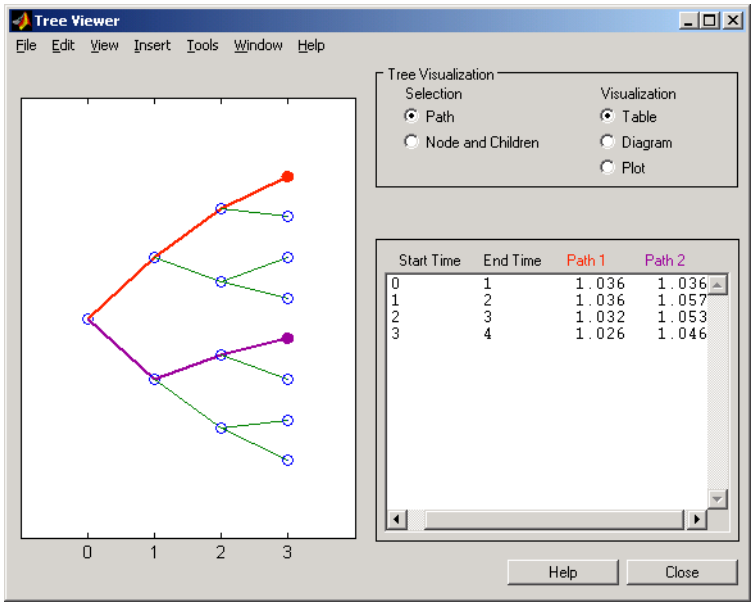
The entire upper path is highlighted in red.

To complete the process, select a second path by clicking the last node (`t = 3`) of another branch. The second path is highlighted in purple. The final display looks like this.

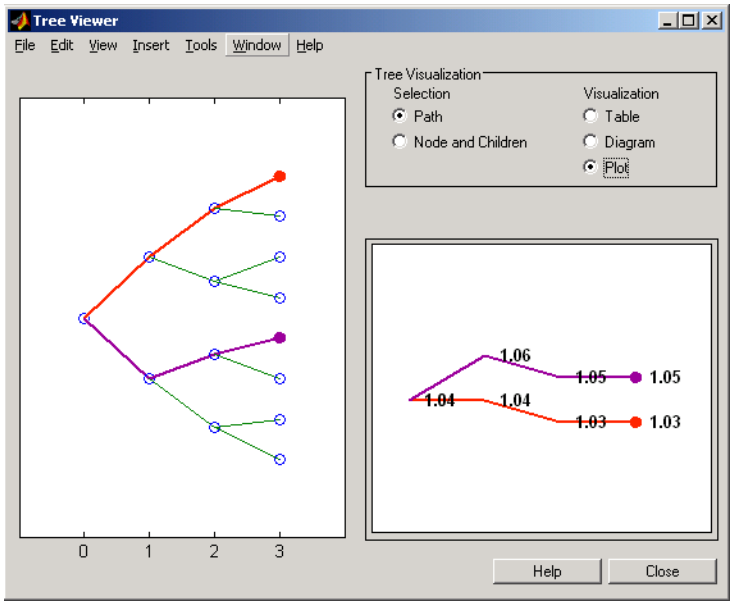


Alternative Forms of Display

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.

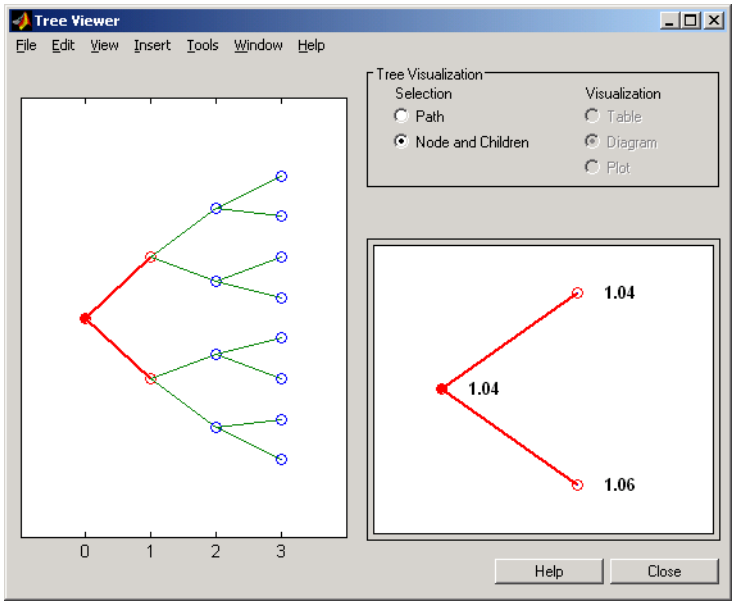


To see a plot of interest rates along the chosen branches, click **Path** and **Plot** in the **Tree Visualization** pane.



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.

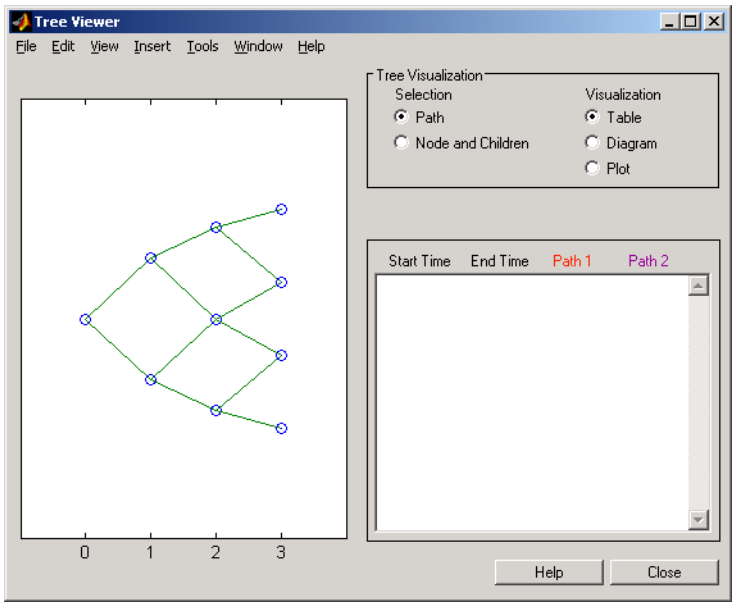


With **Node and Children** selected, the choices under **Visualization** are unavailable.

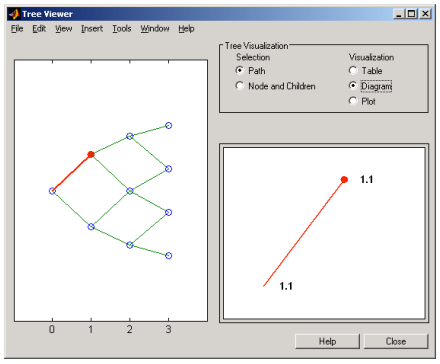
Display a BDT Interest-Rate Tree

```
load deriv.mat
treeviewer(BDTTree)
```

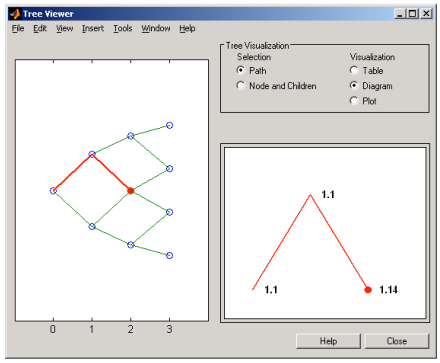
The `treeviewer` function displays the structure of a BDT tree in the left pane. The tree visualization in the right pane is blank.



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 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.



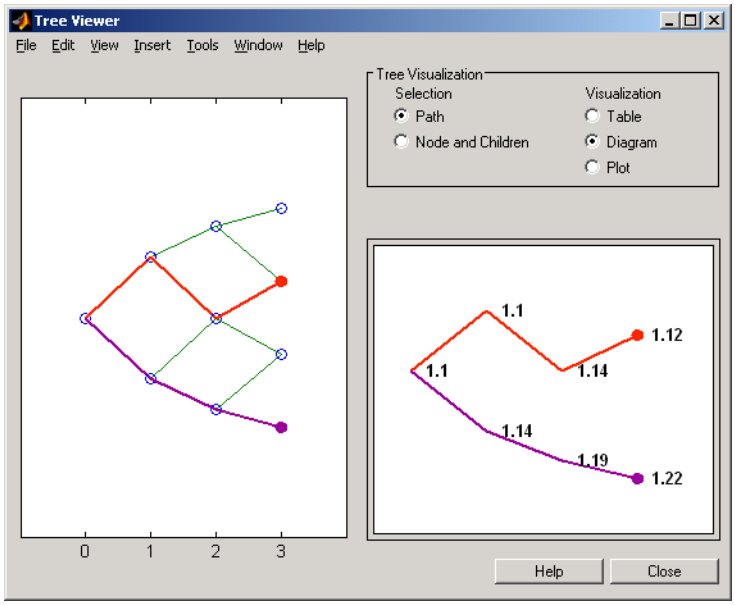
t = 1



t = 2

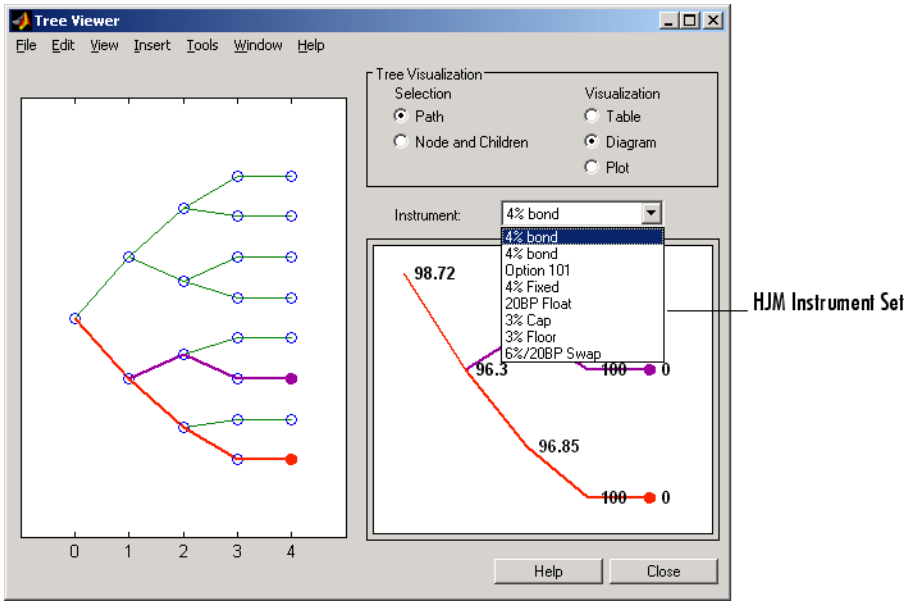
Continue clicking all nodes in succession until you reach the end of the branch. The entire path you have selected is highlighted in red.

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. The second branch is highlighted in purple. The final display looks like this.



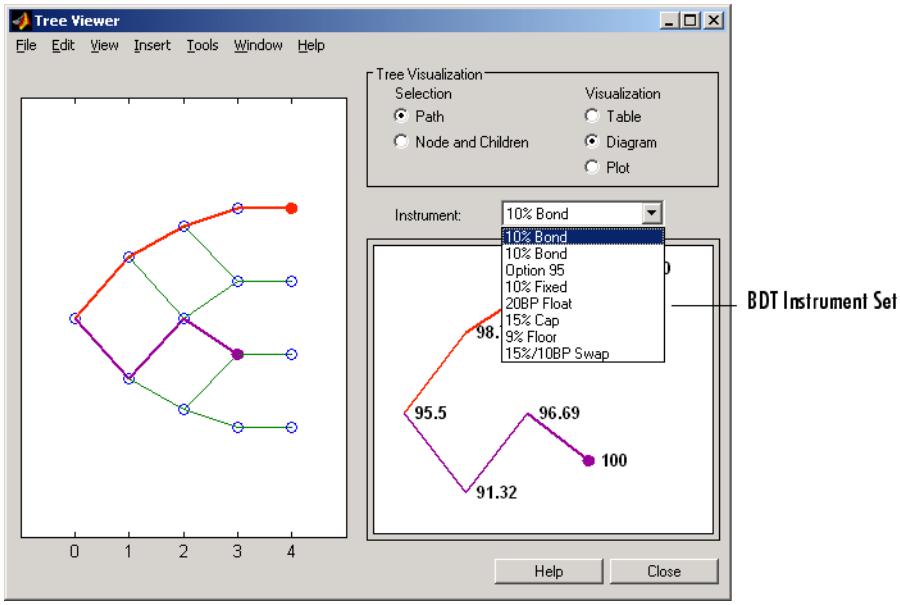
Display an HJM Price Tree for Named Instruments

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



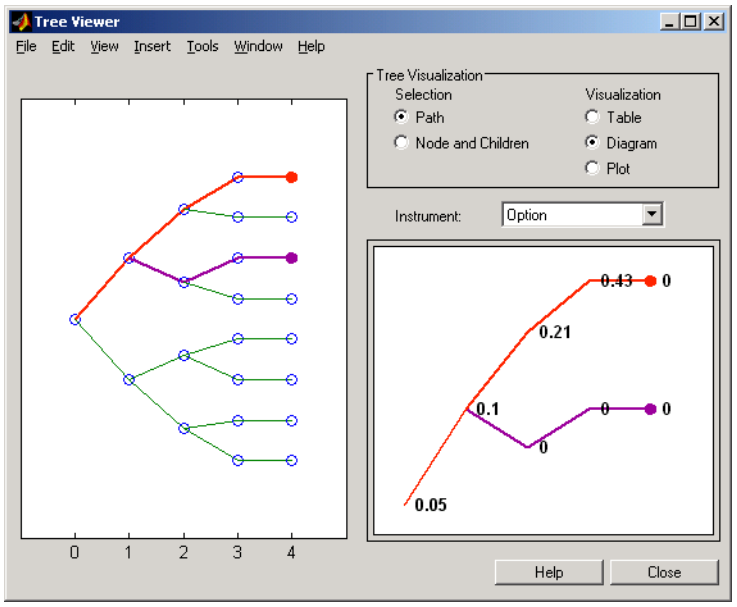
Display a BDT Price Tree for Named Instruments

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



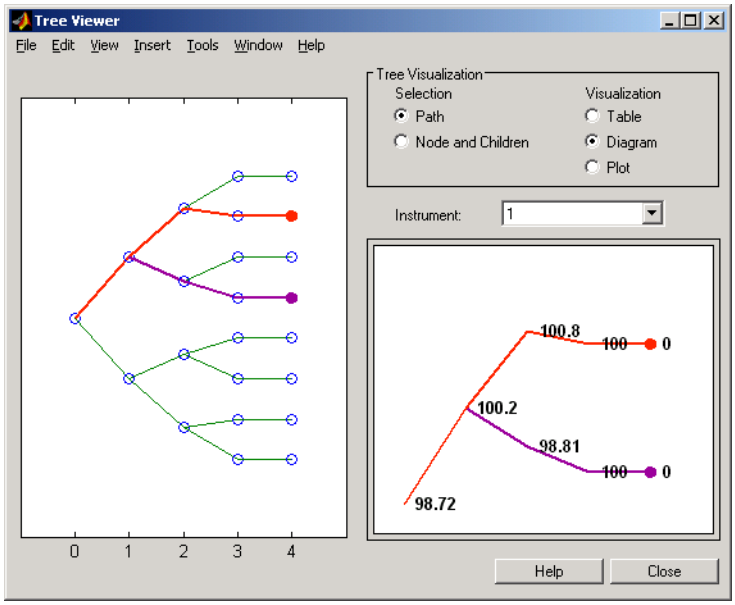
Display an HJM Price Tree with Renamed Instruments

```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet);
Names = {'Bond1', 'Bond2', 'Option', 'Fixed','Float', 'Cap',...
'Floor', 'Swap'};
treeviewer(PriceTree, Names)
```



Display an HJM Price Tree Using Default Instrument Names (Numbers)

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



Input Arguments

Tree — Interest-rate tree, stock price tree, or money-market tree

interest-rate tree structure | stock price tree structure | money-market tree structure

Interest-rate tree, stock price tree, or money-market tree, specified using the associated tree function.

Interest-rate trees:

- Black-Derman-Toy (`BDTTree`) obtained from `bdttree`
- Black-Karasinski (`BKTree`) obtained from `bktree`
- Heath-Jarrow-Morton (`HJMTree`) obtained from `hjmtree`
- Hull-White (`HWTree`) obtained from `hwtree`
- Cox-Ingersoll-Ross (`CIRTree`) obtained from `cirtree`

Money market trees:

- Black-Derman-Toy (`BDTMMktTree`) obtained from `mmktbybdt` for a money-market tree from a BDT interest-rate tree.
- Heath-Jarrow-Morton (`HJMMMktTree`) obtained from `mmktbyhjm` for 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`) obtained from `crrtree`
- Implied Trinomial tree (`ITTTree`) obtained from `itttree`
- Standard Trinomial tree (`STTTree`) obtained from `sttree`

- Leisen-Reimer stock tree (`LRTree`) obtained from `lrtree`
- Equal probabilities (`EQPTree`) obtained from `eqptree`

Cash flow trees:

- Black-Derman-Toy (`BDTCFTree`) obtained as output from the swap function `swapbybdt`
- Heath-Jarrow-Morton (`HJMCFTree`) obtained as output from the swap function `swapbyhjm`

> **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.

Data Types: `struct`

`PriceTree` — Tree structure of instrument prices
structure

Tree structure of instrument prices, specified as:

- Black-Derman-Toy (`BDTPriceTree`) obtained from the portfolio function `bdtprice` or the individual functions, such as `bondbybdt`, `capbybdt`, and so on.
- Black-Karasinski (`BKPriceTree`) obtained from the portfolio function `bkprice` or the individual functions, such as `bondbybk`, `capbybk`, and so on.
- Cox-Ingersoll-Ross (CIRPriceTree) obtained from the portfolio function `cirprice` or the individual functions, such as `bondbycir`, `capbycir`, and so on.
- Heath-Jarrow-Morton (`HJMPriceTree`) obtained from the portfolio function `hjmprice` or the individual functions, such as `bondbyhjm`, `capbyhjm`, and so on.
- Hull-White (`HWPriceTree`) obtained from the portfolio function `hwprice` or the individual functions, such as `bondbyhw`, `capbyhw`, and so on.
- Leisen-Reimer (`LRPriceTree`) obtained from the individual function `optstockbylr`.
- Cox-Ross-Rubinstein (`CRRPriceTree`) obtained from the portfolio function `crrprice` or the individual functions, such as `asianbycrr`, `barrierbycrr`, and so on.
- Equal probabilities (`EQPPriceTree`) obtained from the portfolio function `eqpprice` or the individual functions, such as `asianbyeqp`, `barrierbyeqp`, and so on.
- Implied Trinomial tree (`ITTPriceTree`) obtained from the portfolio function `ittprice` or the individual functions, such as `asianbyitt`, `barrierbyitt`, and so on.
- Standard trinomial tree (`STTPriceTree`) obtained from the portfolio function `sttprice` or the individual functions, such as `asianbystt`, `barrierbystt`, and so on.

Data Types: `struct`

`CFTree` — Tree of swap cash flows
structure

`CFTree` is a tree of swap cash flows, specified when you create cash flow trees by executing the Black-Derman-Toy (obtained as output from the swap function `swapbybdt`) and Heath-Jarrow-Morton (`swapbyhjm`) swap functions. (Black-Derman-Toy cash flow trees contain only `NaN`s.)

Data Types: `struct`

`InstSet` — Variable containing a collection of instruments whose prices or cash flows are contained in a tree
structure

(Optional) Variable containing a collection of instruments whose prices or cash flows are contained in a tree, specified using `instadd`. 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.

Data Types: `struct`

More About

Treeviewer Conventions

`treeviewer` price tree diagrams follow the convention that increasing prices appear on the upper branch of a tree and, so, 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).

Using Treeviewer

`treeviewer` provides an interactive display of prices or interest rates.

The `treeviewer` 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, HW, and CIR 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.
  ```

Note The **Help** button is not available for `treeviewer` in MATLAB Online.

Version History

Introduced before R2006a

See Also

`bdttree` | `bktree` | `cirtree` | `eqptree` | `hjmtree` | `hwtree` | `instadd` | `itttree` | `stttree` | `lrtree` | `mmktbybdt` | `mmktbyhjm` | `swapbybdt` | `swapbyhjm`

Topics

“Graphical Representation of Trees” on page 2-219
“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Overview of Interest-Rate Tree Models” on page 2-44

trintreepath

Entries from node of recombining trinomial tree

Syntax

```
Values = trintreepath(TrinTree,BranchList)
```

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.

Examples

Extract Entries of a Node of a Recombining Trinomial Tree

Create a `HW` tree by loading the example file.

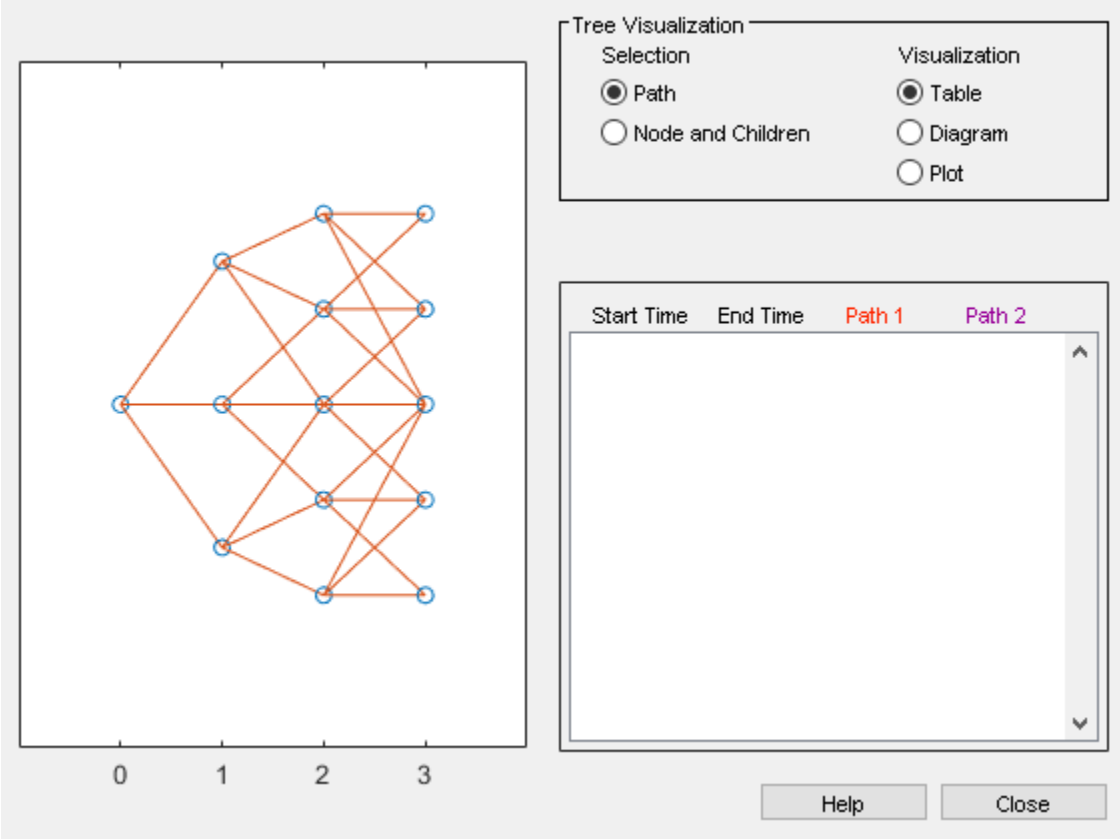
```
load deriv.mat;
FwdRates = trintreepath(HWTree, [1 2 3])
```

```
FwdRates = 4×1

    1.0279
    1.0528
    1.0652
    1.0591
```

This 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.

You can visualize this with the `treeviewer` function.

```
treeviewer(HWTree)
```



Input Arguments

`TrinTree` — Recombining price or interest-rate trinomial tree

struct

Recombining price or interest-rate trinomial tree, specified as a struct that is created using one of the following functions:

- `hjmtree`
- `bdttree`
- `hwtree`
- `bktree`
- `cirtree`

Data Types: `struct`

`BranchList` — Number of paths by path length

matrix

Number of paths (`NUMPATHS`) by path length (`PATHLENGTH`), specified as a matrix containing the sequence of branchings.

Data Types: `double`

Output Arguments

`Values` — Retrieved entries of a recombining tree

matrix

Retrieved entries of a recombining tree, returned a number of values (`NUMVALS`)-by-`NUMPATHS` matrix.

Version History

Introduced before R2006a

See Also

`mktrintree`

Topics

“Graphical Representation of Trees” on page 2-219
“Overview of Interest-Rate Tree Models” on page 2-44

trintreeshape

Shape of recombining trinomial tree

Syntax

```
[NumLevels,NumPos,NumStates] = treeshape(TrinTree)
```

Description

`[NumLevels,NumPos,NumStates] = treeshape(TrinTree)` returns information on a recombining trinomial tree's shape.

Examples

Determine Shape of Recombining Trinomial Tree

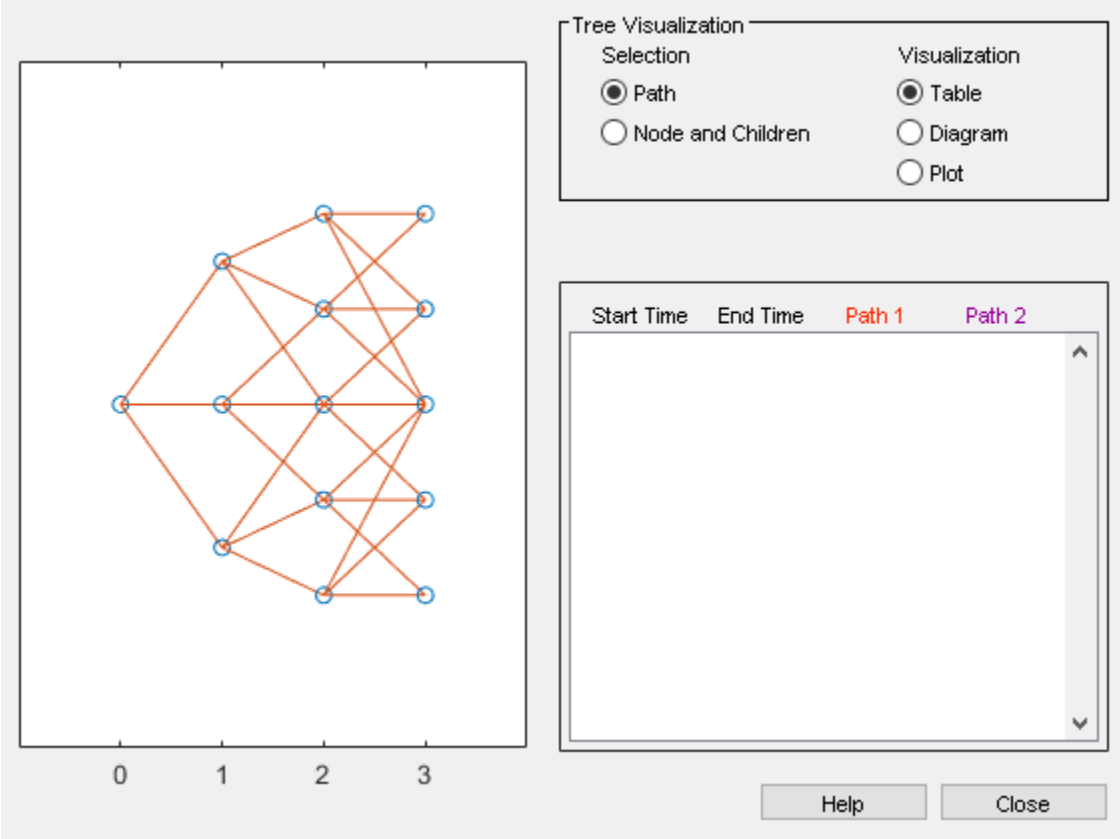
This example shows how to obtain information on a recombining trinomial tree's shape.

Create a `HW` 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)
```



Use `trintreeshape` to display the shape of the binomial tree.

```
[NumLevels, NumPos, IsPriceTree] = trintreeshape(HWTree)
```

```
NumLevels = 4
```

```
NumPos = 1×4
```

```
     1     1     1     1
```

```
IsPriceTree = 1×4
```

```
     1     3     5     5
```

Input Arguments

`TrinTree` — Recombining price or interest-rate trinomial tree

struct

Recombining price or interest-rate trinomial tree, specified as a struct that is created using one of the following functions:

- `hjmtree`

- `bdttree`
- `hwtree`
- `bktree`
- `cirtree`

Data Types: `struct`

Output Arguments

`NumLevels` — Number of time levels of tree
numeric

Number of time levels of tree, returned as a numeric.

`NumPos` — Length of the state vectors in each level
vector

Length of the state vectors in each level, returned as a 1-by-`NUMLEVELS` vector.

`NumStates` — Number of state vectors in each level
vector

Number of state vectors in each level, returned as a 1-by-`NUMLEVELS` vector.

Version History

Introduced before R2006a

See Also

`mktrintree` | `trintreepath`

Topics
“Graphical Representation of Trees” on page 2-219
“Overview of Interest-Rate Tree Models” on page 2-44

agencyoas

Determine option-adjusted spread of callable bond using Agency OAS model

Syntax

```
OAS = agencyoas(ZeroData,Price,CouponRate,Settle,Maturity,Vol,CallDate)
OAS = agencyoas( ___ ,Name,Value)
```

Description

`OAS = agencyoas(ZeroData,Price,CouponRate,Settle,Maturity,Vol,CallDate)` computes OAS of a callable bond given price using the Agency OAS model.

`OAS = agencyoas( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Agency OAS Value

This example shows how to compute the agency `OAS` value.

```
Settle = '20-Jan-2010';
ZeroRates = [.07 .164 .253 1.002 1.732 2.226 2.605 3.316 ...
3.474 4.188 4.902]'/100;
ZeroDates = daysadd(Settle,360*[.25 .5 1 2 3 4 5 7 10 20 30],1);
ZeroData = [ZeroDates ZeroRates];

Maturity = datetime(2013,12,30);
CouponRate = .022;
Price = 99.155;
Vol = .5117;
CallDate = datetime(2010,12,30);
OAS = agencyoas(ZeroData, Price, CouponRate, Settle, Maturity, Vol, CallDate)
```

```
OAS = 8.5837
```

Input Arguments

`ZeroData` — Zero curve

matrix

Zero curve, specified as an `numRates`-by-2 matrix where the first column is zero dates and the second column is the accompanying zero rates.

Data Types: `double`

`Price` — Prices

vector

Prices specified as an `numBonds`-by-`1` vector.

Data Types: `double`

`CouponRate` — Coupon rates
vector in decimals

Coupon rates, specified as an `numBonds`-by-`1` vector in decimals.

Data Types: `double`

`Settle` — Settlement date
datetime scalar | scalar string | date character vector

Settlement date, specified as a scalar datetime, string, date character vector .

Note The `Settle` date must be an identical settlement date for all the bonds and the zero curve.

To support existing code, `agencyoas` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `numBonds`-by-`1` vector using datetime array, string array, or date character vectors.

To support existing code, `agencyoas` also accepts serial date numbers as inputs, but they are not recommended.

`Vol` — Volatilities
vector in decimals

Volatilities specified as a scalar or an `numBonds`-by-`1` vector in decimals. `Vol` is the volatility of interest rates corresponding to the time of the `CallDate`.

Data Types: `double`

`CallDate` — Call dates
datetime array | string array | date character vector

Call dates, specified as a scalar string or date character vector or an `numBonds`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `agencyoas` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `OAS = agencyoas(ZeroData,Price,CouponRate,Settle,Maturity,Vol,CallDate,'Basis',7,'Face',1000)`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `N`-by-`1` vector using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`CurveBasis` — Curve basis
`0` (actual/actual) (default) | integer from `0` to `13`

Curve basis, specified as the comma-separated pair consisting of `'CurveBasis'` and a `N`-by-`1` vector using the following values:

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`CurveCompounding` — Compounding frequency of the zero curve
2 (semiannual) (default) | possible values include: –1, 0, 1, 2, 3, 4, 6, 12.

Compounding frequency of the zero curve, specified as the comma-separated pair consisting of `'CurveCompounding'` and a `N`-by-`1` vector using the supported values: –1, 0, 1, 2, 3, 4, 6, and 12.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
1 (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [0, 1] using a `N`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`Face` — face value of bond
100 (default) | vector

Face value of bond, specified as the comma-separated pair consisting of `'Face'` and an `N`-by-`1` vector of numeric values.

Data Types: `double`

`FirstCouponDate` — Irregular first coupon date
if you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs (default) | datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar string or character vector or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `agencyoas` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`InterpMethod` — Interpolation method
`'linear'` (default) | `'cubic'`,`'pchip'`

Interpolation method, specified as the comma-separated pair consisting of `'InterpMethod'` and a `N`-by-`1` vector using a supported value. For more information on interpolation methods, see `interp1`.

Data Types: `char`

IssueDate — Bond issue date
if you do not specify an IssueDate, the cash flow payment dates are determined from other inputs (default) | datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of 'IssueDate' and scalar string or character vector or an N-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, agencyoas also accepts serial date numbers as inputs, but they are not recommended.

LastCouponDate — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a scalar string or character vector or a N-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, agencyoas also accepts serial date numbers as inputs, but they are not recommended.

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.

Period — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of 'Period' and an N-by-1 vector. Values for Period are 0, 1, 2, 3, 4, 6, and 12.

Data Types: double

StartDate — Forward starting date of payments
Settle date (default) | datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of 'StartDate' and a scalar string or character vector or an N-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, agencyoas also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify StartDate, the effective start date is the Settle date.

Output Arguments

OAS — Option-adjusted spreads
matrix

Option-adjusted spreads, returned as an numBonds-by-1 matrix.

More About

Agency OAS Model

The BMA European Callable Securities Formula provides a standard methodology for computing price and option-adjusted spread for European Callable Securities (ECS).

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `agencyoas` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] SIFMA, The BMA European Callable Securities Formula, `https://www.sifma.org`.

See Also

`agencyprice`

Topics

“Computing the Agency OAS for Bonds” on page 6-2
“Agency Option-Adjusted Spreads” on page 6-2
“Supported Interest-Rate Instrument Functions” on page 2-3

agencyprice

Price callable bond using Agency OAS model

Syntax

```
Price = agencyprice(ZeroData,OAS,CouponRate,Settle,Maturity,Vol,CallDate)
Price = agencyprice( ___ ,Name,Value)
```

Description

`Price = agencyprice(ZeroData,OAS,CouponRate,Settle,Maturity,Vol,CallDate)` computes the price for a callable bond, given OAS, using the Agency OAS model.

`Price = agencyprice( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Agency Price

This example shows how to compute the agency `Price`.

```
Settle = '20-Jan-2010';
ZeroRates = [.07 .164 .253 1.002 1.732 2.226 2.605 3.316 ...
3.474 4.188 4.902]'/100;
ZeroDates = daysadd(Settle,360*[.25 .5 1 2 3 4 5 7 10 20 30],1);
ZeroData = [ZeroDates ZeroRates];

Maturity = datetime(2013,12,30);
CouponRate = .022;
OAS = 6.53/10000;
Vol = .5117;
CallDate = datetime(2010,12,30);
Price = agencyprice(ZeroData, OAS, CouponRate, Settle, Maturity, Vol, CallDate)
```

```
Price = 99.4212
```

Input Arguments

ZeroData — Zero curve

matrix

Zero curve, specified as an `numRates`-by-2 matrix where the first column is zero dates and the second column is the accompanying zero rates.

Data Types: `double`

OAS — Option-adjusted spreads

vector in decimals

Option-adjusted spreads, specified as an `numBonds`-by-`1` vector expressed as a decimal (that is, 50 basis points is entered as `.005`).

Data Types: `double`

`CouponRate` — Coupon rates
vector in decimals

Coupon rates, specified as an `numBonds`-by-`1` vector in decimals.

Data Types: `double`

`Settle` — Settlement date
datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

Note The `Settle` date must be an identical settlement date for all the bonds and the zero curve.

To support existing code, `agencyprice` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | scalar array | date character vector

Maturity date, specified as a scalar string or character vector or `numBonds`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `agencyprice` also accepts serial date numbers as inputs, but they are not recommended.

`Vol` — Volatilities
vector in decimals

Volatilities specified as a scalar or an `numBonds`-by-`1` vector in decimals. `Vol` is the volatility of interest rates corresponding to the time of the `CallDate`.

Data Types: `double`

`CallDate` — Call dates
datetime array | string array | date character vector

Call dates, specified as a scalar string or character vector or an `numBonds`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `agencyprice` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Price = agencyprice(ZeroData,OAS,CouponRate,Settle,Maturity,Vol,CallDate,'Basis',7,'Face',1000)`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a `N`-by-`1` vector using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`CurveBasis` — Curve basis
`0` (actual/actual) (default) | integer from `0` to `13`

Curve basis, specified as the comma-separated pair consisting of `'CurveBasis'` and a `N`-by-`1` vector using the following values:

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`CurveCompounding` — Compounding frequency of the zero curve
2 (semiannual) (default) | possible values include: –1, 0, 1, 2, 3, 4, 6, 12.

Compounding frequency of the zero curve, specified as the comma-separated pair consisting of `'CurveCompounding'` and a N-by-1 vector using the supported values: –1, 0, 1, 2, 3, 4, 6, and 12.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
1 (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer [0, 1] using a N-by-1 vector with date character vectors or a datetime array.

- 0 = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- 1 = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`Face` — face value of bond
100 (default) | vector

Face value of bond, specified as the comma-separated pair consisting of `'Face'` and an N-by-1 vector of numeric values.

Data Types: `double`

`FirstCouponDate` — Irregular first coupon date
if you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs (default) | datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar string or character vector or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `agencyprice` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`InterpMethod` — Interpolation method
`'linear'` (default) | `'cubic'`,`'pchip'`

Interpolation method, specified as the comma-separated pair consisting of `'InterpMethod'` and a N-by-1 vector using a supported value. For more information on interpolation methods, see `interp1`.

Data Types: `char`

`IssueDate` — Bond issue date
if you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs (default) | datetime array | scalar array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar string or character vector or an `N`-by-`1` vector using a datetime array, string array, or date character vectors.

`LastCouponDate` — Irregular last coupon date
datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar string or character vector or an `N`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `agencyprice` also accepts serial date numbers as inputs, but they are not recommended.

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.

`Period` — Coupons per year
`2` per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and an `N`-by-`1` vector. Values for `Period` are `0`, `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`StartDate` — Forward starting date of payments
`Settle` date (default) | datetime array | string array | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a scalar string or character vector or an `N`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `agencyprice` also accepts serial date numbers as inputs, but they are not recommended.

If you do not specify `StartDate`, the effective start date is the `Settle` date.

Output Arguments

`Price` — Prices
matrix

Prices returned as an `numBonds`-by-`1` matrix.

More About

Agency OAS Model

The BMA European Callable Securities Formula provides a standard methodology for computing price and option-adjusted spread for European Callable Securities (ECS).

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `agencyprice` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] SIFMA, The BMA European Callable Securities Formula, `https://www.sifma.org`.

See Also

`agencyoas`

Topics

“Computing the Agency OAS for Bonds” on page 6-2
“Agency Option-Adjusted Spreads” on page 6-2
“Supported Interest-Rate Instrument Functions” on page 2-3

bkcall

Price European call option on bonds using Black model

Syntax

```
CallPrice = bkcall(Strike,ZeroData,Sigma,BondData,Settle,Expiry)
CallPrice = bkcall( ___ ,Period,Basis,EndMonthRule,InterpMethod,
StrikeConvention)
```

Description

`CallPrice = bkcall(Strike,ZeroData,Sigma,BondData,Settle,Expiry)` computes prices of European call options using Black's model.

`CallPrice = bkcall( ___ ,Period,Basis,EndMonthRule,InterpMethod, StrikeConvention)` adds optional input arguments for `Period`, `Basis`, `EndMonthRule`, `InterpMethod`, and `StrikeConvention`.

Examples

Price a European Call Option On Bonds Using the Black Model

This example shows how to price a European call option on bonds using the Black model. Consider a European call option on a bond maturing in 9.75 years. The underlying bond has a clean price of $935, a face value of $1000, and pays 10% semiannual coupons. Since the bond matures in 9.75 years, a $50 coupon will be paid in 3 months and again in 9 months. Also, assume that the annualized volatility of the forward bond price is 9%. Furthermore, suppose the option expires in 10 months and has a strike price of $1000, and that the annualized continuously compounded risk-free discount rates for maturities of 3, 9, and 10 months are 9%, 9.5%, and 10%, respectively.

```
% specify the option information
Settle       =  '15-Mar-2004';
Expiry       =  '15-Jan-2005'; % 10 months from settlement
Strike       =  1000;
Sigma        =  0.09;
Convention   =  [0 1]';

% specify the interest-rate environment
ZeroData     = [datenum('15-Jun-2004')  0.09   -1;  % 3 months
                datenum('15-Dec-2004')  0.095  -1;  % 9 months
                datenum(Expiry)         0.10   -1]; % 10 months

% specify the bond information
CleanPrice   =  935;
CouponRate   =  0.1;
Maturity     = '15-Dec-2013';  % 9.75 years from settlement
Face         =  1000;
BondData     = [CleanPrice CouponRate datenum(Maturity) Face];
Period       =  2;
Basis        =  1;
```

```
% call Black's model
CallPrices = bkcall(Strike, ZeroData, Sigma, BondData, Settle,...
Expiry, Period, Basis, [], [], Convention)

CallPrices = 2×1

    9.4873
    7.9686
```

When the strike price is the dirty price (`Convention = 0`), the call option value is $9.49. When the strike price is the clean price (`Convention = 1`), the call option value is $7.97.

Input Arguments

`Strike` — Strike price

numeric

Strike price, specified as a scalar numeric or an `NOPT`-by-`1` vector of strike prices.

Data Types: `double`

`ZeroData` — Zero rate information used to discount future cash flows

matrix

Zero rate information used to discount future cash flows, specified using a two-column (optionally three-column) matrix containing zero (spot) rate information used to discount future cash flows.

- Column 1 — Serial maturity date associated with the zero rate in the second column.
- Column 2 — Annualized zero rates, in decimal form, appropriate for discounting cash flows occurring on the date specified in the first column. All dates must occur after `Settle` (dates must correspond to future investment horizons) and must be in ascending order.
- Column 3 — (optional) Annual compounding frequency. Values are `1` (annual), `2` (semiannual, default), `3` (three times per year), `4` (quarterly), `6` (bimonthly), `12` (monthly), and `-1` (continuous).

If cash flows occur beyond the dates spanned by `ZeroData`, the input zero curve, the appropriate zero rate for discounting such cash flows is obtained by extrapolating the nearest rate on the curve (that is, if a cash flow occurs before the first or after the last date on the input zero curve, a flat curve is assumed).

In addition, you can use the method `getZeroRates` for an `IRDataCurve` object with a `Dates` property to create a vector of dates and data acceptable for `bkcall`. For more information, see “Converting an IRDataCurve or IRFunctionCurve Object” on page 9-30.

Data Types: `double`

`Sigma` — Annualized price volatilities required by Black model

numeric

Annualized price volatilities required by the Black model, specified as a scalar or an `NOPT`-by-`1` vector.

Data Types: `struct`

BondData — Characteristics of underlying bonds
vector

Characteristics of underlying bonds, specified as a row vector with three (optionally four) columns or NOPT-by-3 (optionally NOPT-by-4) matrix specifying characteristics of underlying bonds in the form:

```
[CleanPrice CouponRate Maturity Face]
```

- `CleanPrice` is the price excluding accrued interest.
- `CouponRate` is the decimal coupon rate.
- `Maturity` is the bond maturity date using a datetime format.
- `Face` is the face value of the bond. If unspecified, the face value is assumed to be 100.

Data Types: `double` | `datetime`

Settle — Settlement date
string scalar | date character vector | serial date number

Settlement date, specified as a scaler datetime, string, date character vector, or serial date number. `Settle` also represents the starting reference date for the input zero curve.

Data Types: `char` | `double` | `string`

Expiry — Option maturity date
string array | date character vector | serial date number

Option maturity date, specified as an NOPT-by-1 vector using a datetime array, string array, date character vectors, or serial date numbers.

Data Types: `char` | `string` | `double`

Period — Number of coupons per year for underlying bond
2 (semiannual) (default) | integer with value 0, 1, 2, 3, 4, 6, or 12

(Optional) Number of coupons per year for the underlying bond, specified as an integer with supported values of 0, 1, 2, 3, 4, 6, and 12.

Data Types: `double`

Basis — Day-count basis of underlying bonds
0 (actual/actual) (default) | integer from 0 to 13

(Optional) Day-count basis of underlying bonds, specified as a scalar or an NOPT-by-1 vector using the following values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `[0,1]`

(Optional) End-of-month rule flag, specified as a scalar or an `NOPT`-by-`1` vector of end-of-month rules.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`InterpMethod` — Zero curve interpolation method
`1` (linear interpolation) (default) | integer with value `0`, `1`, or `2`

(Optional) Zero curve interpolation method for cash flows that do not fall on a date found in the `ZeroData` spot curve, specified as a scalar integer. `InterpMethod` is used to interpolate the appropriate zero discount rate. Available interpolation methods are (`0`) nearest, (`1`) linear, and (`2`) cubic. For more information on interpolation methods, see `interp1`.

Data Types: `double`

`StrikeConvention` — Option contract strike price convention
`0` (default) | integer with value `0` or `1`

(Optional) Option contract strike price convention, specified as a scalar or an `NOPT`-by-`1` vector.

`StrikeConvention = 0` (default) defines the strike price as the cash (dirty) price paid for the underlying bond.

`StrikeConvention = 1` defines the strike price as the quoted (clean) price paid for the underlying bond. When evaluating Black's model, the accrued interest of the bond at option expiration is added to the input strike price.

Data Types: `double`

Output Arguments

`CallPrice` — Price for European call option on bonds derived from Black model
vector

Price for European call option on bonds derived from the Black model, returned as a `NOPT`-by-`1` vector.

Version History

Introduced before R2006a

References

[1] Hull, John C. *Options, Futures, and Other Derivatives*. 5th Edition, Prentice Hall, 2003, pp. 287–288, 508–515.

See Also

`bkput`

Topics

“Analysis of Bond Futures” on page 7-12
“Fitting the Diebold Li Model” on page 2-150
“Supported Interest-Rate Instrument Functions” on page 2-3

bkcaplet

Price interest-rate caplet using Black model

Note `bkcaplet` has been removed. Use `capbyblk` instead.

Syntax

```
CapPrices = bkcaplet(CapData,FwdRates,ZeroPrice,Settle,StartDate,EndDate,Sigma)
```

Arguments

`CapData`	Number of caps (`NCAP`)-by-2 matrix containing cap rates and bases: `[CapRates Basis]`. Values for bases are: • 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/365 (ISDA) • 13 = BUS/252 For more information, see “Basis” on page 2-228.
`FwdRates`	Scalar or `NCAP`-by-1 vector containing forward rates in decimal. `FwdRates` accrue on the same basis as `CapRates`.
`ZeroPrice`	Scalar or `NCAP`-by-1 vector containing zero coupon prices with maturities corresponding to those of each cap in `CapData`, per $100 nominal value.
`Settle`	Scalar or `NCAP`-by-1 vector of identical elements containing settlement date of caplets.
`StartDate`	Scalar or `NCAP`-by-1 vector containing start dates of the caplets.
`EndDate`	Scalar or `NCAP`-by-1 vector containing maturity dates of caplets.

`Sigma`	Scalar or `NCAP`-by-`1` vector containing volatility of forward rates in decimal, corresponding to each caplet.

Description

`CapPrices = bkcaplet(CapData,FwdRates,ZeroPrice,Settle,StartDate,EndDate,Sigma)` computes the prices of interest-rate caplets for every $100 face value of principal.

Examples

Compute the Price of Interest-Rate Caplets for Every $100 Face Value of Principal

This example shows how to compute the price of interest-rate caplets for every $100 face value of principal. Given a notional amount of $1,000,000, compute the value of a caplet on October 15, 2002 that starts on October 15, 2003 and ends on January 15, 2004.

```
CapData = [0.08, 1];
FwdRates = 0.07;
ZeroPrice = 100*exp(-0.065*1.25);
Settle = datenum('15-Oct-2002');
BeginDates = datenum('15-Oct-2003');
EndDates = datenum('15-Jan-2004');
Sigma = 0.20;

% because the caplet is $100 notional, divide $1,000,000 by $100
Notional   = 1000000/100;

CapPrice = Notional*bkcaplet(CapData, FwdRates, ZeroPrice, ...
Settle, BeginDates, EndDates, Sigma)

Error using bkcaplet (line 117)
BKCAPLET has been removed. Use CAPBYBLK instead.
```

Version History

Introduced before R2006a

See Also

Topics

“Analysis of Bond Futures” on page 7-12
“Fitting the Diebold Li Model” on page 2-150
“Supported Interest-Rate Instrument Functions” on page 2-3

bkfloorlet

Price interest-rate floorlet using Black model

Note `bkfloorlet` has been removed. Use `floorbyblk` instead.

Syntax

```
FloorPrices = bkfloorlet(FloorData,FwdRates,ZeroPrice,Settle,StartDate,EndDate,Sigma)
```

Arguments

`FloorData`	Number of floors (`NFLR`)-by-2 matrix containing floor rates and bases: `[FloorRate Basis]`. Values for bases are: • 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/365 (ISDA) • 13 = BUS/252 For more information, see “Basis” on page 2-228.
`FwdRates`	Scalar or `NFLR`-by-`1` vector containing forward rates in decimal. `FwdRates` accrue on the same basis as `FloorRates`.
`ZeroPrice`	Scalar or `NFLR`-by-`1` vector containing zero coupon prices with maturities corresponding to those of each floor in `FloorData`, per $100 nominal value.
`Settle`	Scalar or `NFLR`-by-`1` vector of identical elements containing settlement date of floorlets.
`StartDate`	Scalar or `NFLR`-by-`1` vector containing start dates of the floorlets.
`EndDate`	Scalar or `NFLR`-by-`1` vector containing maturity dates of floorlets.

`Sigma`	Scalar or `NFLR`-by-`1` vector containing volatility of forward rates in decimal, corresponding to each floorlet.

Description

`FloorPrices = bkfloorlet(FloorData,FwdRates,ZeroPrice,Settle,StartDate,EndDate,Sigma)` computes the prices of interest-rate floorlets for every $100 of notional value.

Examples

Price an Interest-Rate Floorlet For Every $100 of Notional Value Using the Black Model

This example shows how to price an interest-rate floorlet for every $100 of notional value using the Black model. Given a notional amount of $1,000,000, compute the value of a floorlet on October 15, 2002 that starts on October 15, 2003 and ends on January 15, 2004.

```
FloorData = [0.08, 1];
FwdRates = 0.07;
ZeroPrice = 100*exp(-0.065*1.25);
Settle = datenum('15-Oct-2002');
BeginDates = datenum('15-Oct-2003');
EndDates = datenum('15-Jan-2004');
Sigma = 0.20;

% because floorlet is $100 notional, divide $1,000,000 by $100
Notional = 1000000/100;

FloorPrice = Notional*bkfloorlet(FloorData, FwdRates, ...
ZeroPrice, Settle, BeginDates, EndDates, Sigma)

Error using bkfloorlet (line 115)
BKFLOORLET has been removed. Use FLOORBYBLK instead.
```

Version History

Introduced before R2006a

See Also

Topics

“Analysis of Bond Futures” on page 7-12
“Fitting the Diebold Li Model” on page 2-150
“Supported Interest-Rate Instrument Functions” on page 2-3

bkput

Price European put option on bonds using Black model

Syntax

```
PutPrice = bkput(Strike,ZeroData,Sigma,BondData,Settle,Expiry)
PutPrice = bkput( ___ ,Period,Basis,EndMonthRule,InterpMethod,StrikeConvention)
```

Description

`PutPrice = bkput(Strike,ZeroData,Sigma,BondData,Settle,Expiry)` computes prices of European put options using a Black model.

`PutPrice = bkput( ___ ,Period,Basis,EndMonthRule,InterpMethod,StrikeConvention)` adds optional input arguments for `Period`, `Basis`, `EndMonthRule`, `InterpMethod`, and `StrikeConvention`.

Examples

Price European Put Options On Bonds Using the Black Model

This example shows how to price European put options on bonds using the Black model. Consider a European put option on a bond maturing in 10 years. The underlying bond has a clean price of $122.82, a face value of $100, and pays 8% semiannual coupons. Also, assume that the annualized volatility of the forward bond yield is 20%. Furthermore, suppose the option expires in 2.25 years and has a strike price of $115, and that the annualized continuously compounded risk free zero (spot) curve is flat at 5%. For a hypothetical settlement date of March 15, 2004, the following code illustrates the use of Black's model to duplicate the put prices in Example 22.2 of the Hull reference. In particular, it illustrates how to convert a broker's yield volatility to a price volatility suitable for Black's model.

```
% Specify the option information.
Settle       =  '15-Mar-2004';
Expiry       =  '15-Jun-2006';  % 2.25 years from settlement
Strike       =  115;
YieldSigma   =  0.2;
Convention   =  [0; 1];

% Specify the interest-rate environment. Since the
% zero curve is flat, interpolation into the curve always returns
% 0.05. Thus, the following curve is not unique to the solution.
ZeroData     = [datenum('15-Jun-2004') 0.05   -1;
                datenum('15-Dec-2004') 0.05   -1;
                datenum(Expiry)        0.05   -1];

% Specify the bond information.
CleanPrice   =  122.82;
CouponRate   =  0.08;
Maturity     = '15-Mar-2014';  % 10 years from settlement
Face         =  100;
```

```
BondData     = [CleanPrice CouponRate datenum(Maturity) Face];
Period       =  2;  % semiannual coupons
Basis        =  1;  % 30/360 day-count basis

% Convert a broker's yield volatility quote to a price volatility
% required by Black's model. To duplicate Example 22.2 in Hull,
% first compute the periodic (semiannual) yield to maturity from
% the clean bond price.
Yield  = bndyield(CleanPrice, CouponRate, Settle, Maturity,...
Period, Basis);

% Compute the duration of the bond at option expiration. Most
% fixed-income sensitivity analyses use the modified duration
% statistic to examine the impact of small changes in periodic
% yields on bond prices. However, Hull's example operates in
% continuous time (annualized instantaneous volatilities and
% continuously compounded zero yields for discounting coupons).
% To duplicate Hull's results, use the second output of BNDDURY,
% the Macaulay duration.
[Modified, Macaulay] = bnddury(Yield, CouponRate, Expiry,...
Maturity, Period, Basis);

% Convert the yield-to-maturity from a periodic to a
% continuous yield.
Yield  = Period .* log(1 + Yield./Period);

% Convert the yield volatility to a price volatility via
% Hull's Equation 22.6 (page 514).
PriceSigma = Macaulay .* Yield .* YieldSigma;

% Finally, call Black's model.
PutPrices  = bkput(Strike, ZeroData, PriceSigma, BondData,...
Settle, Expiry, Period, Basis, [], [], Convention)
```

```
PutPrices = 2×1

    1.7838
    2.4071
```

When the strike price is the dirty price (`Convention = 0`), the call option value is $1.78. When the strike price is the clean price (`Convention = 1`), the call option value is $2.41.

Input Arguments

`Strike` — Strike price

numeric

Strike price, specified as a scalar numeric or an `NOPT`-by-`1` vector of strike prices.

Data Types: `double`

`ZeroData` — Zero rate information used to discount future cash flows

matrix

Zero rate information used to discount future cash flows, specified using a two-column (optionally three-column) matrix containing zero (spot) rate information used to discount future cash flows.

- Column 1 — Serial maturity date associated with the zero rate in the second column.
- Column 2 — Annualized zero rates, in decimal form, appropriate for discounting cash flows occurring on the date specified in the first column. All dates must occur after `Settle` (dates must correspond to future investment horizons) and must be in ascending order.
- Column 3 — (optional) Annual compounding frequency. Values are `1` (annual), `2` (semiannual, default), `3` (three times per year), `4` (quarterly), `6` (bimonthly), `12` (monthly), and `-1` (continuous).

If cash flows occur beyond the dates spanned by `ZeroData`, the input zero curve, the appropriate zero rate for discounting such cash flows is obtained by extrapolating the nearest rate on the curve (that is, if a cash flow occurs before the first or after the last date on the input zero curve, a flat curve is assumed).

In addition, you can use the method `getZeroRates` for an `IRDataCurve` object with a `Dates` property to create a vector of dates and data acceptable for `bkput`. For more information, see “Converting an IRDataCurve or IRFunctionCurve Object” on page 9-30.

Data Types: `double`

`Sigma` — Annualized price volatilities required by Black model
numeric

Annualized price volatilities required by the Black model, specified as a scalar or an `NOPT`-by-`1` vector.

Data Types: `struct`

`BondData` — Characteristics of underlying bonds
vector

Characteristics of underlying bonds, specified as a row vector with three (optionally four) columns or `NOPT`-by-`3` (optionally `NOPT`-by-`4`) matrix specifying characteristics of underlying bonds in the form:

```
[CleanPrice CouponRate Maturity Face]
```

- `CleanPrice` is the price excluding accrued interest.
- `CouponRate` is the decimal coupon rate.
- `Maturity` is the bond maturity date using a serial date number, date character vector, or string.
- `Face` is the face value of the bond. If unspecified, the face value is assumed to be 100.

Data Types: `double` | `char` | `string`

`Settle` — Settlement date
string scalar | date character vector | serial date number

Settlement date, specified as a scalar string, date character vector, or serial date number. `Settle` also represents the starting reference date for the input zero curve.

Data Types: `char` | `double` | `string`

`Expiry` — Option maturity date
string array | date character vector | serial date number

Option maturity date, specified as an `NOPT`-by-`1` vector using a string array, date character vectors, or serial date numbers.

Data Types: `char` | `string` | `double`

`Period` — Number of coupons per year for underlying bond
2 (semiannual) (default) | integer with value `0`, `1`, `2`, `3`, `4`, `6`, or `12`

(Optional) Number of coupons per year for the underlying bond, specified as an integer with supported values of `0`, `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`Basis` — Day-count basis of underlying bonds
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis of underlying bonds, specified as a scalar or an `NOPT`-by-`1` vector using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `[0,1]`

(Optional) End-of-month rule flag, specified as a scalar or an `NOPT`-by-`1` vector of end-of-month rules.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`InterpMethod` — Zero curve interpolation method
`1` (linear interpolation) (default) | integer with value `0`, `1`, or `2`

(Optional) Zero curve interpolation method for cash flows that do not fall on a date found in the `ZeroData` spot curve, specified as a scalar integer. `InterpMethod` is used to interpolate the

appropriate zero discount rate. Available interpolation methods are (`0`) nearest, (`1`) linear, and (`2`) cubic. For more information on interpolation methods, see `interp1`.

Data Types: `double`

`StrikeConvention` — Option contract strike price convention
`0` (default) | integer with value `0` or `1`

(Optional) Option contract strike price convention, specified as a scalar or an `NOPT`-by-`1` vector.

`StrikeConvention = 0` (default) defines the strike price as the cash (dirty) price paid for the underlying bond.

`StrikeConvention = 1` defines the strike price as the quoted (clean) price paid for the underlying bond. When evaluating Black's model, the accrued interest of the bond at option expiration is added to the input strike price.

Data Types: `double`

Output Arguments

`PutPrice` — Price for European put option on bonds derived from Black model
vector

Price for European put option on bonds derived from the Black model, returned as a `NOPT`-by-`1` vector.

Version History

Introduced before R2006a

References

[1] Hull, John C. *Options, Futures, and Other Derivatives*. 5th Edition, Prentice Hall, 2003, pp. 287–288, 508–515.

See Also

`bkcall`

Topics

“Analysis of Bond Futures” on page 7-12
“Fitting the Diebold Li Model” on page 2-150
“Supported Interest-Rate Instrument Functions” on page 2-3

bndfutimprepo

Implied repo rates for bond future given price

Syntax

```
ImpRepo = bndfutimprep(Price,FutPrice,FutSettle,Delivery,ConvFactor,
CouponRate,Maturity)
ImpRepo = bndfutimprep( ___ ,Name,Value)
```

Description

`ImpRepo = bndfutimprep(Price,FutPrice,FutSettle,Delivery,ConvFactor, CouponRate,Maturity)` computes the implied repo rate for a bond future given the price of a bond, the bond properties, the price of the bond future, and the bond conversion factor. The default behavior is that the coupon reinvestment rate matches the repo rate. However, you can specify a separate reinvestment rate using optional inputs.

`ImpRepo = bndfutimprep( ___ ,Name,Value)` specifies options using one or more optional name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Repo Rate For a Bond Future

This example shows how to compute the repo rate for a bond future using the following data.

```
bndfutimprepo(129,98,datetime(2000,9,21),datetime(2000,12,29),1.3136,.0875,datetime(2020,8,15))
```

```
ans = 0.0584
```

Input Arguments

`Price` — Bond prices

vector

Bond prices, specified as an `numBonds`-by-`1` vector in decimals.

Data Types: `double`

`FutPrice` — Future prices

vector

Future prices, specified as an `numBonds`-by-`1` vector.

Data Types: `double` | `cell`

`FutSettle` — Future settlement dates

datetime array | string array | date character vector

Future settlement dates, specified as an `numBonds`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndfutimprepo` also accepts serial date numbers as inputs, but they are not recommended.

`Delivery` — Future delivery dates
datetime array | string array | date character vector

Future delivery dates, specified as an `numBonds`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndfutimprepo` also accepts serial date numbers as inputs, but they are not recommended.

`ConvFactor` — Bond conversion factors
vector

Bond conversion factors, specified as an `numBonds`-by-`1` vector. For more information, see `convfactor`.

Data Types: `double`

`CouponRate` — Coupon rates
vector

Coupon rates, specified as an `numBonds`-by-`1` vector of numeric decimals.

Data Types: `double`

`Maturity` — Maturity dates
datetime array | string array | date character vector

Maturity dates, specified as an `numBonds`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndfutimprepo` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ImpRepo = bndfutimprepo(Price,FutPrice,FutSettle,Delivery,ConvFactor,CouponRate,Maturity,'Basis',5,'Face',1000,'Period',4)`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer from `0` to `13`.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating floorlet dates
`1` (in effect) (default) | scalar of nonnegative integer `[0,1]`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar with a nonnegative integer `[0, 1]`.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`Face` — Face value of the bond
`100` (default) | scalar numeric

Face value of the bond, specified as the comma-separated pair consisting of `'Face'` and a scalar numeric. `Face` has no impact on key rate duration.

Data Types: `double`

`FirstCouponDate` — Irregular first coupon date
if you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs (default) | datetime scalar | string scalar | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar datetime, string, or date character vector.

To support existing code, `bndfutimprepo` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`IssueDate` — Bond issue date
if you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs (default) | datetime scalar | string scalar | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar datetime, string, or date character vector.

To support existing code, `bndfutimprepo` also accepts serial date numbers as inputs, but they are not recommended.

`LastCouponDate` — Irregular last coupon date
datetime scalar | string scalar | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar datetime, string, or date character vector.

To support existing code, `bndfutimprepo` also accepts serial date numbers as inputs, but they are not recommended.

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.

`Period` — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a scalar integer. Values for `Period` are `0`, `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`ReinvestBasis` — Day count basis for reinvestment rate
identical to `RepoBasis` (default) | integer from `0` to `13`

Day count basis for the reinvestment rate, specified as the comma-separated pair consisting of `'ReinvestBasis'` and a scalar integer from `0` to `13`.

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

- 13 = BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`ReinvestRate` — Underlying bond annual coupon
scalar decimal numeric

Underlying bond annual coupon, specified as the comma-separated pair consisting of `'ReinvestRate'` and a scalar decimal numeric.

Data Types: `double`

`RepoBasis` — Day count basis for repo rate
2 (actual/360) (default) | integer from `0` to `13`

Day count basis for repo rate, specified as the comma-separated pair consisting of `'RepoBasis'` and a scalar integer from `0` to `13`.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`StartDate` — Forward starting date of payments
`Settle` date (default) | datetime scalar | string scalar | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a scalar datetime, string, or date character vector.

To support existing code, `bndfutimprepo` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`ImpRepo` — Implied repo rate
vector

Implied repo rate, or the repo rate that would produce the price input, returned as `numBonds`-by-1 vector.

Version History

Introduced in R2009b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `bndfutimprepo` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Burghardt, G., T. Belton, M. Lane, and J. Papa. *The Treasury Bond Basis.* McGraw-Hill, 2005.

[2] Krgin, Dragomir. *Handbook of Global Fixed Income Calculations.* John Wiley & Sons, 2002.

See Also

`bndfutprice` | `convfactor`

Topics
“Analysis of Bond Futures” on page 7-12
“Fitting the Diebold Li Model” on page 2-150
“Supported Interest-Rate Instrument Functions” on page 2-3

bndfutprice

Price bond future given repo rates

Syntax

```
[FutPrice,AccrInt] = bndfutprice(RepoRatePrice,FutSettle,Delivery,ConvFactor,
CouponRate,Maturity)
[FutPrice,AccrInt] = bndfutprice( ___ ,Name,Value)
```

Description

`[FutPrice,AccrInt] = bndfutprice(RepoRatePrice,FutSettle,Delivery,ConvFactor, CouponRate,Maturity)` computes the price of a bond futures contract for one or more bonds given a repo rate, and bond properties, including the bond conversion factor. The default behavior is that the coupon reinvestment rate matches the repo rate. However, you can specify a separate reinvestment rate using optional arguments.

`[FutPrice,AccrInt] = bndfutprice( ___ ,Name,Value)` specifies options using one or more optional name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Price For a Bond Future

This example shows how to compute the price for a bond future using the following data.

```
bndfutprice(.064, 129, datetime(2000,9,21),datetime(2000,12,29), 1.3136, .0875, datetime(2020,8,1
```

```
ans = 98.1516
```

Input Arguments

`RepoRate` — Repo rates

vector

Repo rates, specified as an `numBonds`-by-1 vector in decimals.

Data Types: `double`

`Price` — Bond prices

vector

Bond prices, specified as an `numBonds`-by-1 vector in decimals.

Data Types: `double`

`FutSettle` — Future settlement date

datetime array | string array | date character vector

Future settlement date, specified as an `numBonds`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndfutprice` also accepts serial date numbers as inputs, but they are not recommended.

`Delivery` — Future delivery dates
datetime array | string array | date character vector

Future delivery dates, specified as an `numBonds`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndfutprice` also accepts serial date numbers as inputs, but they are not recommended.

`ConvFactor` — Bond conversion factors
vector

Bond conversion factors, specified as an `numBonds`-by-`1` vector. For more information, see `convfactor`.

Data Types: `double`

`CouponRate` — Coupon rates
vector

Coupon rates, specified as an `numBonds`-by-`1` vector of numeric decimals.

Data Types: `double`

`Maturity` — Maturity dates
datetime array | string array | date character vector

Maturity dates, specified as an `numBonds`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndfutprice` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[FutPrice,AccrInt] = bndfutprice(RepoRate,Price,FutSettle,Delivery,ConvFactor,CouponRate,Maturity, 'Basis',5,'Face',1000,'Period',4)`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer from `0` to `13`.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating floorlet dates
`1` (in effect) (default) | scalar of nonnegative integer `[0,1]`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar with a nonnegative integer [`0`, `1`].

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
if you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs (default) | datetime scalar | string scalar | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar datetime, string, or date character vector.

To support existing code, `bndfutprice` also accepts serial date numbers as inputs, but they are not recommended.

`Face` — Face value of the bond
`100` (default) | scalar numeric

Face value of the bond, specified as the comma-separated pair consisting of `'Face'` and a scalar numeric. `Face` has no impact on key rate duration.

Data Types: `double`

`FirstCouponDate` — Irregular first coupon date
if you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs (default) | datetime scalar | string scalar | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar datetime, string, or date character vector.

To support existing code, `bndfutprice` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`LastCouponDate` — Irregular last coupon date
datetime scalar | string scalar | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar datetime, string, or date character vector.

To support existing code, `bndfutprice` also accepts serial date numbers as inputs, but they are not recommended.

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.

`Period` — Coupons per year
2 per year (default) | vector

Coupons per year, specified as the comma-separated pair consisting of `'Period'` and a scalar integer. Values for `Period` are `0`, `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`ReinvestBasis` — Day count basis for reinvestment rate
identical to `RepoBasis` (default) | integer from `0` to `13`

Day count basis for the reinvestment rate, specified as the comma-separated pair consisting of `'ReinvestBasis'` and a scalar integer from `0` to `13`.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`ReinvestRate` — Underlying bond annual coupon
scalar decimal numeric

Underlying bond annual coupon, specified as the comma-separated pair consisting of `'ReinvestRate'` and a scalar decimal numeric.

Data Types: `double`

`RepoBasis` — Day count basis for repo rate
2 (actual/360) (default) | integer from `0` to `13`

Day count basis for repo rate, specified as the comma-separated pair consisting of `'RepoBasis'` and a scalar integer from `0` to `13`.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`StartDate` — Forward starting date of payments
`Settle` date (default) | datetime scalar | string scalar | date character vector

Forward starting date of payments (the date from which a bond cash flow is considered), specified as the comma-separated pair consisting of `'StartDate'` and a scalar datetime, string, or date character vector.

To support existing code, `bndfutprice` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`FutPrice` — Quoted futures price, per $100 notional
vector

Quoted futures price, per $100 notional, returned as `numBonds`-by-`1` vector.

`AccrInt` — Accrued interest due at delivery date, per $100 notional
vector

Accrued interest due at delivery date, per $100 notional, returned as `numBonds`-by-`1` vector.

Version History

Introduced in R2009b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `bndfutprice` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Burghardt, G., T. Belton, M. Lane, and J. Papa. *The Treasury Bond Basis.* McGraw-Hill, 2005.

[2] Krgin, Dragomir. *Handbook of Global Fixed Income Calculations.* John Wiley & Sons, 2002.

See Also

`bndfutimprepo` | `convfactor`

Topics

“Analysis of Bond Futures” on page 7-12
“Fitting the Diebold Li Model” on page 2-150
“Supported Interest-Rate Instrument Functions” on page 2-3

bootstrap

Bootstrap interest-rate curve from market data

Syntax

```
DCurve = IRDataCurve.bootstrap(Type,Settle,InstrumentTypes,Instruments)
DCurve = IRDataCurve.bootstrap( ___ ,Name,Value)
```

Description

`DCurve = IRDataCurve.bootstrap(Type,Settle,InstrumentTypes,Instruments)` bootstraps an interest-rate curve from market data. The dates of the bootstrapped curve correspond to the maturity dates of the input instruments.

Note The `ratecurve` object and associated methods were introduced in R2020a as part of a new object-based framework in the Financial Instruments Toolbox which supports end-to-end workflows in instrument modeling and analysis. For more information, see `irbootstrap` and "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

`DCurve = IRDataCurve.bootstrap( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Use the bootstrap Method to Create an IRDataCurve Object

In this bootstrapping example, `InstrumentTypes`, `Instruments`, and a `Settle` date are defined:

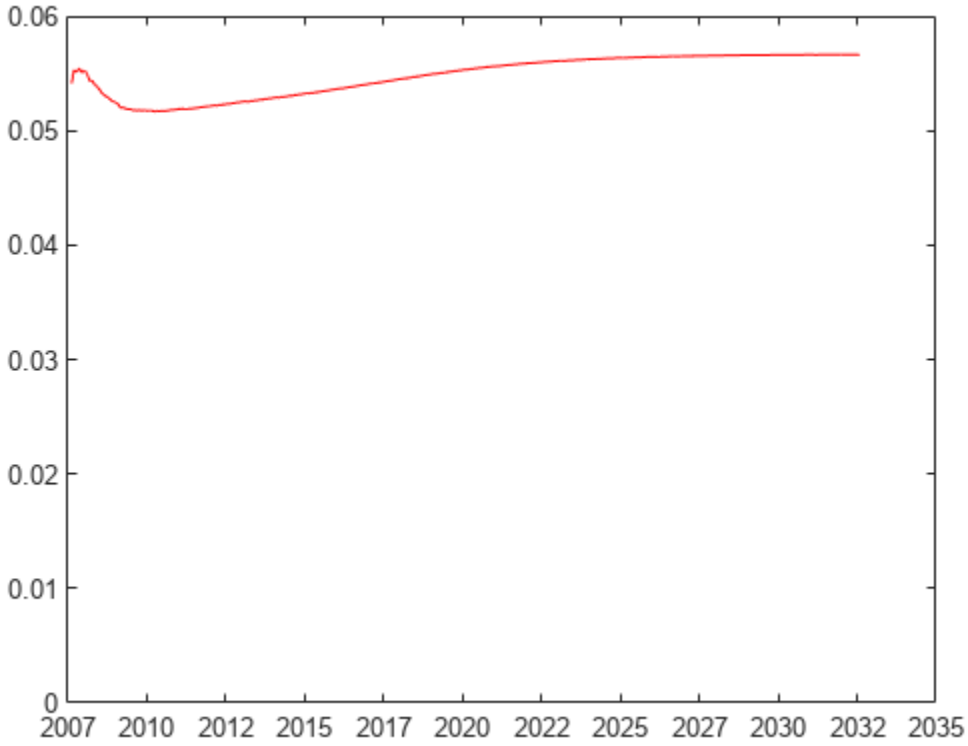
```
InstrumentTypes = {'Deposit';'Deposit';...
'Futures';'Futures';'Futures';'Futures';'Futures';'Futures';...
'Swap';'Swap';'Swap';'Swap';};

Instruments = [datenum('08/10/2007'),datenum('09/17/2007'),.0532000; ...
datenum('08/10/2007'),datenum('11/17/2007'),.0535866; ...
datenum('08/08/2007'),datenum('19-Dec-2007'),9485; ...
datenum('08/08/2007'),datenum('19-Mar-2008'),9502; ...
datenum('08/08/2007'),datenum('18-Jun-2008'),9509.5; ...
datenum('08/08/2007'),datenum('17-Sep-2008'),9509; ...
datenum('08/08/2007'),datenum('17-Dec-2008'),9505.5; ...
datenum('08/08/2007'),datenum('18-Mar-2009'),9501; ...
datenum('08/08/2007'),datenum('08/08/2014'),.0530; ...
datenum('08/08/2007'),datenum('08/08/2019'),.0551; ...
datenum('08/08/2007'),datenum('08/08/2027'),.0565; ...
datenum('08/08/2007'),datenum('08/08/2037'),.0566];

CurveSettle = datenum('08/10/2007');
```

Use the `bootstrap` method to create an `IRDataCurve` object.

```
bootModel = IRDataCurve.bootstrap('Forward', CurveSettle, ...
InstrumentTypes, Instruments,'InterpMethod','pchip');
```

To create the plot for the bootstrapped market data:

```
PlottingDates = (datenum('08/11/2007'):30:CurveSettle+365*25)';
plot(PlottingDates, getParYields(bootModel, PlottingDates),'r')
ylim([0 .06])
datetick
```



Use the `bootstrap` Method to Create an `IRDataCurve` Object That Includes Bonds

In this bootstrapping example, `InstrumentTypes`, `Instruments`, and a `Settle` date are defined:

```
CurveSettle = datenum('8-Mar-2010');

InstrumentTypes = {'Deposit';'Deposit';'Deposit';'Deposit';...
    'Futures';'Futures';'Futures';'Futures';'Swap';'Swap';'Bond';'Bond'};

Instruments = [datenum('8-Mar-2010'),datenum('8-Apr-2010'),.003; ...
    datenum('8-Mar-2010'),datenum('8-Jun-2010'),.005; ...
    datenum('8-Mar-2010'),datenum('8-Sep-2010'),.007; ...
    datenum('8-Mar-2010'),datenum('8-Mar-2011'),.009; ...
    datenum('8-Mar-2010'),datenum('18-Jun-2011'),9840; ...
    datenum('8-Mar-2010'),datenum('17-Sep-2011'),9820; ...
```

```
    datenum('8-Mar-2010'),datenum('17-Dec-2011'),9810; ...
    datenum('8-Mar-2010'),datenum('18-Mar-2012'),9800; ...
    datenum('8-Mar-2010'),datenum('8-Mar-2015'),.025; ...
    datenum('8-Mar-2010'),datenum('8-Mar-2020'),.035; ...
    datenum('8-Mar-2010'),datenum('8-Mar-2030'),99; ...
    datenum('8-Mar-2010'),datenum('8-Mar-2040'),101];
```

When bonds are used, `InstrumentCouponRate` must be specified:

```
InstrumentCouponRate = [zeros(10,1);.045;.05];
```

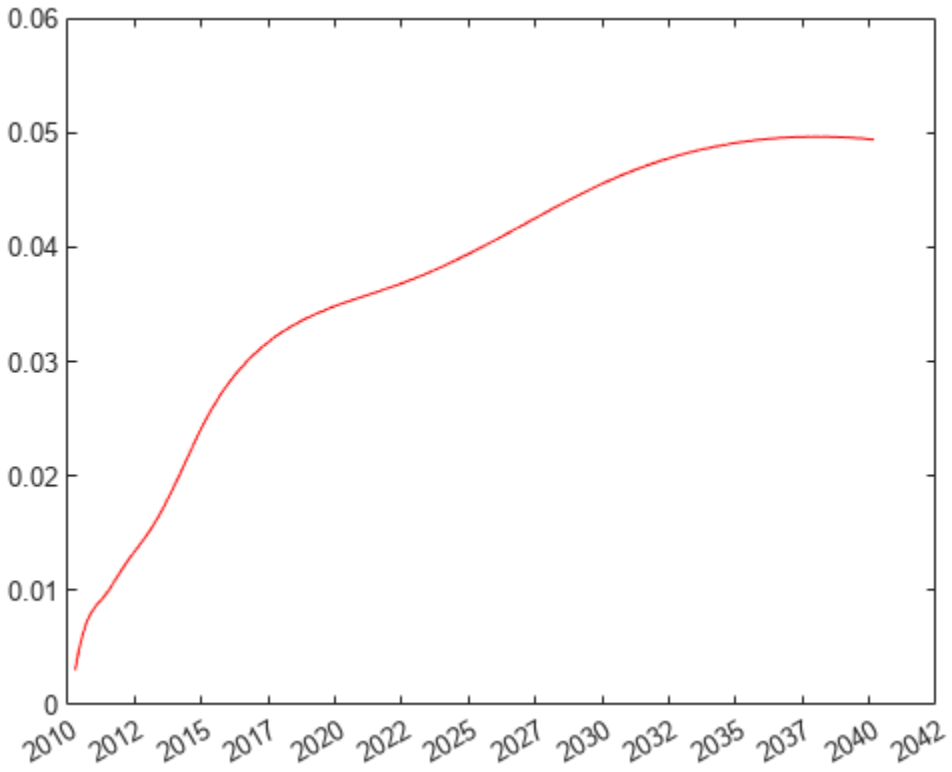
Note, for parameters that are only applicable to bonds (`InstrumentFirstCouponDate`, `InstrumentLastCouponDate`, `InstrumentIssueDate`, `InstrumentFace`) the entries for non-bond instruments (deposits and futures) are ignored.

Use the `bootstrap` method to create an `IRDataCurve` object.

```
bootModel = IRDataCurve.bootstrap('Forward', CurveSettle, ...
InstrumentTypes, Instruments,'InterpMethod','pchip',...
'InstrumentCouponRate',InstrumentCouponRate);
```

Create the plot for the bootstrapped market data.

```
PlottingDates = datemnth(CurveSettle,1:30*12);
plot(PlottingDates, getParYields(bootModel, PlottingDates),'r')
ylim([0 .06])
datetick
```



Use IRBootstrapOptionsObj with bootstrap for Negative Zero Interest-Rates

Use the `IRBootstrapOptionsObj` optional argument with the `bootstrap` method to allow for negative zero rates when solving for the swap zero points.

```
Settle = datenum('15-Mar-2015');
InstrumentTypes = {'Deposit';'Deposit';'Swap';'Swap';'Swap';'Swap';};

Instruments = [Settle,datenum('15-Jun-2015'),.001; ...
Settle,datenum('15-Dec-2015'),.0005; ...
Settle,datenum('15-Mar-2016'),-.001; ...
Settle,datenum('15-Mar-2017'),-0.0005; ...
Settle,datenum('15-Mar-2018'),.0017; ...
Settle,datenum('15-Mar-2020'),.0019];

irbo = IRBootstrapOptions('LowerBound',-1);

bootModel = IRDataCurve.bootstrap('zero', Settle, InstrumentTypes,...
    Instruments,'IRBootstrapOptions',irbo);

bootModel.getZeroRates(datemnth(Settle,1:60))
```

```
ans = 60×1

    0.0012
    0.0011
    0.0010
    0.0009
    0.0008
    0.0008
    0.0007
    0.0006
    0.0005
   -0.0000
      ⋮
```

Note that optional argument for `LowerBound` is set to `-1` for negative zero rates when solving the swap zero points.

Input Arguments

Type — Type of interest-rate curve bootstrapped from market instruments

character vector with value of `'zero'`, `'discount'`, or `'forward'`

Type of interest-rate curve bootstrapped from market instruments, specified by using a scalar character vector.

When using the `bootstrap`, the choice of the `Type` parameter can impact the curve construction because it will affect the type of data that will be interpolated on (that is, forward rates, zero rates, or discount factors) during the bootstrapping process. So curves that are bootstrapped using different `Type` parameters undergo different bootstrapping algorithms with different interpolation methods, and they can sometimes produce different results when using the “get” functions (for example, `getForwardRates`).

Data Types: char

Settle — Settle date of interest-rate curve
serial date number | character vector

Settle date of interest-rate curve, specified using a serial date number or date character vector.

Data Types: double | char

InstrumentTypes — Instrument types
cell array of character vectors with values of 'deposit', 'futures', 'swap', 'bond', and 'fra'

Instrument types, specified using an N-by-1 cell array (where N is the number of instruments) indicating what kind of instrument is in the Instruments matrix. Acceptable values are 'deposit', 'futures', 'swap', 'bond', and 'fra'.

Data Types: char | cell

Instruments — Instruments
matrix

Instruments, specified as an N-by-3 data matrix for Instruments where the first column is Settle date using a serial date number, the second column is Maturity using a serial date number, and the third column is the market quote. The market quote represents the following for each instrument:

- deposit: rate
- futures: price (for example, 9628.54)
- swap: rate
- bond: clean price
- fra: forward rate

Note Instruments input for fra and for futures are different. Specifically, the forward rate underlying a fra starts on the start date (column 1 of Instruments) and ends on the end date (column 2 of Instruments). While the forward rate underlying a futures contract starts on the maturity date of the futures contract and ends on a date *n* months after the futures maturity, where *n* is the periodicity of the futures contract.

Data Types: double

Name-Value Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: DCurve = IRDataCurve.bootstrap('Forward',CurveSettle,InstrumentTypes,Instruments,'InterpMethod','pchip')

Name-Value Pair Arguments for All Bond Instruments

Compounding — Compounding frequency per-year for IRDataCurve object
CurveObj.Compounding (default) | possible values include: –1, 0, 1, 2, 3, 4, 6, 12.

Compounding frequency per-year for the `IRDataCurve` object, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using one of the supported values:

- `-1` = Continuous compounding
- `0` = Simple interest (no compounding)
- `1` = Annual compounding
- `2` = Semiannual compounding
- `3` = Compounding three times per year
- `4` = Quarterly compounding
- `6` = Bimonthly compounding
- `12` = Monthly compounding

Data Types: `double`

`Basis` — Day count basis of the interest-rate curve
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of the interest-rate curve, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`InterpMethod` — Interpolation method
`'linear'` (default) | character vector with values of `'linear'`, `'cubic'`, `'pchip'`, or `'spline'`

Interpolation method, specified as the comma-separated pair consisting of `'InterpMethod'` and a scalar character vector. For more information on interpolation methods, see `interp1`.

Data Types: `char`

`IRBootstrapOptionsObj` — IRBootstrapOptions object
`[]` (default) | `IRBootstrapOptions` object

IRBootstrapOptions object, specified as the comma-separated pair consisting of `'IRBootstrapOptionsObj'` and an `IRBootstrapOptions` object previously created using `IRBootstrapOptions`.

Data Types: `object`

`DiscountCurve` — `RateSpec` for curve used to discount cash flows
`[]` (default) | `RateSpec` object

`RateSpec` for curve used to discount cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and a `RateSpec` object previously created using `intenvset` or `toRateSpec`.

Data Types: `object`

Name-Value Pair Arguments for Each Bond Instrument

`InstrumentCouponRate` — Annual percentage rate to determine the coupons payable on an instrument
`[]` (default) | decimal

Annual percentage rate to determine the coupons payable on an instrument, specified as the comma-separated pair consisting of `'InstrumentCouponRate'` and a scalar decimal value.

Data Types: `double`

`InstrumentPeriod` — Coupons per year for the instrument
`2` (default) | numeric with value of `0`, `1`, `2`, `3`, `4`, `6`, and `12`

Coupons per year for the instrument, specified as the comma-separated pair consisting of `'InstrumentPeriod'` and a scalar numeric value.

Data Types: `double`

`InstrumentBasis` — Day-count basis of the instrument
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of the instrument, specified as the comma-separated pair consisting of `'InstrumentBasis'` and a scalar integer.

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)
- 11 — 30/360E (ICMA)
- 12 — actual/365 (ISDA)

- 13 — BUS/252

Note `InstrumentBasis` distinguishes a bond instrument's `Basis` value from the interest-rate curve's `Basis` value.

For more information, see “Basis” on page 2-228.

Data Types: `double`

`InstrumentEndMonthRule` — End-of-month rule
`1` (default) | logical with value `0` or `1`

End-of-month rule, specified as the comma-separated pair consisting of `'InstrumentEndMonthRule'` and a logical value. 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.

Data Types: `logical`

`InstrumentIssueDate` — Instrument issue date
`[]` (default) | string scalar | date character vector | serial date number

Instrument issue date, specified as the comma-separated pair consisting of `'InstrumentIssueDate'` and a scalar string, data character vector, or serial date number.

Data Types: `char` | `double` | `string`

`InstrumentFirstCouponDate` — Date when a bond makes its first coupon payment
cash flow payment dates are determined from other inputs (default) | string scalar | date character vector | serial date number

Date when a bond makes its first coupon payment (used when bond has an irregular first coupon period), specified as the comma-separated pair consisting of `'InstrumentFirstCouponDate'` and a scalar string, date character vector, or serial date number. When `InstrumentFirstCouponDate` and `InstrumentLastCouponDate` are both specified, `InstrumentFirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `InstrumentFirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `char` | `double` | `string`

`InstrumentLastCouponDate` — Last coupon date of a bond before the maturity date
cash flow payment dates are determined from other inputs (default) | string scalar | date character vector | serial date number

Last coupon date of a bond before the maturity date (used when bond has an irregular last coupon period), specified as the comma-separated pair consisting of `'InstrumentLastCouponDate'` and a scalar string, date character vector, or serial date number. In the absence of a specified `InstrumentFirstCouponDate`, a specified `InstrumentLastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `InstrumentLastCouponDate`, regardless of where it falls, and is followed only by the bond's

maturity cash flow date. If you do not specify a `InstrumentLastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `char` | `double` | `string` | `datetime`

`InstrumentFace` — Face or par value
`100` (default) | numeric

Face or par value, specified as the comma-separated pair consisting of `'InstrumentFace'` and a scalar numeric.

Data Types: `double`

Note When using `Instrument` name-value pairs, you can specify simple interest for an `Instrument` by specifying the `InstrumentPeriod` value as `0`. If `InstrumentBasis` and `InstrumentPeriod` are not specified for an `Instrument`, the following default values are used:

- `deposit` instrument uses `InstrumentBasis` as `2` (act/360) and `InstrumentPeriod` is `0` (simple interest).
- `futures` instrument uses `InstrumentBasis` as `2` (act/360) and `InstrumentPeriod` is `4` (quarterly).
- `swap` instrument uses `InstrumentBasis` as `2` (act/360) and `InstrumentPeriod` is `2`.
- `bond` instrument uses `InstrumentBasis` as `0` (act/act) and `InstrumentPeriod` is `2`.
- `FRA` instrument uses `InstrumentBasis` as `2` (act/360) and `InstrumentPeriod` is `4` (quarterly).

Output Arguments

`DCurve` — Interest-rate curve from market data
structure

Interest-rate curve from market data, returned as a structure.

Version History

Introduced in R2008b

See Also

`IRDataCurve` | `IRBootstrapOptions` | `toRateSpec` | `getForwardRates` | `getZeroRates` | `getDiscountFactors` | `getParYields` | `irbootstrap`

Topics

“Creating Interest-Rate Curve Objects” on page 9-4
“Bootstrap IRDataCurve Based on Market Instruments” on page 9-7
“Dual Curve Bootstrapping” on page 9-12
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

cdsoptprice

Price payer and receiver credit default swap options

Syntax

```
[Payer,Receiver] = cdsoptprice(ZeroData,ProbData,Settle,OptionMaturity,
CDSMaturity,Strike,SpreadVol)
[Payer,Receiver] = cdsoptprice( ___ ,Name,Value)
```

Description

`[Payer,Receiver] = cdsoptprice(ZeroData,ProbData,Settle,OptionMaturity, CDSMaturity,Strike,SpreadVol)` computes the price of payer and receiver credit default swap options.

Note Alternatively, you can use the `CDSOption` object to price credit default swap options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[Payer,Receiver] = cdsoptprice( ___ ,Name,Value)` computes the price of payer and receiver credit default swap options with additional options specified by one or more `Name,Value` pair arguments.

Examples

Obtain Payer and Receiver Values for a Credit Default Swap Option

Use `cdsoptprice` to generate `Payer` and `Receiver` values for a credit default swap option.

```
Settle = datenum('12-Jun-2012');
OptionMaturity = datenum('20-Sep-2012');
CDSMaturity = datenum('20-Sep-2017');
OptionStrike = 200;
SpreadVolatility = .4;

Zero_Time = [.5 1 2 3 4 5]';
Zero_Rate = [.5 .75 1.5 1.7 1.9 2.2]'/100;
Zero_Dates = daysadd(Settle,360*Zero_Time,1);
ZeroData = [Zero_Dates Zero_Rate];

Market_Time = [1 2 3 5 7 10]';
Market_Rate = [100 120 145 220 245 270]';
Market_Dates = daysadd(Settle,360*Market_Time,1);
MarketData = [Market_Dates Market_Rate];

ProbData = cdsbootstrap(ZeroData, MarketData, Settle);

[Payer,Receiver] = cdsoptprice(ZeroData, ProbData, Settle,...
OptionMaturity, CDSMaturity, OptionStrike, SpreadVolatility)
```

```
Payer = 223.5780

Receiver = 22.7460
```

Input Arguments

`ZeroData` — Zero rates

vector | `IRDataCurve` object

Zero rates, specified by using a `M`-by-2 vector of dates and zero rates or an `IRDataCurve` object of zero rates. For more information on an `IRDataCurve` object, see “Creating an IRDataCurve Object” on page 9-6..

Data Types: `double` | `object`

`ProbData` — Probability of default

array

Probability of default, specified as a `P`-by-2 array of dates and default probabilities.

Data Types: `double`

`Settle` — Settlement date

scalar for serial nonnegative date number | scalar for date character vector

Settlement date, specified as a scalar using a serial nonnegative date number or date character vector. `Settle` must be earlier than the `OptionMaturity` date.

Data Types: `double` | `char`

`OptionMaturity` — Option maturity dates

serial nonnegative date number | date character vector

Option maturity dates, specified as an `NINST`-by-1 vector using a serial nonnegative date number or date character vector.

Data Types: `double` | `char`

`CDSMaturity` — CDS maturity dates

serial nonnegative date number | date character vector

CDS maturity dates, specified as an `NINST`-by-1 vector using a serial nonnegative date number or date character vector.

Data Types: `double` | `char`

`Strike` — Option strikes in basis points

vector

Option strikes in basis points, specified as an `NINST`-by-1 vector.

Data Types: `double`

`SpreadVol` — Annualized credit spread volatilities

positive decimal

Annualized credit spread volatilities, specified an `NINST`-by-1 vector of positive decimals.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Payer,Receiver] = cdsoptprice(ZeroData,ProbData,Settle,OptionMaturity,CDSMaturity,OptionStrike, SpreadVolatility)`

`AdjustedForwardSpread` — Adjusted forward spread in basis points
unadjusted forward spread normally used for single-name CDS options (default) | vector

Adjusted forward spread in basis points, specified as the comma-separated pair consisting of `'AdjustedForwardSpread'` and an `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and an `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`BusinessDayConvention` — Business day conventions
`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a `N`-by-`1` cell array of character vectors of

business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`Knockout` — Indicator of knockout
`False` (default) | vector

Indicator of knockout, specified as the comma-separated pair consisting of `'Knockout'` and an `NINST`-by-`1` vector of boolean flags. If the credit default swaptions is a knockout, the flag is `True`, otherwise it is `False`.

Data Types: `logical`

`PayAccruedPremium` — Indicator of accrued premium
`True` (default) | vector

Indicator of accrued premium, specified as the comma-separated pair consisting of `'PayAccruedPremium'` and an `NINST`-by-`1` vector of boolean flags. If accrued premiums are paid upon default, the flag is `True`, otherwise it is `False`.

Data Types: `logical`

`Period` — Premiums per year of CDS
4 per year (default) | vector

Premiums per year of CDS, specified as the comma-separated pair consisting of `'Period'` and a `NINST`-by-`1` vector. Allowed values are `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`RecoveryRate` — Recovery rates
`0.4` per year (default) | vector

Recovery rates, specified as the comma-separated pair consisting of `'RecoveryRate'` and a `NINST`-by-`1` vector of decimal values from `0` to `1`.

Data Types: `double`

`ZeroBasis` — Basis of zero curve
`0` (actual/actual) (default) | integer from `0` to `13`

Basis of zero curve, specified as the comma-separated pair consisting of `'ZeroBasis'` and an `NINST`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`ZeroCompounding` — Compounding frequency of zero curve
2 (default) | integer with value of 1, 2, 3, 4, 6, 12, or -1

Compounding frequency of zero curve, specified as the comma-separated pair consisting of `'ZeroCompounding'` and an integer with one of the following allowed values:

- 1 — Annual compounding
- 2 — Semiannual compounding
- 3 — Compounding three times per year
- 4 — Quarterly compounding
- 6 — Bimonthly compounding
- 12 — Monthly compounding
- −1 — Continuous compounding

Note When `ZeroData` is an `IRDataCurve` object, the arguments `ZeroCompounding` and `ZeroBasis` are implicit in `ZeroData` and are redundant inside this function. In that case, specify these optional arguments when constructing the `IRDataCurve` object before calling this function.

Data Types: `double`

Output Arguments

`Payer` — Payer swap options in `Basis` points
vector

Payer swap options in `Basis` points, returned as an `NINST`-by-`1` vector of prices.

`Receiver` — Receiver swap options in `Basis` points
vector

Receiver swap options in `Basis` points, returned as an `NINST`-by-`1` vector of prices.

More About

Credit Default Swap Option

A credit default swap (CDS) option, or credit default swaption, is a contract that provides the option holder with the right, but not the obligation, to enter into a credit default swap in the future.

CDS options can either be payer swaptions or receiver swaptions. In a payer swaption, the option holder has the right to enter into a CDS in which they are paying premiums and in a receiver swaption, the option holder is receiving premiums.

Algorithms

The payer and receiver credit default swap options are computed using the Black's model as described in O'Kane [1]:

$$V_{Pay(Knockout)} = RPV01(t, t_E, T)(F\Phi(d_1) - K\Phi(d_2))$$

$$V_{Rec(Knockout)} = RPV01(t, t_E, T)(K\Phi(-d_2) - F\Phi(-d_1))$$

$$d_1 = \frac{\ln\left(\frac{F}{K}\right) + \frac{1}{2}\sigma^2(t_E - t)}{\sigma\sqrt{t_E - t}}$$

$$d_2 = d_1 - \sigma\sqrt{t_E - t}$$

$$V_{Pay(Non-Knockout)} = V_{Pay(Knockout)} + FEP$$

$$V_{Pay(Non-Knockout)} = V_{Rec(Knockout)}$$

where

RPV01 is the risky present value of a basis point (see `cdsrpv01`).

Φ is the normal cumulative distribution function.

σ is the spread volatility.

t is the valuation date.

t_E is the option expiry date.

T is the CDS maturity date.

F is the forward spread (from option expiry to CDS maturity).

K is the strike spread.

FEP is the front-end protection (from option initiation to option expiry).

Version History

Introduced in R2011a

References

[1] O'Kane, D. *Modelling Single-name and Multi-name Credit Derivatives.* Wiley, 2008, pp. 156–169.

See Also

`cdsbootstrap` | `cdsspread` | `cdsprice` | `cdsrpv01` | `IRDataCurve` | `CDSOption`

Topics

“Pricing a Single-Name CDS Option” on page 8-28
“Pricing a CDS Index Option” on page 8-30
“Price Multiple CDS Option Instruments Using CDS Black Model and CDS Black Pricer” on page 8-46
“Credit Default Swap Option” on page 8-27
“Mapping Financial Instruments Toolbox Functions for Credit Derivative Instrument Objects” on page 1-94

External Websites

Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

cmosched

Generate principal balance schedule for planned amortization class (PAC) or targeted amortization class (TAC) bond

Syntax

```
[BalanceSchedule,InitialBalance] = cmosched(Principal,Coupon,OriginalTerm,
TermRemaining,PrepaySpeed)
[BalanceSchedule,InitialBalance] = cmosched( ___ ,TranchePrincipal)
```

Description

`[BalanceSchedule,InitialBalance] = cmosched(Principal,Coupon,OriginalTerm, TermRemaining,PrepaySpeed)` generates a principal balance schedule for planned amortization class (PAC) bonds using two bands of Public Securities Association Prepayment Model (PSA) speeds or targeted amortization class (TAC) bonds using a single PSA speed.

`[BalanceSchedule,InitialBalance] = cmosched( ___ ,TranchePrincipal)` adds an optional argument for `TranchePrincipal`.

Examples

Calculate the Principal Balance Schedule for a CMO PAC Bond

Define the mortgage pool under consideration and generate a principal balance schedule for planned amortization class (PAC) bonds using two bands of PSA speeds.

```
Principal = 128687000;
GrossRate = 0.0648;
OriginalTerm = 360;
TermRemaining = 325;
PrepaySpeed = [300 525];
PacPrincipal = 100250000;

[BalanceSchedule, InitialBalance] ...
= cmosched(Principal, GrossRate, OriginalTerm, TermRemaining, ...
PrepaySpeed, PacPrincipal)
```

```
BalanceSchedule = 1×325
10^7 ×

    9.7996    9.5780    9.3602    9.1461    8.9357    8.7289    8.5257    8.3259    8.1296    7.9


InitialBalance = 100250000
```

Input Arguments

`Principal` — Principal of the underlying mortgage pool

numeric

Principal of the underlying mortgage pool, specified as a scalar numeric value.

Data Types: `double`

`Coupon` — Coupon rate of the underlying mortgage pool

decimal

Coupon rate of the underlying mortgage pool, specified as a scalar decimal value.

Data Types: `double`

`OriginalTerm` — Original term in months of the underlying mortgage pool

numeric

Original term in months of the underlying mortgage pool, specified as a scalar numeric value.

Data Types: `double`

`TermRemaining` — Terms remaining in months of the underlying mortgage pool

numeric

Terms remaining in months of the underlying mortgage pool, specified as a scalar numeric value.

Data Types: `double`

`PrepaySpeed` — PSA speed

matrix numeric | scalar numeric

PSA speed is specified as follows:

- For planned amortization class (PAC) bonds, PSA speed is specified as a 1-by-2 matrix, where the first element is the lower band and the second element is the upper band.
- For targeted amortization class (TAC) bonds, the PSA speed is specified as a scalar numeric value

Data Types: `double`

`TranchePrincipal` — Principal of the scheduled tranche

numeric

(Optional) Principal of the scheduled tranche, specified as a scalar numeric value. If `TranchePrincipal` is unspecified or empty `[]`, the principal of the scheduled tranche is assumed to be the sum of the payment schedule calculated from the PSA prepayment speeds.

Data Types: `double`

Output Arguments

`BalanceSchedule` — Number of terms remaining

matrix

Number of terms remaining, returned as a matrix of size 1-by-`NUMTERMS`, where `NUMTERMS` is the number of terms remaining. Each column contains the scheduled principal balance for the time period corresponding to the column number.

`InitialBalance` — Initial principal balance of the scheduled tranche
scalar

initial principal balance of the scheduled tranche, returned as a scalar numeric value.

More About

Planned Amortization Class (PAC) Bond

PAC bonds are a type of CMO bond and are designed to largely eliminate prepayment risk for investors.

They do this by transferring essentially all prepayment risk to other bonds in the CMO that are called support bonds.

Targeted Amortization Class (TAC) Bond

TAC bonds are analogous to PAC bonds, but are structured differently.

TAC bonds offer one-sided protection, shielding investors from high prepayment rates up to a specified PSA and do not protect against low prepayment rates.

Version History

Introduced in R2012a

References

[1] Hayre, Lakhbir, ed. *Salomon Smith Barney Guide to Mortgage-Backed and Asset-Backed Securities*. John Wiley and Sons, New York, 2001.

[2] Lyuu, Yuh-Dah. *Financial Engineering and Computation*. Cambridge University Press, 2004.

See Also

`cmoschedcf`

Topics

“Create PAC and Sequential CMO” on page 5-49
“What Are CMOs?” on page 5-40
“Prepayment Risk” on page 5-41
“CMO Workflow” on page 5-47

cmoschedcf

Generate cash flows for scheduled collateralized mortgage obligation (CMO) using PAC or TAC model

Syntax

```
[Balance,Principal,Interest] = cmoschedcf(PrincipalPayments,TranchePrincipals
TrancheCoupons,BalanceSchedule)
```

Description

`[Balance,Principal,Interest] = cmoschedcf(PrincipalPayments,TranchePrincipals TrancheCoupons,BalanceSchedule)` generates cash flows for a scheduled CMO, such as the planned amortization class (PAC) or targeted amortization class (TAC), given the underlying mortgage pool payments (or payments from another CMO tranche). The output `Balances`, `Principal`, and `Interest` from this function can be used as input into `cmoseqcf` to further divide the PAC, TAC, or support dividing a tranche into sequential tranches.

Examples

Calculate Cash Flows for Each PAC Tranche

Define the mortgage pool under consideration for CMO structuring using `mbscfamounts` or `mbspassthrough`. Calculate the underlying mortgage cash flow, define the PAC schedule and CMO tranches, and calculate the cash flows for each tranche.

```
MortgagePrincipal = 1000000; % underlying mortgage
Coupon = 0.12;
Terms = 6; % months

[PrincipalBalance, MonthlyPayments, SchedPrincipalPayments, ...
InterestPayments, Prepayments] = ...
mbspassthrough(MortgagePrincipal, Coupon, Terms, Terms, 0, []);
PrincipalPayments = SchedPrincipalPayments.' + Prepayments.'
```

```
PrincipalPayments = 1×6
10^5 ×

    1.6255    1.6417    1.6582    1.6747    1.6915    1.7084
```

Calculate the PAC schedule for CMO using `cmosched`.

```
PrepaySpeed = [100 300];
[BalanceSchedule, InitialBalance] ...
= cmosched(MortgagePrincipal, Coupon, Terms, Terms, PrepaySpeed, [])
```

```
BalanceSchedule = 1×6
10^5 ×

    8.3617    6.7180    5.0581    3.3828    1.6955         0
```

```
InitialBalance = 9.9886e+05
```

Define CMO tranches.

```
TranchePrincipals = ...
[InitialBalance; MortgagePrincipal-InitialBalance];
TrancheCoupons = [0.12; 0.12];
```

Calculate cash flows for each tranche.

```
[Balance, Principal, Interest] = ...
cmoschedcf(PrincipalPayments, TranchePrincipals, ...
TrancheCoupons, BalanceSchedule)
```

```
Balance = 2×6
10^5 ×

    8.3631    6.7213    5.0632    3.3885    1.6970         0
    0.0114    0.0114    0.0114    0.0114    0.0114    0.0000


Principal = 2×6
10^5 ×

    1.6255    1.6417    1.6582    1.6747    1.6915    1.6970
         0         0         0         0         0    0.0114


Interest = 2×6
10^3 ×

    9.9886    8.3631    6.7213    5.0632    3.3885    1.6970
    0.0114    0.0114    0.0114    0.0114    0.0114    0.0114
```

Input Arguments

`PrincipalPayments` — Number of terms remaining for underlying principal payments

numeric matrix

Number of terms remaining for underlying principal payments, specified as a matrix of size 1-by-`NUMTERMS`, where `NUMTERMS` is the number of terms remaining. Each column contains the underlying principal payment for the time period corresponding to the row number. Calculate underlying principal payments using `mbscfamounts` or `mbspassthrough`. The underlying principal payments can also be outputs from other CMO cash flow functions.

Data Types: `double`

`TranchePrincipals` — Initial principal for the scheduled and the support tranche

numeric matrix

Initial principal for the scheduled and the support tranche, specified as a matrix of size 2-by-1.

Data Types: `double`

`TrancheCoupons` — Coupons for the schedule tranche and the support tranche

matrix of coupon values

Coupons for the schedule tranche and the support tranche, specified as a matrix of size 2-by-1 of coupon values. The weighted average coupon for the CMO should not exceed the coupon of the underlying mortgage.

Data Types: `double`

`BalanceSchedule` — Number of terms remaining for targeted balance
numeric matrix

Number of terms remaining for targeted balance, specified as a matrix of size 1-by-`NUMTERMS`, where `NUMTERMS` is the number of terms remaining. Each element represents the targeted balance schedule for the time period corresponding to that column.

Data Types: `double`

Output Arguments

`Balance` — Number of terms remaining and principal balances
matrix

Number of terms remaining and principal balances, returned as a matrix of size 2-by-`NUMTERMS`, where `NUMTERMS` is the number of terms remaining. The first row is the principal balances of the scheduled tranche, and the second row is the principal balances of the support tranche at the time period corresponding to the column.

`Principal` — Number of terms remaining and principal payments
matrix

Number of terms remaining and principal payments, returned as a matrix of size 2-by-`NUMTERMS`, where `NUMTERMS` is the number of terms remaining. The first row is the principal payments of the scheduled tranche, and the second row is the principal payments of the support tranche at the time period corresponding to the column.

`Interest` — Number of terms remaining and interest payments
scalar

Number of terms remaining and interest payments, returned as a matrix of size 2-by-`NUMTERMS`, where `NUMTERMS` is the number of terms remaining. The first row is the interest payments of the schedule tranche, and the second row is the interest payments of the support tranche at the time period corresponding to the column.

More About

Planned Amortization Class (PAC) Tranches

In a PAC CMO, there is a main tranche, known as the schedule tranche, and a support tranche.

The main purpose of a schedule tranche is to give investors in the PAC tranche a more certain cash flow.

Targeted Amortization Class (TAC) Tranches

TACs are like PACs, but principal payment is specified for only one prepayment rate.

If prepayment rates are higher or lower, then the principal payment to TAC holders are higher or lower accordingly.

Schedule and Support Tranche

The main purpose of a PAC tranche is to give investors in the PAC tranche a more certain cash flow.

The PAC tranche receives priority for receiving payments of principal and interest that gives investors in the PAC tranche a steadier income. If prepayments differ from what was expected, then the support tranche gets the variable portion of the payments. While income to the support tranche is more variable, it is also higher yielding. Estimates of the yield, average life, and lockout periods of the PAC tranche is more certain.

Version History

Introduced in R2012a

References

[1] Hayre, Lakhbir, ed. *Salomon Smith Barney Guide to Mortgage-Backed and Asset-Backed Securities*. John Wiley and Sons, New York, 2001.

[2] Lyuu, Yuh-Dah. *Financial Engineering and Computation*. Cambridge University Press, 2004.

See Also

`cmoseqcf` | `cmosched` | `mbscfamounts` | `mbspassthrough`

Topics

“Create PAC and Sequential CMO” on page 5-49
“What Are CMOs?” on page 5-40
“Prepayment Risk” on page 5-41
“CMO Workflow” on page 5-47

cmoseqcf

Generate cash flows for sequential collateralized mortgage obligation (CMO)

Syntax

```
[Balance,Principal,Interest] = cmoseqcf(PrincipalPayments,TranchePrincipals
TrancheCoupons)
[Balance,Principal,Interest] = cmoseqcf( ___ ,HasZ)
```

Description

`[Balance,Principal,Interest] = cmoseqcf(PrincipalPayments,TranchePrincipals TrancheCoupons)` generates cash flows for a sequential CMO without a Z-bond, given the underlying mortgage pool payments.

`[Balance,Principal,Interest] = cmoseqcf( ___ ,HasZ)` generates cash flows for a sequential CMO with a Z-bond, given the underlying mortgage pool payments, by adding an additional optional input for `HasZ`.

Examples

Calculate Cash Flows for a Sequential Collateralized Mortgage Obligation (CMO)

Define the mortgage pool under consideration for CMO structuring using `mbscfamounts` or `mbspassthrough` and calculate the cash flows with an A and B tranche for a sequential CMO.

```
MortgagePrincipal = 1000000;
Coupon = 0.12;
Terms = 6; % months

% Calculate underlying mortgage cash flows
[PrincipalBalance, MonthlyPayments, SchedPrincipalPayments, ...
InterestPayments, Prepayments] = ...
mbspassthrough(MortgagePrincipal, Coupon, Terms, Terms, 0, []);
PrincipalPayments = SchedPrincipalPayments.' + Prepayments.'
```

```
PrincipalPayments = 1×6
10^5 ×

    1.6255    1.6417    1.6582    1.6747    1.6915    1.7084
```

Define CMO tranches, A and B.

```
TranchePrincipals = [500000; 500000];
TrancheCoupons = [0.12; 0.12];
```

Calculate cash flows for each tranche.

```
[Balance, Principal, Interest] = ...
cmoseqcf(PrincipalPayments, TranchePrincipals, TrancheCoupons, false)
```

```
Balance = 2×6
10^5 ×

    3.3745    1.7328    0.0746         0         0         0
    5.0000    5.0000    5.0000    3.3999    1.7084    0.0000


Principal = 2×6
10^5 ×

    1.6255    1.6417    1.6582    0.0746         0         0
         0         0         0    1.6001    1.6915    1.7084


Interest = 2×6
10^3 ×

    5.0000    3.3745    1.7328    0.0746         0         0
    5.0000    5.0000    5.0000    5.0000    3.3999    1.7084
```

Input Arguments

`PrincipalPayments` — Number of terms remaining for underlying principal payments

numeric matrix

Number of terms remaining for underlying principal payments, specified as a matrix of size 1-by-`NUMTERMS`, where `NUMTERMS` is the number of terms remaining. Each column contains the underlying principal payment for the time period corresponding to the row number. Calculate underlying principal payments using `mbscfamounts` or `mbspassthrough`. The underlying principal payments can also be outputs from other CMO cash flow functions.

Data Types: `double`

`TranchePrincipals` — Initial principal for each tranche

numeric matrix

Initial principal for each tranche, specified as a matrix of size `NUMTRANCHES`-by-1, where `NUMTRANCHES` is the number of tranches in the sequential CMO. Each element of the matrix represents the initial principal for each tranche. If the sequential CMO includes a Z-bond (`HasZ` is `true`), the last element of this matrix is the principal of the Z-bond.

Data Types: `double`

`TrancheCoupons` — Coupon for each tranche

matrix of coupon values

Coupon for each tranche, specified as a matrix of size `NUMTRANCHES`-by-1, where `NUMTRANCHES` is the number of tranches in the sequential CMO. Each element of the matrix represents the coupon for each tranche. If the sequential CMO includes a Z-bond (`HasZ` is `true`), the last element of this matrix is the coupon of the Z-bond. The weighted average coupon for the CMO should not exceed the coupon of the underlying mortgage.

Data Types: `double`

`HasZ` — Indicates that the sequential CMO contains a Z-bond

`false` (default) | `true` | `false`

(Optional) Indicates that the sequential CMO contains a Z-bond, specified as a Boolean (`true` or `false`). A value of `true` indicates that the sequential CMO contains a Z-bond, and the last element of `TranchePrincipals` and `TrancheCoupons` is treated as that of the Z-bond. A value of `false` indicates that there is no Z-bond in the sequential CMO, and the last element of `TranchePrincipals` and `TrancheCoupons` is treated as an ordinary tranche.

Data Types: `logical`

Output Arguments

`Balance` — Principal balance for time period and tranche

matrix

Principal balance for time period and tranche, returned as a matrix of size `NUMTRANCHES`-by-`NUMTERMS`, where `NUMTRANCHES` is the number of terms remaining and `NUMTRANCHES` is the number of tranches. Each element represents the principal balance at the time period corresponding to the column, and for the tranche corresponding to the row.

`Principal` — Principal payments for time period and tranche

matrix

Principal payments for time period and tranche, returned as a matrix of size `NUMTRANCHES`-by-`NUMTERMS`, where `NUMTRANCHES` is the number of terms remaining and `NUMTRANCHES` is the number of tranches. Each element represents the principal payments made at the time period corresponding to the column, and to the tranche corresponding to the row.

`Interest` — Interest payments for time period and tranche

matrix

Interest payments for time period and tranche, returned as a matrix of size `NUMTRANCHES`-by-`NUMTERMS`, where `NUMTRANCHES` is the number of terms remaining and `NUMTRANCHES` is the number of tranches. Each element represents the interest payments made at the time period corresponding to the column, and to the tranche corresponding to the row.

More About

Sequential Pay CMO

A sequential pay CMO involves tranches that pay off principal sequentially.

For example, consider the following case, where all principal from the underlying mortgage pool is repaid on tranche A first, then tranche B, then tranche C. Interest is paid on each tranche as long as the principal for the tranche has not been retired.

CMO Tranche

Tranche is a term often used to describe a specific class of bonds within an offering wherein each tranche offers varying degrees of risk to the investor.

Version History

Introduced in R2012a

References

[1] Hayre, Lakhbir, ed. *Salomon Smith Barney Guide to Mortgage-Backed and Asset-Backed Securities*. John Wiley and Sons, New York, 2001.

[2] Lyuu, Yuh-Dah. *Financial Engineering and Computation*. Cambridge University Press, 2004.

See Also

`cmoschedcf` | `cmosched` | `mbscfamounts` | `mbspassthrough`

Topics

“Create PAC and Sequential CMO” on page 5-49
“What Are CMOs?” on page 5-40
“Prepayment Risk” on page 5-41
“CMO Workflow” on page 5-47

convfactor

Bond conversion factors

Syntax

```
CF = convfactor(RefDate,Maturity,CouponRate)
CF = convfactor( ___ ,Name,Value)
```

Description

`CF = convfactor(RefDate,Maturity,CouponRate)` computes a conversion factor for a bond futures contract.

`CF = convfactor( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Conversion Factors For a Bond Futures Contract

This example shows how to calculate `CF`, given the following `RefDate`, `Maturity`, and `CouponRate`.

```
RefDate  = [datetime(2002,12,1);
               datetime(2003,3,1);
               datetime(2003,6,1);
               datetime(2003,9,1);
               datetime(2003,12,1);
               datetime(2003,9,1);
               datetime(2002,12,1);
               datetime(2003,6,1)];

Maturity = [datetime(2012,11,15);
               datetime(2012,8,15);
               datetime(2012,2,15);
               datetime(2011,2,15);
               datetime(2011,8,15);
               datetime(2010,8,15);
               datetime(2009,8,15);
               datetime(2010,2,15)];

CouponRate = [0.04; 0.04375; 0.04875; 0.05; 0.05; 0.0575; 0.06; 0.065];

CF = convfactor(RefDate, Maturity, CouponRate)
```

```
CF = 8×1

    0.8539
    0.8858
    0.9259
    0.9418
    0.9403
```

```
    0.9862
    1.0000
    1.0266
```

Compute the Conversion Factor For a German Bond

This example shows how to calculate `cf`, given the following `RefDate`, `Maturity`, and `CouponRate` for a German bond.

```
cf = convfactor(datetime(2009,3,10),datetime(2018,1,4), .04,.06,3)
```

```
cf = 0.8659
```

Input Arguments

`RefDate` — Reference dates
datetime array | string array | date character vector

Reference dates for which conversion factor is computed (usually the first day of delivery months), specified as an `N`-by-`1` vector using a datetime array, string array, or date character vectors

To support existing code, `convfactor` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `N`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `convfactor` also accepts serial date numbers as inputs, but they are not recommended.

`CouponRate` — Annual coupon rates for underlying bond
vector in decimals

Annual coupon rates for underlying bond, specified as an `numBonds`-by-`1` vector in decimals.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `CF = convfactor(RefDate,Maturity,CouponRate,'Convention',2)`

`Convention` — Conversion factor convention
1 US Treasury bond (30-year) and Treasury note (10-year) futures contract (default) | integer from 1 to 5

Conversion factor convention, specified as the comma-separated pair consisting of `'Convention'` and a `N`-by-`1` vector using the following values:

- `1` = US Treasury bond (30-year) and Treasury note (10-year) futures contract
- `2` = US 2-year and 5-year Treasury note futures contract
- `3` = German Bobl, Bund, Buxl, and Schatz
- `4` = UK gilts
- `5` = Japanese Government Bonds (JGBs)

Data Types: `double`

`FirstCouponDate` — Irregular first coupon date
datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a `N`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `convfactor` also accepts serial date numbers as inputs, but they are not recommended.

`RefYield` — Reference semiannual yield
`0.06` (6%) (default) | vector in decimals

Reference semiannual yield, specified as the comma-separated pair consisting of `'RefYield'` and an `N`-by-`1` vector in decimals.

Data Types: `double`

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a `N`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `convfactor` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`CF` — Conversion factors against the 6% yield par-bond
vector

Conversion factors against the 6% yield par-bond, returned as an `N`-by-`1` vector.

More About

Conversion Factors

Conversion factors of US Treasury bonds and other government bonds are based on a bond yielding 6%.

Optionally, you can specify other types of bonds and yields using inputs for `RefYield` and `Convention`. For US Treasury bonds, verify the output of `convfactor` by comparing the output against the quotations provided by the Chicago Board of Trade (`https://www.cmegroup.com/company/cbot.html`).

For German bonds, verify the output of `convfactor` by comparing the output against the quotations provided by Eurex (`https://www.eurexchange.com`).

For UK Gilts, verify the output of `convfactor` by comparing the output against the quotations provided by Euronext (`https://www.euronext.com`).

For Japanese Government Bonds, verify the output of `convfactor` by comparing the output against the quotations provided by the Tokyo Stock Exchange (`https://www.jpx.co.jp/english/`).

Version History

Introduced in R2009b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `convfactor` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Burghardt, G., T. Belton, M. Lane, and J. Papa. *The Treasury Bond Basis.* McGraw-Hill, 2005.

[2] Krgin, Dragomir. *Handbook of Global Fixed Income Calculations.* John Wiley & Sons, 2002.

See Also

`tfutbyprice` | `tfutbyyield` | `tfutimprepo` | `bndfutimprepo` | `bndfutprice`

Topics

“Analysis of Bond Futures” on page 7-12

“Fitting the Diebold Li Model” on page 2-150
“Managing Present Value with Bond Futures” on page 7-14

fitFunction

Custom fit interest-rate curve object to bond market data

Syntax

```
CurveObj = fitFunction(Type,Settle,FunctionHandle,Instruments,
IRFitOptionsObj)
CurveObj = fitFunction( ___ ,Name,Value)
```

Description

`CurveObj = fitFunction(Type,Settle,FunctionHandle,Instruments, IRFitOptionsObj)` fits a bond to a custom fitting function.

`CurveObj = fitFunction( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Fit a Bond Using Custom Fitting Function

This example shows how to use `fitFunction` to custom fit a bond.

```
Settle = repmat(datenum('30-Apr-2008'),[6 1]);
Maturity = [datenum('07-Mar-2009');datenum('07-Mar-2011');...
datenum('07-Mar-2013');datenum('07-Sep-2016');...
datenum('07-Mar-2025');datenum('07-Mar-2036')];
CleanPrice = [100.1;100.1;100.8;96.6;103.3;96.3];
CouponRate = [0.0400;0.0425;0.0450;0.0400;0.0500;0.0425];
Instruments = [Settle Maturity CleanPrice CouponRate];
CurveSettle = datenum('30-Apr-2008');
OptOptions = optimoptions('lsqnonlin','display','iter');
functionHandle = @(t,theta) polyval(theta,t);

CustomModel = IRFunctionCurve.fitFunction('Zero', CurveSettle, ...
functionHandle,Instruments, ...
IRFitOptions([.05 .05 .05],'FitType','price',...
'OptOptions',OptOptions))
```

```
                                        Norm of      First-order
 Iteration  Func-count     f(x)          step          optimality
     0          4         38036.7                     4.92e+04
     1          8         38036.7             10      4.92e+04
     2         12         38036.7            2.5      4.92e+04
     3         16         38036.7          0.625      4.92e+04
     4         20         38036.7        0.15625      4.92e+04
     5         24         30741.5      0.0390625      1.72e+05
     6         28         30741.5       0.078125      1.72e+05
     7         32         30741.5      0.0195312      1.72e+05
     8         36         28713.6     0.00488281      2.33e+05
     9         40         20323.3     0.00976562      9.47e+05
    10         44         20323.3      0.0195312      9.47e+05
```

```
    11         48         20323.3     0.00488281       9.47e+05
    12         52         20323.3      0.0012207       9.47e+05
    13         56         19698.8    0.000305176       1.08e+06
    14         60           17493    0.000610352          7e+06
    15         64           17493      0.0012207          7e+06
    16         68           17493    0.000305176          7e+06
    17         72         15455.1    7.62939e-05       2.25e+07
    18         76         15455.1    0.000177499       2.25e+07
    19         80         13317.1     3.8147e-05       3.18e+07
    20         84         12865.3    7.62939e-05       7.83e+07
    21         88         11779.8    7.62939e-05       7.58e+06
    22         92         11747.6    0.000152588       1.45e+05
    23         96         11720.9    0.000305176       2.33e+05
    24        100         11667.2    0.000610352       1.48e+05
    25        104         11558.6      0.0012207       3.55e+05
    26        108         11335.5     0.00244141       1.57e+05
    27        112         10863.8     0.00488281       6.36e+05
    28        116         9797.14     0.00976562       2.53e+05
    29        120         6882.83      0.0195312       9.18e+05
    30        124         6882.83      0.0373993       9.18e+05
    31        128         3218.45     0.00934981       1.96e+06
    32        132         612.703      0.0186996       3.01e+06
    33        136         13.0998      0.0253882       3.05e+06
    34        140       0.0762922     0.00154002       5.05e+04
    35        144       0.0731652    3.61102e-06           29.9
    36        148       0.0731652    6.32344e-08          0.063

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

CustomModel =
            Type: Zero
          Settle: 733528 (30-Apr-2008)
    Compounding: 2
           Basis: 0 (actual/actual)
```

Input Arguments

Type — Type of interest-rate curve

character vector with value of `'zero'`, `'forward'`, or `'discount'`

Type of interest-rate curve, specified by using a scalar character vector.

Data Types: `char`

Settle — Settle date of interest-rate curve

serial date number | date character vector

Settle date of interest-rate curve, specified using a scalar serial date number or date character vector.

Data Types: `double` | `char`

FunctionHandle — Function handle that defines the interest-rate curve

function handle

Function handle that defines the interest-rate curve, specified using a function handle. The function handle takes two numeric vectors (time-to-maturity and a vector of function coefficients) and returns one numeric output (interest rate or discount factor). For more information on defining a function handle, see the MATLAB Programming Fundamentals documentation.

Data Types: `function_handle`

`Instruments` — Instruments
matrix

Instruments, specified using an N-by-4 data matrix where the first column is `Settle` date using a serial date number, the second column is `Maturity` using a serial date number, the third column is the clean price, and the fourth column is a `CouponRate` for the bond.

Data Types: `double`

`IRFitOptionsObj` — IRFitOptions object
`IRFitOptions` object

`IRFitOptions` object, specified using previously created object using `IRFitOptions`.

Data Types: `object`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `CurveObj = IRFunctionCurve.fitFunction('Zero',CurveSettle,functionHandle,Instruments,IRFitOptions([.05 .05 .05],'FitType','price','OptOptions',OptOptions))`

Name-Value Pair Arguments for All Bond Instruments

`Compounding` — Compounding frequency per-year for `IRFunctionCurve` object
`CurveObj.Compounding` (default) | possible values include: –1, 0, 1, 2, 3, 4, 6, 12.

Compounding frequency per-year for the `IRFunctionCurve` object, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using one of the supported values:

- `–1` = Continuous compounding
- `0` = Simple interest (no compounding)
- `1` = Annual compounding
- `2` = Semiannual compounding
- `3` = Compounding three times per year
- `4` = Quarterly compounding
- `6` = Bimonthly compounding
- `12` = Monthly compounding

Data Types: `double`

`Basis` — Day count basis of the bond

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of the bond, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Name-Value Pair Arguments for Each Bond Instrument

`InstrumentPeriod` — Coupons per year for the bond

`2` (default) | numeric with value of `0`, `1`, `2`, `3`, `4`, `6`, and `12`

Coupons per year for the bond, specified as the comma-separated pair consisting of `'InstrumentPeriod'` and a scalar numeric value.

Data Types: `double`

`InstrumentBasis` — Day-count basis of the bond

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of the bond, specified as the comma-separated pair consisting of `'InstrumentBasis'` and a scalar integer.

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)

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

Note `InstrumentBasis` distinguishes a bond instrument's `Basis` value from the interest-rate curve's `Basis` value.

For more information, see “Basis” on page 2-228.

Data Types: `double`

`InstrumentEndMonthRule` — End-of-month rule
`1` (default) | logical with value `0` or `1`

End-of-month rule, specified as the comma-separated pair consisting of `'InstrumentEndMonthRule'` and a logical value. 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.

Data Types: `logical`

`InstrumentIssueDate` — Instrument issue date
`[]` (default) | serial date number | date character vector

Instrument issue date, specified as the comma-separated pair consisting of `'InstrumentIssueDate'` and a scalar serial date number or date character vector.

Data Types: `double` | `char`

`InstrumentFirstCouponDate` — Date when a bond makes its first coupon payment
cash flow payment dates are determined from other inputs (default) | serial date number | date character vector

Date when a bond makes its first coupon payment (used when bond has an irregular first coupon period), specified as the comma-separated pair consisting of `'InstrumentFirstCouponDate'` and a scalar serial date number or date character vector. When `InstrumentFirstCouponDate` and `InstrumentLastCouponDate` are both specified, `InstrumentFirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `InstrumentFirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char`

`InstrumentLastCouponDate` — Last coupon date of a bond before the maturity date
cash flow payment dates are determined from other inputs (default) | serial date number | date character vector

Last coupon date of a bond before the maturity date (used when bond has an irregular last coupon period), specified as the comma-separated pair consisting of `'InstrumentLastCouponDate'` and a scalar serial date number or date character vector. In the absence of a specified `InstrumentFirstCouponDate`, a specified `InstrumentLastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `InstrumentLastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `InstrumentLastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char`

`InstrumentFace` — Face or par value
`100` (default) | numeric

Face or par value, specified as the comma-separated pair consisting of `'InstrumentFace'` and a scalar numeric.

Data Types: `double`

Note When using `Instrument` name-value pairs, you can specify simple interest for a bond by specifying the `InstrumentPeriod` value as `0`. If `InstrumentBasis` and `InstrumentPeriod` are not specified for a bond, the following default values are used: `InstrumentBasis` is `0` (act/act) and `InstrumentPeriod` is `2`.

Output Arguments

`CurveObj` — Curve model
structure

Curve model, returned as a structure.

Version History

Introduced in R2008b

See Also

`IRFunctionCurve` | `IRFitOptions`

Topics

“Creating an IRFunctionCurve Object” on page 9-16
“Fitting Interest-Rate Curve Functions” on page 9-24
“Using fitFunction to Create Custom Fitting Function” on page 9-21
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Creating Interest-Rate Curve Objects” on page 9-4
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

External Websites

Calibration and Simulation Best Practices: Multifactor Interest Rate Models for Risk Applications (30 min 00 sec)

fitNelsonSiegel

Fit Nelson-Siegel function to bond market data

Note `fitNelsonSiegel` for an `IRFunctionCurve` is not recommended. Use `fitNelsonSiegel` with a `parametercurve` object instead. For more information, see `fitNelsonSiegel`.

Syntax

```
CurveObj = IRFunctionCurve.fitNelsonSiegel(Type,Settle,Instruments)
CurveObj = IRFunctionCurve.fitNelsonSiegel(___,Name,Value)
```

Description

`CurveObj = IRFunctionCurve.fitNelsonSiegel(Type,Settle,Instruments)` fits a Nelson-Siegel function to market data for a bond.

`CurveObj = IRFunctionCurve.fitNelsonSiegel(___,Name,Value)` adds optional name-value pair arguments.

Examples

Use the Nelson-Siegel Function to Fit Bond Market Data

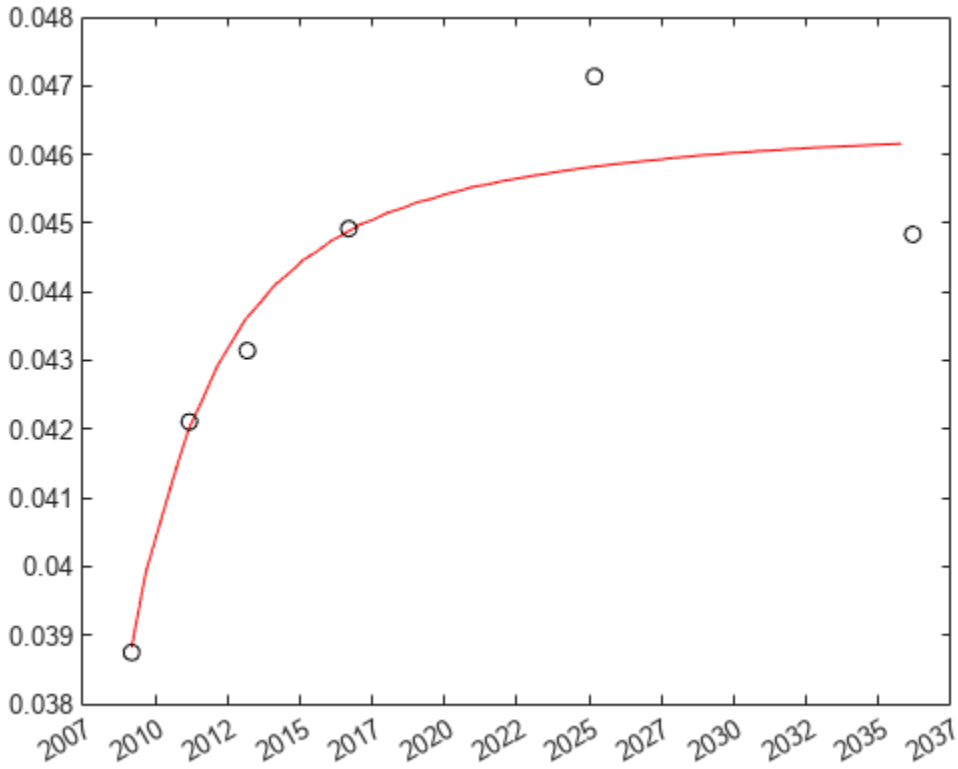
This example shows how to use the Nelson-Siegel function to fit bond market data.

```
Settle = repmat(datenum('30-Apr-2008'),[6 1]);
Maturity = [datenum('07-Mar-2009');datenum('07-Mar-2011');...
datenum('07-Mar-2013');datenum('07-Sep-2016');...
datenum('07-Mar-2025');datenum('07-Mar-2036')];

CleanPrice = [100.1;100.1;100.8;96.6;103.3;96.3];
CouponRate = [0.0400;0.0425;0.0450;0.0400;0.0500;0.0425];
Instruments = [Settle Maturity CleanPrice CouponRate];
PlottingPoints = datenum('07-Mar-2009'):180:datenum('07-Mar-2036');
Yield = bndyield(CleanPrice,CouponRate,Settle,Maturity);

NSModel = IRFunctionCurve.fitNelsonSiegel('Zero',datenum('30-Apr-2008'),Instruments);

NSModel.Parameters
```

```
ans = 1×4

    4.6617   -1.0227   -0.3484    1.2386
```

```
% create the plot
plot(PlottingPoints, getParYields(NSModel, PlottingPoints),'r')
hold on
```

```
scatter(Maturity,Yield,'black')
datetick('x')
```



Input Arguments

Type — Type of interest-rate curve for a bond

character vector with value of `'zero'` or `'forward'`

Type of interest-rate curve for a bond, specified by using a scalar character vector.

Data Types: `char`

Settle — Settle date of interest-rate curve

serial date number | date character vector

Settle date of interest-rate curve, specified using a scalar serial date number or date character vector.

Data Types: `double` | `char`

Instruments — Instruments

matrix

Instruments, specified using an N-by-4 data matrix where the first column is `Settle` date using a serial date number, the second column is `Maturity` using a serial date number, the third column is the clean price, and the fourth column is a `CouponRate` for the bond.

Data Types: `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `CurveObj = IRFunctionCurve.fitNelsonSiegel('Zero',datenum('30-Apr-2008'),Instruments)`

Name-Value Pair Arguments for All Bond Instruments

`Compounding` — Compounding frequency per-year for `IRFunctionCurve` object
`CurveObj.Compounding` (default) | possible values include: –1, 0, 1, 2, 3, 4, 6, 12.

Compounding frequency per-year for the `IRFunctionCurve` object, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using one of the supported values:

- `–1` = Continuous compounding
- `0` = Simple interest (no compounding)
- `1` = Annual compounding
- `2` = Semiannual compounding
- `3` = Compounding three times per year
- `4` = Quarterly compounding
- `6` = Bimonthly compounding
- `12` = Monthly compounding

Data Types: `double`

`Basis` — Day count basis of the interest-rate curve
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of the interest-rate curve, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`IRFitOptions` — IRFitOptions object

`IRFitOptions` object

`IRFitOptions` object, specified using previously created object using `IRFitOptions`. When using `IRFitOption`, the default `FitType` is `DurationWeightedPrice`. Duration weighted price refers to the form of the objective function that needs to be minimized to find the optimal Nelson-Siegel parameters. Specifically, this objective function minimizes using the following three algorithms:

- The difference between observed and model-predicted yields for each bond, `ObsY_`*i* – `PredY_`*i*
- The difference between observed and model-predicted prices for each bond, `ObsP_`*i* – `PredP_`*i*
- The difference between observed and model-predicted prices, weighted by the inverse of the duration of each bond (`ObsP_`*i* – `PredP_`*i*) / `D_`*i*. Weighting price by inverse duration converts the pricing errors into yield fitting errors, to a first approximation.

Data Types: `object`

Name-Value Pair Arguments for Each Bond Instrument

`InstrumentPeriod` — Coupons per year for the bond

2 (default) | numeric with value of `0`, `1`, `2`, `3`, `4`, `6`, and `12`

Coupons per year for the bond, specified as the comma-separated pair consisting of `'InstrumentPeriod'` and a scalar numeric value.

Data Types: `double`

`InstrumentBasis` — Day-count basis of the bond

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of the bond, specified as the comma-separated pair consisting of `'InstrumentBasis'` and a scalar integer.

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

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

Note `InstrumentBasis` distinguishes a bond instrument's `Basis` value from the interest-rate curve's `Basis` value.

For more information, see “Basis” on page 2-228.

Data Types: `double`

`InstrumentEndMonthRule` — End-of-month rule
`1` (default) | logical with value `0` or `1`

End-of-month rule, specified as the comma-separated pair consisting of `'InstrumentEndMonthRule'` and a logical value. 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.

Data Types: `logical`

`InstrumentIssueDate` — Instrument issue date
`[]` (default) | serial date number | date character vector

Instrument issue date, specified as the comma-separated pair consisting of `'InstrumentIssueDate'` and a scalar serial date number or date character vector.

Data Types: `double` | `char`

`InstrumentFirstCouponDate` — Date when a bond makes its first coupon payment
cash flow payment dates are determined from other inputs (default) | serial date number | date character vector

Date when a bond makes its first coupon payment (used when bond has an irregular first coupon period), specified as the comma-separated pair consisting of `'InstrumentFirstCouponDate'` and a scalar serial date number or date character vector. When `InstrumentFirstCouponDate` and `InstrumentLastCouponDate` are both specified, `InstrumentFirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `InstrumentFirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char`

`InstrumentLastCouponDate` — Last coupon date of a bond before the maturity date
cash flow payment dates are determined from other inputs (default) | serial date number | date character vector

Last coupon date of a bond before the maturity date (used when bond has an irregular last coupon period), specified as the comma-separated pair consisting of `'InstrumentLastCouponDate'` and a

scalar serial date number or date character vector. In the absence of a specified `InstrumentFirstCouponDate`, a specified `InstrumentLastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `InstrumentLastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `InstrumentLastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char`

`InstrumentFace` — Face or par value
`100` (default) | numeric

Face or par value, specified as the comma-separated pair consisting of `'InstrumentFace'` and a scalar numeric.

Data Types: `double`

Note When using `Instrument` name-value pairs, you can specify simple interest for a bond by specifying the `InstrumentPeriod` value as `0`. If `InstrumentBasis` and `InstrumentPeriod` are not specified for a bond, the following default values are used: `InstrumentBasis` is `0` (act/act) and `InstrumentPeriod` is `2`.

Output Arguments

`CurveObj` — Nelson-Siegel curve model
structure

Nelson-Siegel curve model, returned as a structure. After creating a Nelson-Siegel model, you can view the Nelson-Siegel model parameters using:

```
CurveObj.Parameters
```

where the order of parameters is [`Beta0`,`Beta1`,`Beta2`,`tau1`].

Algorithms

The Nelson-Siegel model proposes that the instantaneous forward curve can be modeled with the following:

$$f = \beta_0 + \beta_1 e^{\frac{-m}{\tau}} + \beta_2 \frac{m}{\tau} e^{\frac{-m}{\tau}}$$

This can be integrated to derive an equation for the zero curve (see [6] for more information on the equations and the derivation):

$$s = \beta_0 + (\beta_1 + \beta_2)\frac{\tau}{m}(1 - e^{\frac{-m}{\tau}}) - \beta_2 e^{\frac{-m}{\tau}}$$

See [1] for more information.

Version History

Introduced in R2008b

References

[1] Nelson, C.R., Siegel, A.F. “Parsimonious modelling of yield curves.” *Journal of Business.* Vol. 60, 1987, pp 473–89.

[2] Svensson, L.E.O. *“Estimating and interpreting forward interest rates: Sweden 1992-4.”* International Monetary Fund, IMF Working Paper, 1994/114.

[3] Fisher, M., Nychka, D., Zervos, D. *“Fitting the term structure of interest rates with smoothing splines.”* Board of Governors of the Federal Reserve System, Federal Reserve Board Working Paper 1995-1.

[4] Anderson, N., Sleath, J. *“New estimates of the UK real and nominal yield curves.”* Bank of England Quarterly Bulletin, November, 1999, pp 384–92.

[5] Waggoner, D. *“Spline Methods for Extracting Interest Rate Curves from Coupon Bond Prices.”* Federal Reserve Board Working Paper 1997–10.

[6] *“Zero-coupon yield curves: technical documentation.”* BIS Papers No. 25, October 2005.

[7] Bolder, D.J., Gusba, S. *“Exponentials, Polynomials, and Fourier Series: More Yield Curve Modelling at the Bank of Canada.”* Working Papers 2002–29, Bank of Canada.

[8] Bolder, D.J., Streliski, D. *“Yield Curve Modelling at the Bank of Canada.”* Technical Reports 84, 1999, Bank of Canada.

See Also

`IRFitOptions` | `IRFunctionCurve` | `fitSvensson` | `fitSmoothingSpline` | `fitFunction`

Topics

“Fitting IRFunctionCurve Object Using Nelson-Siegel Method” on page 9-16
“Fitting Interest-Rate Curve Functions” on page 9-24
“Using fitFunction to Create Custom Fitting Function” on page 9-21
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Creating Interest-Rate Curve Objects” on page 9-4
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

External Websites

Calibration and Simulation Best Practices: Multifactor Interest Rate Models for Risk Applications (30 min 00 sec)

fitSmoothingSpline

Fit smoothing spline to bond market data

Syntax

```
CurveObj = IRFunctionCurve.fitSmoothingSpline(Type,Settle,Instruments,
Lambdafun)
CurveObj = IRFunctionCurve.fitSmoothingSpline( ___ ,Name,Value)
```

Description

`CurveObj = IRFunctionCurve.fitSmoothingSpline(Type,Settle,Instruments, Lambdafun)` fits a smoothing spline to market data for a bond.

Note You must have a license for Curve Fitting Toolbox software to use the `fitSmoothingSpline` method.

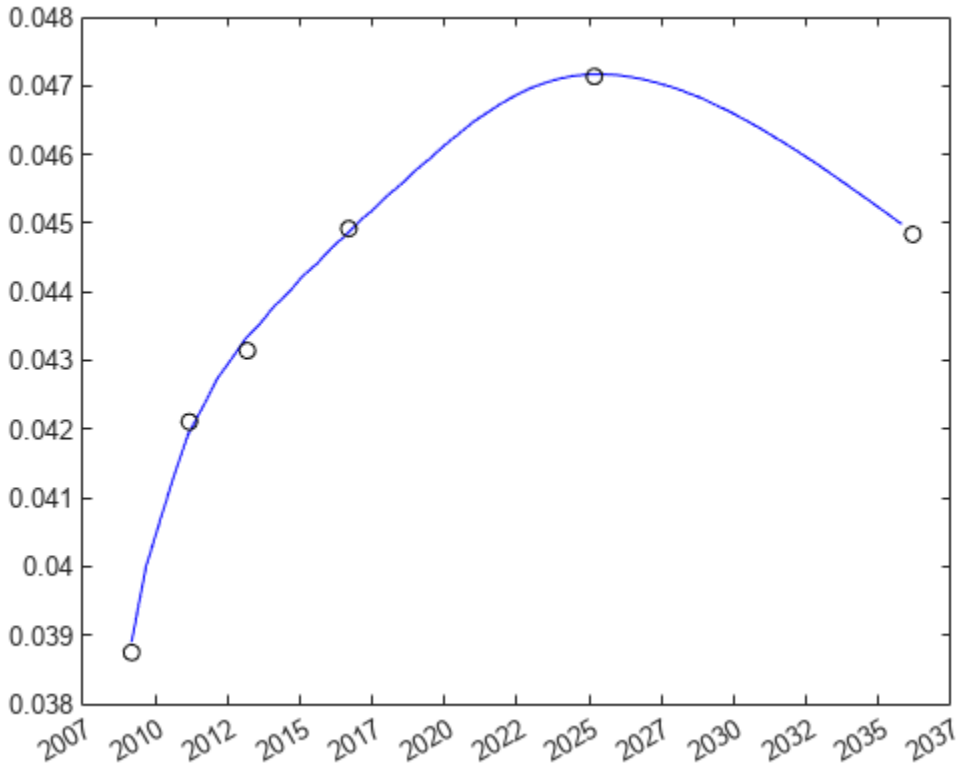
`CurveObj = IRFunctionCurve.fitSmoothingSpline( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Use a Smoothing Spline Function to Fit Market Data For a Bond

This example shows how to use a `fitSmoothingSpline` function to fit market data for a bond.

```
Settle = repmat(datenum('30-Apr-2008'),[6 1]);
Maturity = [datenum('07-Mar-2009');datenum('07-Mar-2011');...
datenum('07-Mar-2013');datenum('07-Sep-2016');...
datenum('07-Mar-2025');datenum('07-Mar-2036')];

CleanPrice = [100.1;100.1;100.8;96.6;103.3;96.3];
CouponRate = [0.0400;0.0425;0.0450;0.0400;0.0500;0.0425];
Instruments = [Settle Maturity CleanPrice CouponRate];
PlottingPoints = datenum('07-Mar-2009'):180:datenum('07-Mar-2036');
Yield = bndyield(CleanPrice,CouponRate,Settle,Maturity);

% Use the AUGKNT function to construct the knots for a cubic spline at
% every 5 years.
CustomKnots = augknt(0:5:30,4);
SmoothingModel = IRFunctionCurve.fitSmoothingSpline('Zero',datenum('30-Apr-2008'),...
Instruments,@(t) 1000,'knots', CustomKnots);

% Create the plot.
plot(PlottingPoints, getParYields(SmoothingModel, PlottingPoints),'b')
hold on
scatter(Maturity,Yield,'black')
datetick('x')
```



Fitting an `IRFunctionCurve` Object Using `fitSmoothingSpline` with Penalty Function

This example shows for to use `fitSmoothinSpline` function to fit the interest-rate curve and model the `Lambdafun` penalty function.

First, load the data.

```
load ukdata20080430
```

Convert the repo rates to be equivalent zero coupon bonds.

```
RepoCouponRate = repmat(0,size(RepoRates));
RepoPrice = bndprice(RepoRates, RepoCouponRate, RepoSettle, RepoMaturity);
```

Aggregate the data.

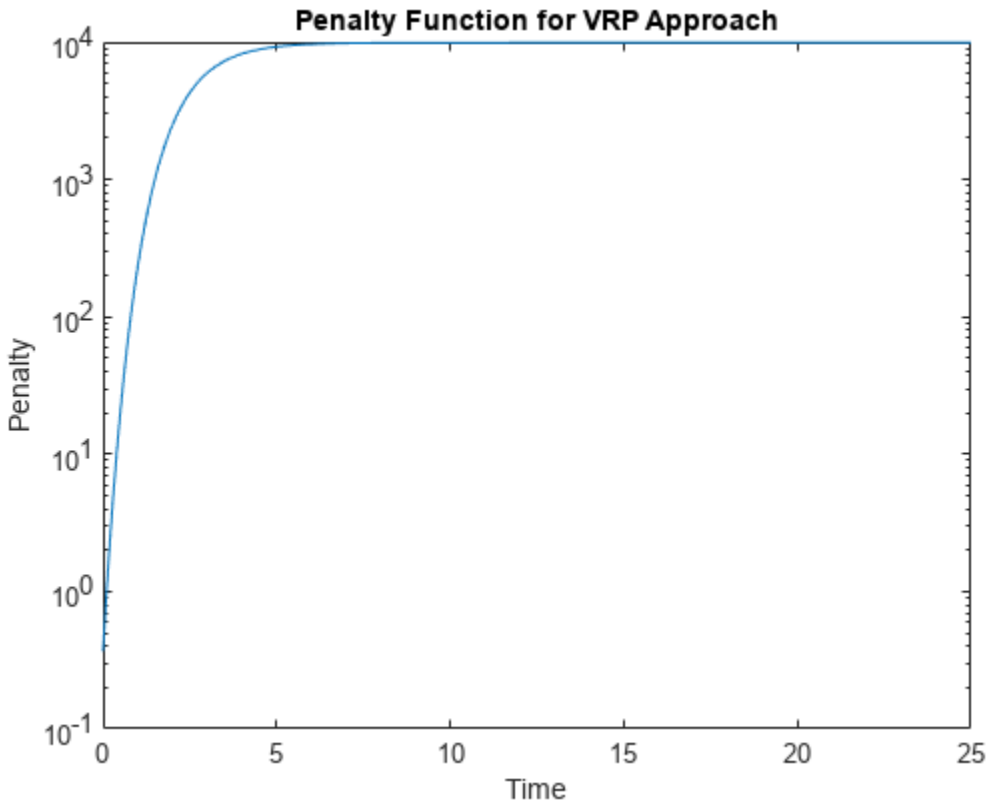
```
Settle = [RepoSettle;BondSettle];
Maturity = [RepoMaturity;BondMaturity];
CleanPrice = [RepoPrice;BondCleanPrice];
CouponRate = [RepoCouponRate;BondCouponRate];
Instruments = [Settle Maturity CleanPrice CouponRate];
InstrumentPeriod = [repmat(0,6,1);repmat(2,31,1)];
CurveSettle = datenum('30-Apr-2008');
```

Choose the parameters for the `Lambdafun` input argument.

```
L = 9.2;
S = -1;
mu = 1;
```

Define the `Lambdafun` penalty function.

```
lambdafun = @(t) exp(L - (L-S)*exp(-t/mu));
t = 0:.1:25;
y = lambdafun(t);
figure
semilogy(t,y);
title('Penalty Function for VRP Approach')
ylabel('Penalty')
xlabel('Time')
```



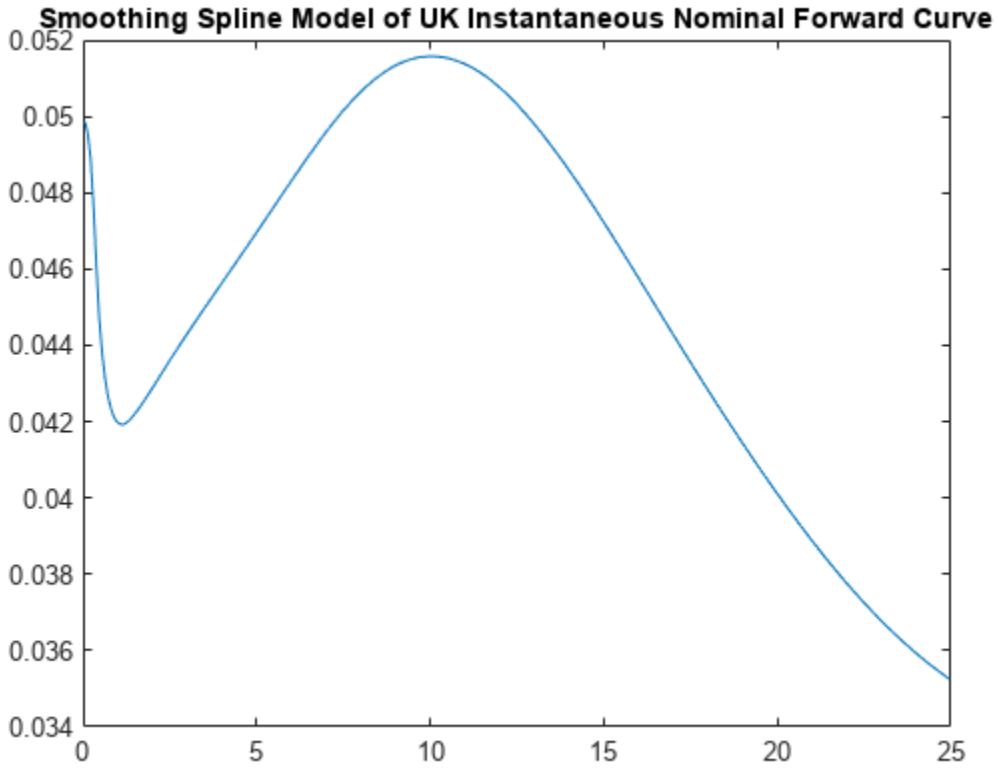
Use the `fitSmoothinSpline` function to fit the interest-rate curve and model the `Lambdafun` penalty function.

```
VRPModel = IRFunctionCurve.fitSmoothingSpline('Forward',CurveSettle,...
Instruments,lambdafun,'Compounding',-1, 'InstrumentPeriod',InstrumentPeriod)
```

```
VRPModel =
            Type: Forward
          Settle: 733528 (30-Apr-2008)
     Compounding: -1
           Basis: 0 (actual/actual)
```

Plot the smoothing spline interest-rate curve for the forward rates.

```
PlottingDates = CurveSettle+20:30:CurveSettle+365*25;
TimeToMaturity = yearfrac(CurveSettle,PlottingDates);
VRPForwardRates = getForwardRates(VRPModel, PlottingDates);
figure;plot(TimeToMaturity,VRPForwardRates)
title('Smoothing Spline Model of UK Instantaneous Nominal Forward Curve')
```



Input Arguments

Type — Type of interest-rate curve for a bond

character vector with value of `'zero'`, `'discount'`, or `'forward'`

Type of interest-rate curve for a bond, specified by using a scalar character vector.

Data Types: `char`

Settle — Settle date of interest-rate curve

serial date number | date character vector

Settle date of interest-rate curve, specified serial date number or date character vector.

Data Types: `double` | `char`

Instruments — Instruments

matrix

Instruments, specified using an N-by-4 data matrix where the first column is `Settle` date using a serial date number, the second column is `Maturity` using a serial date number, the third column is the clean price, and the fourth column is a `CouponRate` for the bond.

Data Types: `double`

`Lambdafun` — Penalty function
function handle

Penalty function, specified using a function handle. The penalty function that takes as its input time and returns a penalty value. The function handle for the penalty function takes one numeric input (time-to-maturity) and returns one numeric output (penalty to be applied to the curvature of the spline). For more information on defining a function handle, see the MATLAB Programming Fundamentals documentation.

Note The smoothing spline represents the forward curve. The spline is penalized for curvature by specifying a penalty function. This fit can only be done with a `FitType` of `DurationWeightedPrice`.

Data Types: `function_handle`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `CurveObj = IRFunctionCurve.fitSmoothingSpline('Zero',datenum('30-Apr-2008'),Instruments,@(t) 1000,'knots',CustomKnots)`

Name-Value Pair Arguments for All Bond Instruments

`Compounding` — Compounding frequency per-year for `IRFunctionCurve` object
`CurveObj.Compounding` (default) | possible values include: –1, 0, 1, 2, 3, 4, 6, 12.

Compounding frequency per-year for the `IRFunctionCurve` object, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using one of the supported values:

- `–1` = Continuous compounding
- `0` = Simple interest (no compounding)
- `1` = Annual compounding
- `2` = Semiannual compounding
- `3` = Compounding three times per year
- `4` = Quarterly compounding
- `6` = Bimonthly compounding
- `12` = Monthly compounding

Data Types: `double`

Basis — Day count basis of the interest-rate curve

0 (actual/actual) (default) | integer from 0 to 13

Day count basis of the interest-rate curve, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Name-Value Pair Arguments for Each Bond Instrument

InstrumentPeriod — Coupons per year for the bond

2 (default) | numeric with value of 0, 1, 2, 3, 4, 6, and 12

Coupons per year for the bond, specified as the comma-separated pair consisting of `'InstrumentPeriod'` and a scalar numeric value.

Data Types: `double`

InstrumentBasis — Day-count basis of the bond

0 (actual/actual) (default) | integer from 0 to 13

Day count basis of the bond, specified as the comma-separated pair consisting of `'InstrumentBasis'` and a scalar integer.

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)

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

Note `InstrumentBasis` distinguishes a bond instrument's `Basis` value from the interest-rate curve's `Basis` value.

For more information, see “Basis” on page 2-228.

Data Types: `double`

`InstrumentEndMonthRule` — End-of-month rule
`1` (default) | logical with value `0` or `1`

End-of-month rule, specified as the comma-separated pair consisting of `'InstrumentEndMonthRule'` and a logical value. 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.

Data Types: `logical`

`InstrumentIssueDate` — Instrument issue date
`[]` (default) | serial date number | date character vector

Instrument issue date, specified as the comma-separated pair consisting of `'InstrumentIssueDate'` and a scalar serial date number or date character vector.

Data Types: `double` | `char`

`InstrumentFirstCouponDate` — Date when a bond makes its first coupon payment
cash flow payment dates are determined from other inputs (default) | serial date number | date character vector

Date when a bond makes its first coupon payment (used when bond has an irregular first coupon period), specified as the comma-separated pair consisting of `'InstrumentFirstCouponDate'` and a scalar serial date number or date character vector. When `InstrumentFirstCouponDate` and `InstrumentLastCouponDate` are both specified, `InstrumentFirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `InstrumentFirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char`

`InstrumentLastCouponDate` — Last coupon date of a bond before the maturity date
cash flow payment dates are determined from other inputs (default) | serial date number | date character vector

Last coupon date of a bond before the maturity date (used when bond has an irregular last coupon period), specified as the comma-separated pair consisting of `'InstrumentLastCouponDate'` and a scalar serial date number or date character vector. In the absence of a specified `InstrumentFirstCouponDate`, a specified `InstrumentLastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `InstrumentLastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `InstrumentLastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char`

`InstrumentFace` — Face or par value
`100` (default) | numeric

Face or par value, specified as the comma-separated pair consisting of `'InstrumentFace'` and a scalar numeric.

Data Types: `double`

Note When using `Instrument` name-value pairs, you can specify simple interest for a bond by specifying the `InstrumentPeriod` value as `0`. If `InstrumentBasis` and `InstrumentPeriod` are not specified for a bond, the following default values are used: `InstrumentBasis` is `0` (act/act) and `InstrumentPeriod` is `2`.

Output Arguments

`CurveObj` — Smoothing spline curve model
structure

Smoothing spline curve model, returned as a structure.

Algorithms

The term structure can be modeled with a spline — specifically, one way to model the term structure is by representing the forward curve with a cubic spline. To ensure that the spline is sufficiently smooth, a penalty is imposed relating to the curvature (second derivative) of the spline:

$$\sum_{i=1}^{N}[\frac{P_i - \hat{P}_i(f)}{D_i}]^2 + \int_0^M \lambda_t(m)[f''(m)]^2 dm$$

where the first term is the difference between the observed price P and the predicted price, $\widehat{P}$, (weighted by the bond's duration, D) summed over all bonds in our data set and the second term is the penalty term (where λ is a penalty function and f is the spline).

See [3], [4], [5] below.

There have been different proposals for the specification of the penalty function λ. One approach, advocated by [4], and currently used by the UK Debt Management Office, is a penalty function of the following form:

$$log(\lambda(m)) = L - (L - S)e^{\frac{-m}{k}}$$

Version History

Introduced in R2008b

References

[1] Nelson, C.R., Siegel, A.F. "Parsimonious modelling of yield curves." *Journal of Business*. Vol. 60, 1987, pp 473–89.

[2] Svensson, L.E.O. *"Estimating and interpreting forward interest rates: Sweden 1992-4."* International Monetary Fund, IMF Working Paper, 1994/114.

[3] Fisher, M., Nychka, D., Zervos, D. *"Fitting the term structure of interest rates with smoothing splines."* Board of Governors of the Federal Reserve System, Federal Reserve Board Working Paper 1995-1.

[4] Anderson, N., Sleath, J. *"New estimates of the UK real and nominal yield curves."* Bank of England Quarterly Bulletin, November, 1999, pp 384–92.

[5] Waggoner, D. *"Spline Methods for Extracting Interest Rate Curves from Coupon Bond Prices."* Federal Reserve Board Working Paper 1997–10.

[6] *"Zero-coupon yield curves: technical documentation."* BIS Papers No. 25, October 2005.

[7] Bolder, D.J., Gusba, S. *"Exponentials, Polynomials, and Fourier Series: More Yield Curve Modelling at the Bank of Canada."* Working Papers 2002–29, Bank of Canada.

[8] Bolder, D.J., Streliski, D. *"Yield Curve Modelling at the Bank of Canada."* Technical Reports 84, 1999, Bank of Canada.

See Also

`IRFunctionCurve` | `fitNelsonSiegel` | `fitSvensson` | `fitFunction`

Topics

"Fitting IRFunctionCurve Object Using Smoothing Spline Method" on page 9-19
"Fitting Interest-Rate Curve Functions" on page 9-24
"Interest-Rate Curve Objects and Workflow" on page 9-2
"Creating Interest-Rate Curve Objects" on page 9-4
"Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework" on page 1-95

External Websites

Calibration and Simulation Best Practices: Multifactor Interest Rate Models for Risk Applications (30 min 00 sec)

fitSvensson

Fit Svensson function to bond market data

Note `fitSvensson` for an `IRFunctionCurve` is not recommended. Use `fitSvensson` with a `parametercurve` object instead. For more information, see `fitSvensson`.

Syntax

```
CurveObj = IRFunctionCurve.fitSvensson(Type,Settle,Instruments)
CurveObj = IRFunctionCurve.fitSvensson( ___ ,Name,Value)
```

Description

`CurveObj = IRFunctionCurve.fitSvensson(Type,Settle,Instruments)` fits the Svensson function to market data for a bond.

`CurveObj = IRFunctionCurve.fitSvensson( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

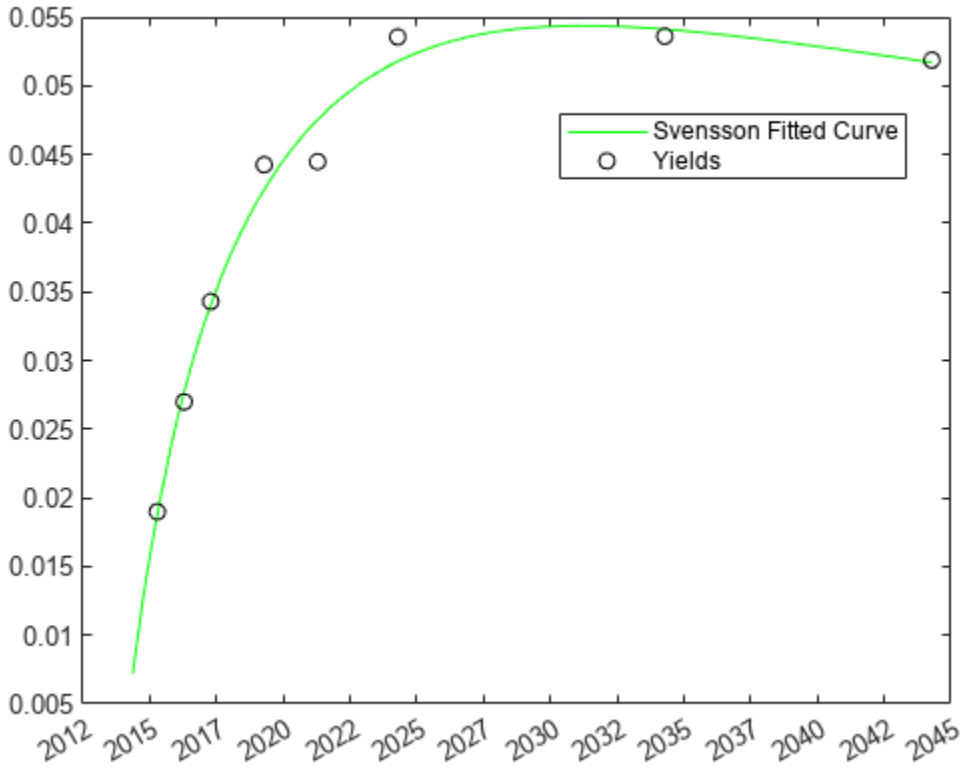
Use a Svensson Function to Fit Bond Market Data

This example shows how to use a Svensson function to fit bond market data.

```
Settle = datenum('15-Apr-2014');
Maturity = datemnth(Settle,12*[1 2 3 5 7 10 20 30]');

CleanPrice = [100.1 100.1 100.2 99.0 101.8 99.2 101.7 100.2]';
CouponRate = [0.0200 0.0275 0.035 0.042 0.0475 0.0525 0.055 0.052]';
Instruments = [repmat(Settle,size(Maturity)) Maturity CleanPrice CouponRate];
PlottingPoints = datemnth(Settle,1:360);
Yield = bndyield(CleanPrice,CouponRate,Settle,Maturity);

SvenssonModel = IRFunctionCurve.fitSvensson('Zero',Settle,Instruments);

SvenssonModel.Parameters
```

```
ans = 1×6

    1.8298   -1.2299    1.6316   12.3890    1.6982    8.9422
```

```
% create the plot
plot(PlottingPoints, getParYields(SvenssonModel, PlottingPoints),'g')
hold on
scatter(Maturity,Yield,'black')
datetick('x')
legend({'Svensson Fitted Curve','Yields'},'location','best')
```



Input Arguments

Type — Type of interest-rate curve for a bond

character vector with value of `'zero'` or `'forward'`

Type of interest-rate curve for a bond, specified by using a scalar character vector.

Data Types: `char`

Settle — Settle date of interest-rate curve

serial date number | date character vector

Settle date of interest-rate curve, specified using a scalar serial date number or date character vector.

Data Types: `double` | `char`

Instruments — Instruments

matrix

Instruments, specified using an N-by-4 data matrix where the first column is `Settle` date using a serial date number, the second column is `Maturity` using a serial date number, the third column is the clean price, and the fourth column is a `CouponRate` for the bond.

Data Types: `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `CurveObj = IRFunctionCurve.fitSvensson('Zero',Settle,Instruments)`

Name-Value Pair Arguments for All Bond Instruments

`Compounding` — Compounding frequency per-year for `IRFunctionCurve` object
`CurveObj.Compounding` (default) | possible values include: –1, 0, 1, 2, 3, 4, 6, 12.

Compounding frequency per-year for the `IRFunctionCurve` object, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using one of the supported values:

- `-1` = Continuous compounding
- `0` = Simple interest (no compounding)
- `1` = Annual compounding
- `2` = Semiannual compounding
- `3` = Compounding three times per year
- `4` = Quarterly compounding
- `6` = Bimonthly compounding
- `12` = Monthly compounding

Data Types: `double`

`Basis` — Day count basis of the interest-rate curve
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of the interest-rate curve, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)
- 11 — 30/360E (ICMA)
- 12 — actual/365 (ISDA)

- 13 — BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`IRFitOptions` — IRFitOptions object
`IRFitOptions` object

`IRFitOptions` object, specified using previously created object using `IRFitOptions`.

Data Types: `object`

Name-Value Pair Arguments for Each Bond Instrument

`InstrumentPeriod` — Coupons per year for the bond
2 (default) | numeric with value of `0`, `1`, `2`, `3`, `4`, `6`, and `12`

Coupons per year for the bond, specified as the comma-separated pair consisting of `'InstrumentPeriod'` and a scalar numeric value.

Data Types: `double`

`InstrumentBasis` — Day-count basis of the bond
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of the bond, specified as the comma-separated pair consisting of `'InstrumentBasis'` and a scalar integer.

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

Note `InstrumentBasis` distinguishes a bond instrument's `Basis` value from the interest-rate curve's `Basis` value.

For more information, see “Basis” on page 2-228.

Data Types: `double`

`InstrumentEndMonthRule` — End-of-month rule
`1` (default) | logical with value `0` or `1`

End-of-month rule, specified as the comma-separated pair consisting of `'InstrumentEndMonthRule'` and a logical value. 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.

Data Types: `logical`

`InstrumentIssueDate` — Instrument issue date
`[]` (default) | serial date number | date character vector

Instrument issue date, specified as the comma-separated pair consisting of `'InstrumentIssueDate'` and a scalar serial date number or date character vector.

Data Types: `double` | `char`

`InstrumentFirstCouponDate` — Date when a bond makes its first coupon payment
cash flow payment dates are determined from other inputs (default) | serial date number | date character vector

Date when a bond makes its first coupon payment (used when bond has an irregular first coupon period), specified as the comma-separated pair consisting of `'InstrumentFirstCouponDate'` and a scalar serial date number or date character vector. When `InstrumentFirstCouponDate` and `InstrumentLastCouponDate` are both specified, `InstrumentFirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `InstrumentFirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char`

`InstrumentLastCouponDate` — Last coupon date of a bond before the maturity date
cash flow payment dates are determined from other inputs (default) | serial date number | date character vector

Last coupon date of a bond before the maturity date (used when bond has an irregular last coupon period), specified as the comma-separated pair consisting of `'InstrumentLastCouponDate'` and a scalar serial date number or date character vector. In the absence of a specified `InstrumentFirstCouponDate`, a specified `InstrumentLastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `InstrumentLastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `InstrumentLastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char`

`InstrumentFace` — Face or par value
`100` (default) | numeric

Face or par value, specified as the comma-separated pair consisting of `'InstrumentFace'` and a scalar numeric.

Data Types: `double`

Note When using `Instrument` name-value pairs, you can specify simple interest for a bond by specifying the `InstrumentPeriod` value as `0`. If `InstrumentBasis` and `InstrumentPeriod` are not specified for a bond, the following default values are used: `InstrumentBasis` is `0` (act/act) and `InstrumentPeriod` is `2`.

Output Arguments

`CurveObj` — Svensson curve model

structure

Svensson curve model, returned as a structure. After creating a Nelson-Siegel model, you can view the Nelson-Siegel model parameters using:

```
CurveObj.Parameters
```

where the order of parameters is [`Beta0`,`Beta1`,`Beta2`,`Beta3`,`tau1`,`tau2`].

Algorithms

A similar model to the Nelson-Siegel is the Svensson model, which adds two additional parameters to account for greater flexibility in the term structure. This model proposes that the forward rate can be modeled with the following form:

$$f = \beta_0 + \beta_1 e^{\frac{-m}{\tau_1}} + \beta_2 e^{\frac{-m}{\tau_1}} \frac{m}{\tau_1} + \beta_3 e^{\frac{-m}{\tau_2}} \frac{m}{\tau_2}$$

As above, this can be integrated to derive an equation for the zero curve:

$$s = \beta_0 + \beta_1 (1 - e^{\frac{-m}{\tau_1}})(-\frac{\tau_1}{m}) + \beta_2((1 - e^{\frac{-m}{\tau_1}})\frac{\tau_1}{m} - e^{\frac{m}{\tau_1}}) + \beta_3((1 - e^{\frac{-m}{\tau_2}})\frac{\tau_2}{m} - e^{\frac{m}{\tau_2}})$$

Version History

Introduced in R2008b

References

[1] Nelson, C.R., Siegel, A.F. “Parsimonious modelling of yield curves.” *Journal of Business.* Vol. 60, 1987, pp 473–89.

[2] Svensson, L.E.O. *“Estimating and interpreting forward interest rates: Sweden 1992-4.”* International Monetary Fund, IMF Working Paper, 1994/114.

[3] Fisher, M., Nychka, D., Zervos, D. *“Fitting the term structure of interest rates with smoothing splines.”* Board of Governors of the Federal Reserve System, Federal Reserve Board Working Paper 1995-1.

[4] Anderson, N., Sleath, J. *“New estimates of the UK real and nominal yield curves.”* Bank of England Quarterly Bulletin, November, 1999, pp 384–92.

[5] Waggoner, D. *"Spline Methods for Extracting Interest Rate Curves from Coupon Bond Prices."* Federal Reserve Board Working Paper 1997–10.

[6] *"Zero-coupon yield curves: technical documentation."* BIS Papers No. 25, October 2005.

[7] Bolder, D.J., Gusba, S. *"Exponentials, Polynomials, and Fourier Series: More Yield Curve Modelling at the Bank of Canada."* Working Papers 2002–29, Bank of Canada.

[8] Bolder, D.J., Streliski, D. *"Yield Curve Modelling at the Bank of Canada."* Technical Reports 84, 1999, Bank of Canada.

See Also

`IRFunctionCurve` | `IRFitOptions` | `fitNelsonSiegel` | `fitSmoothingSpline` | `fitFunction`

Topics

"Fitting IRFunctionCurve Object Using Svensson Method" on page 9-17
"Fitting Interest-Rate Curve Functions" on page 9-24
"Interest-Rate Curve Objects and Workflow" on page 9-2
"Creating Interest-Rate Curve Objects" on page 9-4
"Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework" on page 1-95

External Websites

Calibration and Simulation Best Practices: Multifactor Interest Rate Models for Risk Applications (30 min 00 sec)

getDiscountFactors

Get discount factors for input dates for `IRDataCurve`

Syntax

```
F = getDiscountFactors(CurveObj,InpDates)
```

Description

`F = getDiscountFactors(CurveObj,InpDates)` computes discount factors for input dates for an `IRDataCurve` object.

Note The `ratecurve` object and the associated `discountfactors` were introduced in R2020a as part of a new object-based framework in the Financial Instruments Toolbox which supports end-to-end workflows in instrument modeling and analysis. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Get Discount Factors For Input Dates for an IRDataCurve

This example shows how to get discount factors for input dates for an `IRDataCurve`.

```
CurveSettle = datetime(2016,3,2);
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = datemnth(CurveSettle,12*[1 2 3 5 7 10 20 30]);
irdc = IRDataCurve('Zero',CurveSettle,Dates,Data);
getDiscountFactors(irdc, CurveSettle+30:30:CurveSettle+720)
```

```
ans = 24×1

    0.9986
    0.9971
    0.9956
    0.9940
    0.9924
    0.9907
    0.9890
    0.9873
    0.9855
    0.9836
      ⋮
```

Input Arguments

CurveObj — Interest-rate curve object

object

Interest-rate curve object, specified by using `IRDataCurve`.

Data Types: `object`

`InpDates` — Input dates
datetime array | string array | date character vector

Input dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The input dates must be after the `Settle` date of `IRDataCurve`.

To support existing code, `getDiscountFactors` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`F` — Discount factors
vector

Discount factors, returned as a vector.

Version History

Introduced in R2008b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `getDiscountFactors` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`IRDataCurve` | `getForwardRates` | `getZeroRates` | `getParYields` | `toRateSpec`

Topics
“Creating Interest-Rate Curve Objects” on page 9-4
“Creating an IRDataCurve Object” on page 9-6
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

getDiscountFactors

Get discount factors for input dates for `IRFunctionCurve`

Syntax

```
F = getDiscountFactors(CurveObj,InpDates)
```

Description

`F = getDiscountFactors(CurveObj,InpDates)` computes discount factors for input dates for an `IRFunctionCurve` object.

Note The `parametercurve` object and the associated `discountfactors` were introduced in R2020a as part of a new object-based framework in the Financial Instruments Toolbox which supports end-to-end workflows in instrument modeling and analysis. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Get Discount Factors for Input Dates For an IRFunctionCurve

This example shows how to get discount factors for input dates for an `IRFunctionCurve`.

```
irfc = IRFunctionCurve('Forward',today,@(t) polyval([-0.0001 0.003 0.02],t));
getDiscountFactors(irfc, today+30:30:today+720)
```

```
ans = 24×1

    0.9984
    0.9967
    0.9950
    0.9933
    0.9916
    0.9899
    0.9881
    0.9864
    0.9846
    0.9828
      ⋮
```

Input Arguments

CurveObj — Interest-rate curve object

object

Interest-rate curve object, specified by using `IRFunctionCurve`.

Data Types: `object`

`InpDates` — Input dates
datetime array | string array | date character vector

Input dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The input dates must be after the `Settle` date of `IRFunctionCurve`.

To support existing code, `getDiscountFactors` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`F` — Discount factors
vector

Discount factors, returned as a vector.

Version History

Introduced in R2008b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `getDiscountFactors` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`IRFunctionCurve` | `getForwardRates` | `getZeroRates` | `getParYields` | `toRateSpec`

Topics
“Creating an IRFunctionCurve Object” on page 9-16
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

getForwardRates

Get forward rates for input dates for `IRDataCurve`

Syntax

```
F = getForwardRates(CurveObj,InpDates)
F = getForwardRates( ___ ,Name,Value)
```

Description

`F = getForwardRates(CurveObj,InpDates)` computes discount factors for input dates for an `IRDataCurve` object. `getForwardRates` returns discrete forward rates for the intervals input into this function. For example, running the following code:

```
getForwardRates(irdc, {Date1, Date2, Date3})
```

gives three forwards rates and the three tenors are: `[Settle, Date1]`, `[Date1, Date2]`, and `[Date2, Date3]`.

Note The `ratecurve` object and the associated `forwardrates` were introduced in R2020a as part of a new object-based framework in the Financial Instruments Toolbox which supports end-to-end workflows in instrument modeling and analysis. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`F = getForwardRates( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Get Forward Rates For Input Dates for an IRDataCurve

This example shows how to get forward rates for input dates for an `IRDataCurve`.

```
CurveSettle = datetime(2016,3,2);
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = datemnth(CurveSettle,12*[1 2 3 5 7 10 20 30]);
irdc = IRDataCurve('Zero',CurveSettle,Dates,Data);
getForwardRates(irdc, CurveSettle+30:30:CurveSettle+720)
```

```
ans = 24×1

    0.0174
    0.0180
    0.0187
    0.0193
    0.0199
    0.0205
    0.0212
    0.0218
    0.0224
```

```
    0.0230
      ⋮
```

Use getForwardRates to Compute the Five Year Forward Rate in Five Years Time

Use `getForwardRates` to compute the forward rate from the `Settle` date to 5 years from March 1, 2017 and then the forward rate for the period from 5 years to 10 years from March 1, 2017.

```
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = daysadd(736755,[360 2*360 3*360 5*360 7*360 10*360 20*360 30*360],1);
irdc = IRDataCurve('Zero',today,Dates,Data);
getForwardRates(irdc,datemnth(irdc.Settle,12*[5 10]))
```

```
ans = 2×1

    0.0389
    0.0461
```

The first element (`.0312`) is the forward rate from the `Settle` to 5 years from March 1, 2017. The second rate (`0.0458`) is the forward rate for the period from 5 years to 10 years from March 1, 2017, in other words, the 5-year forward rate 5 years from March 1, 2017.

Compute the Six Month Forward Rates in 1 Month, 2 Months and 3 Months

Use the following data to create an `IRDataCurve` object:

```
Data = [0.1 0.30 0.70 1.05 1.45 1.71 2.12 2.43 2.85 3.57]/100;
Settle = datetime(2016,8,8)
```

```
Settle = datetime
   08-Aug-2016
```

```
Dates = datemnth(Settle,[3 6 9 12*[1 2 3 5 7 10 20]]);
irdc = IRDataCurve('Zero',Settle,Dates,Data)
```

```
irdc =
             Type: Zero
           Settle: 736550 (08-Aug-2016)
      Compounding: 2
            Basis: 0 (actual/actual)
     InterpMethod: linear
            Dates: [10x1 double]
             Data: [10x1 double]
```

Compute the implied 6 month forward rates in 1 month, 2 months, and 3 months from the `Settle` date.

```
IntervalMonth = 6; % Interval for 6 month forward rates
FwdMonths = [1 2 3]'; % Starting in 1, 2, and 3 months from Settle
```

```
N = length(FwdMonths);
FwdRates_6M = zeros(N,1);

for k = 1:N
    FwdDates = datemnth(irdc.Settle, [FwdMonths(k) FwdMonths(k)+IntervalMonth]);
    f = getForwardRates(irdc,FwdDates);
    FwdRates_6M(k) = f(2);
end

[FwdMonths FwdRates_6M]

ans = 3×2

    1.0000    0.0050
    2.0000    0.0074
    3.0000    0.0101
```

Input Arguments

`CurveObj` — Interest-rate curve object

object

Interest-rate curve object, specified by using `IRDataCurve`.

Data Types: `object`

`InpDates` — Input dates

datetime array | string array | date character vector

Input dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The input dates must be after the `Settle` date of `IRDataCurve`.

To support existing code, `getForwardRates` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `F = getForwardRates(irdc, CurveSettle+30:30:CurveSettle+720)`

`Compounding` — Compounding frequency per-year for forward rates

`CurveObj.Compounding` (default) | possible values include: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

Compounding frequency per-year for forward rates, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using one of the supported values:

- `–1` = Continuous compounding
- `0` = Simple interest (no compounding)

- 1 = Annual compounding
- 2 = Semiannual compounding
- 3 = Compounding three times per year
- 4 = Quarterly compounding
- 6 = Bimonthly compounding
- 12 = Monthly compounding

Data Types: double

Basis — Day count basis for the forward rates
0 (actual/actual) (default) | integer from 0 to 13

Day count basis for the forward rates, specified as the comma-separated pair consisting of 'Basis' and a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: double

Output Arguments

F — Forward rates
vector

Forward rates, returned as a vector. getForwardRates returns forward rates corresponding to the periodicity of the dates input to getForwardRates. For example, where the dates are monthly, monthly forward rates are returned. The first element of the output is the forward rate from the Settle to one month, the second element is the forward rate from one month to two months, and so on.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `getForwardRates` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`IRDataCurve` | `getZeroRates` | `getDiscountFactors` | `getParYields` | `toRateSpec`

Topics

“Creating Interest-Rate Curve Objects” on page 9-4
“Creating an IRDataCurve Object” on page 9-6
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

getForwardRates

Get forward rates for input dates for `IRFunctionCurve`

Syntax

```
F = getForwardRates(CurveObj,InpDates)
F = getForwardRates( ___ ,Name,Value)
```

Description

`F = getForwardRates(CurveObj,InpDates)` computes discount factors for input dates for an `IRFunctionCurve` object. `getForwardRates` returns discrete forward rates for the intervals input into this function. For example, running the following code:

```
getForwardRates(irdc, {Date1, Date2, Date3})
```

gives three forwards rates and the three tenors are: `[Settle, Date1]`, `[Date1, Date2]`, and `[Date2, Date3]`.

Note The `parametercurve` object and the associated `forwardrates` were introduced in R2020a as part of a new object-based framework in the Financial Instruments Toolbox which supports end-to-end workflows in instrument modeling and analysis. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`F = getForwardRates( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Get Forward Rates For Input Dates For an IRFunctionCurve

This example shows how to get forward rates for input dates for an `IRFunctionCurve`.

```
irfc = IRFunctionCurve('Forward',today,@(t) polyval([-0.0001 0.003 0.02],t));
getForwardRates(irfc, today+30:30:today+720)
```

```
ans = 24×1

    0.0202
    0.0205
    0.0207
    0.0210
    0.0212
    0.0215
    0.0217
    0.0219
    0.0222
    0.0224
      ⋮
```

Compute the Implied 2-Year Forward Rates in 1 Year, 2 Years, 5 Years, and 10 Years

This example shows how to compute the implied 2-year forward rates in 1 year, 2 years, 5 years, and 10 years from the `Settle` date by using the `getForwardRates` method.

Use the following data for an `IRFunctionCurve` object that is created when using the `fitSvensson` method.

```
Settle = datenum('15-Apr-2014');
Maturity = datemnth(Settle,12*[1 2 3 5 7 10 20 30]');

CleanPrice = [100.1 100.1 100.2 99.0 101.8 99.2 101.7 100.2]';
CouponRate = [0.0200 0.0275 0.035 0.042 0.0475 0.0525 0.055 0.052]';
Instruments = [repmat(Settle,size(Maturity)) Maturity CleanPrice CouponRate];

SvenssonModel = IRFunctionCurve.fitSvensson('Zero',Settle,Instruments);
```

Compute the implied 2-year forward rates in 1 year, 2 years, 5 years, and 10 years from the `Settle` date.

```
IntervalMonth = 12.*2;         % Interval months for 2-year forward rates
FwdMonths = 12.*[1 2 5 10]';   % Starting in 1, 2, 5, and 10 years from Settle
N = length(FwdMonths);
FwdRates_2Y = zeros(N,1);

for k = 1:N
    FwdDates = datemnth(SvenssonModel.Settle, [FwdMonths(k) FwdMonths(k)+IntervalMonth]);
    f = getForwardRates(SvenssonModel,FwdDates);
    FwdRates_2Y(k) = f(2);
end

[FwdMonths FwdRates_2Y]
```

```
ans = 4×2

   12.0000    0.0418
   24.0000    0.0504
   60.0000    0.0620
  120.0000    0.0629
```

Input Arguments

`CurveObj` — Interest-rate curve object

object

Interest-rate curve object, specified by using `IRFunctionCurve`.

Data Types: `object`

`InpDates` — Input dates

datetime array | string array | date character vector

Input dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The input dates must be after the `Settle` date of `IRFunctionCurve`.

To support existing code, `getForwardRates` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `F = getForwardRates(irfc, CurveSettle+30:30:CurveSettle+720)`

`Compounding` — Compounding frequency per-year for forward rates

`CurveObj.Compounding` (default) | possible values include: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

Compounding frequency per-year for forward rates, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using one of the supported values:

- `–1` = Continuous compounding
- `0` = Simple interest (no compounding)
- `1` = Annual compounding
- `2` = Semiannual compounding
- `3` = Compounding three times per year
- `4` = Quarterly compounding
- `6` = Bimonthly compounding
- `12` = Monthly compounding

Data Types: `double`

`Basis` — Day count basis for the forward rates

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis for the forward rates, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

F — Forward rates

vector

Forward rates, returned as a vector. `getForwardRates` returns forward rates corresponding to the periodicity of the dates input to `getForwardRates`. For example, where the dates are monthly, monthly forward rates are returned. The first element of the output is the forward rate from the `Settle` to one month, the second element is the forward rate from one month to two months, and so on.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `getForwardRates` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`IRFunctionCurve` | `getZeroRates` | `getDiscountFactors` | `getParYields`

Topics

“Creating an IRFunctionCurve Object” on page 9-16
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

getParYields

Get par yields for input dates for `IRDataCurve`

Syntax

```
F = getParYields(CurveObj,InpDates)
F = getParYields( ___ ,Name,Value)
```

Description

`F = getParYields(CurveObj,InpDates)` computes par yields for input dates for an `IRDataCurve` object.

`F = getParYields( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Get Par Yields For Input Dates For an `IRDataCurve`

This example shows how to get par yields for input dates for an `IRDataCurve`.

```
CurveSettle = datetime(2016,3,2);
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = datemnth(CurveSettle,12*[1 2 3 5 7 10 20 30]);
irdc = IRDataCurve('Zero',CurveSettle,Dates,Data);
getParYields(irdc, CurveSettle+30:30:CurveSettle+720)
```

```
ans = 24×1

    0.0175
    0.0177
    0.0181
    0.0183
    0.0186
    0.0189
    0.0194
    0.0197
    0.0200
    0.0203
      ⋮
```

Compute Par Yields From a Curve With Simple Interest Compounding

This example shows how set the compounding of an `IRDataCurve` to `Zero` (simple interest) and then compute par yields from that curve.

```
CurveSettle = datetime(2016,3,2);
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
```

```
Dates = datemnth(CurveSettle,12*[1 2 3 5 7 10 20 30]);
irdc = IRDataCurve('Zero',CurveSettle,Dates,Data,'Compounding',0);
SimpleInt = irdc.getParYields(Dates(1), 'Basis', 2, 'Compounding', 1)
```

```
SimpleInt = 0.0209
```

Input Arguments

`CurveObj` — Interest-rate curve object

object

Interest-rate curve object, specified by using `IRDataCurve`.

Data Types: `object`

`InpDates` — Input dates

datetime array | string array | date character vector

Input dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The input dates must be after the `Settle` date of `IRDataCurve`.

To support existing code, `getParYields` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `F = getParYields(irdc, CurveSettle+30:30:CurveSettle+720)`

`Compounding` — Compounding frequency per-year for par yield rates

`CurveObj.Compounding` (default) | possible values include: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

Compounding frequency per-year for par yield rates, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using one of the supported values:

- `–1` = Continuous compounding
- `0` = Simple interest (no compounding)
- `1` = Annual compounding
- `2` = Semiannual compounding
- `3` = Compounding three times per year
- `4` = Quarterly compounding
- `6` = Bimonthly compounding
- `12` = Monthly compounding

Data Types: `double`

`Basis` — Day count basis for the par yield rates

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis for the par yield rates, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

F — Par yields

vector

Par yields, returned as a vector.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `getParYields` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`IRDataCurve` | `getForwardRates` | `getZeroRates` | `getDiscountFactors` | `toRateSpec`

Topics

“Creating Interest-Rate Curve Objects” on page 9-4
“Creating an IRDataCurve Object” on page 9-6
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

getParYields

Get par yields for input dates for `IRFunctionCurve`

Syntax

```
F = getParYields(CurveObj,InpDates)
F = getParYields( ___ ,Name,Value)
```

Description

`F = getParYields(CurveObj,InpDates)` computes par yields for input dates for an `IRFunctionCurve` object.

`F = getParYields( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Get Par Yields For Input Dates For an `IRFunctionCurve`

This example shows how to get par yields for input dates for an `IRFunctionCurve`.

```
irfc = IRFunctionCurve('Forward',today,@(t) polyval([-0.0001 0.003 0.02],t));
getParYields(irfc, today+30:30:today+720)
```

```
ans = 24×1

    0.0203
    0.0204
    0.0206
    0.0206
    0.0208
    0.0209
    0.0207
    0.0210
    0.0210
    0.0212
      ⋮
```

Input Arguments

`CurveObj` — Interest-rate curve object

object

Interest-rate curve object, specified by using `IRFunctionCurve`.

Data Types: `object`

`InpDates` — Input dates

datetime array | string array | date character vector

Input dates, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors. The input dates must be after the Settle date of IRFunctionCurve.

To support existing code, getParYields also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: F = getParYields(irfc, CurveSettle+30:30:CurveSettle+720)

Compounding — Compounding frequency per-year for par yield rates

CurveObj.Compounding (default) | possible values include: –1, 0, 1, 2, 3, 4, 6, 12.

Compounding frequency per-year for par yield rates, specified as the comma-separated pair consisting of 'Compounding' and a scalar numeric using one of the supported values:

- –1 = Continuous compounding
- 0 = Simple interest (no compounding)
- 1 = Annual compounding
- 2 = Semiannual compounding
- 3 = Compounding three times per year
- 4 = Quarterly compounding
- 6 = Bimonthly compounding
- 12 = Monthly compounding

Data Types: double

Basis — Day count basis for the par yield rates

0 (actual/actual) (default) | integer from 0 to 13

Day count basis for the par yield rates, specified as the comma-separated pair consisting of 'Basis' and a scalar integer.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

F — Par yields

vector

Par yields, returned as a vector.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `getParYields` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`IRFunctionCurve` | `getForwardRates` | `getZeroRates` | `getDiscountFactors`

Topics

“Creating an IRFunctionCurve Object” on page 9-16
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

getZeroRates

Get zero rates for input dates for `IRDataCurve`

Syntax

```
F = getZeroRates(CurveObj,InpDates)
F = getZeroRates( ___ ,Name,Value)
```

Description

`F = getZeroRates(CurveObj,InpDates)` computes zero rates for input dates for an `IRDataCurve` object.

Note The `ratecurve` object and the associated `zerorates` were introduced in R2020a as part of a new object-based framework in the Financial Instruments Toolbox which supports end-to-end workflows in instrument modeling and analysis. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`F = getZeroRates( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Get Zero Rates For Input Dates For an IRDataCurve

This example shows how to get zero rates for input dates for an `IRDataCurve`.

```
CurveSettle = datetime(2016,3,2);
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = datemnth(CurveSettle,12*[1 2 3 5 7 10 20 30]);
irdc = IRDataCurve('Zero',CurveSettle,Dates,Data);
getZeroRates(irdc, CurveSettle+30:30:CurveSettle+720)
```

```
ans = 24×1

    0.0174
    0.0177
    0.0180
    0.0183
    0.0187
    0.0190
    0.0193
    0.0196
    0.0199
    0.0202
      ⋮
```

Input Arguments

`CurveObj` — Interest-rate curve object

object

Interest-rate curve object, specified by using `IRDataCurve`.

Data Types: `object`

`InpDates` — Input dates

datetime array | string array | date character vector

Input dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The input dates must be after the `Settle` date of `IRDataCurve`.

To support existing code, `getZeroRates` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `F = getZeroRates(irdc, CurveSettle+30:30:CurveSettle+720)`

`Compounding` — Compounding frequency per-year for zero rates

`CurveObj.Compounding` (default) | possible values include: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

Compounding frequency per-year for zero rates, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using one of the supported values:

- `–1` = Continuous compounding
- `0` = Simple interest (no compounding)
- `1` = Annual compounding
- `2` = Semiannual compounding
- `3` = Compounding three times per year
- `4` = Quarterly compounding
- `6` = Bimonthly compounding
- `12` = Monthly compounding

Data Types: `double`

`Basis` — Day count basis for the zero rates

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis for the zero rates, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

- 0 — actual/actual

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

F — Zero rates

vector

Zero rates, returned as a vector.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `getZeroRates` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`IRDataCurve` | `getForwardRates` | `getDiscountFactors` | `getParYields` | `toRateSpec`

Topics

“Creating Interest-Rate Curve Objects” on page 9-4
“Creating an IRDataCurve Object” on page 9-6
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

getZeroRates

Get zero rates for input dates for `IRFunctionCurve`

Syntax

```
F = getZeroRates(CurveObj,InpDates)
F = getZeroRates( ___ ,Name,Value)
```

Description

`F = getZeroRates(CurveObj,InpDates)` computes zero rates for input dates for an `IRFunctionCurve` object.

Note The `parametercurve` object and the associated `zerorates` were introduced in R2020a as part of a new object-based framework in the Financial Instruments Toolbox which supports end-to-end workflows in instrument modeling and analysis. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`F = getZeroRates( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Get Zero Rates For Input Dates For an `IRFunctionCurve`

This example shows how to get zero rates for input dates for an `IRFunctionCurve`.

```
irfc = IRFunctionCurve('Forward',today,@(t) polyval([-0.0001 0.003 0.02],t));
getZeroRates(irfc, today+30:30:today+720)
```

```
ans = 24×1

    0.0201
    0.0202
    0.0204
    0.0205
    0.0206
    0.0207
    0.0209
    0.0210
    0.0211
    0.0212
      ⋮
```

Input Arguments

`CurveObj` — Interest-rate curve object

object

Interest-rate curve object, specified by using `IRFunctionCurve`.

Data Types: `object`

`InpDates` — Input dates

datetime array | string array | date character vector

Input dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The input dates must be after the `Settle` date of `IRFunctionCurve`.

To support existing code, `getZeroRates` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `F = getZeroRates(irfc, CurveSettle+30:30:CurveSettle+720)`

`Compounding` — Compounding frequency per-year for zero rates

`CurveObj.Compounding` (default) | possible values include: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

Compounding frequency per-year for zero rates, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using one of the supported values:

- `–1` = Continuous compounding
- `0` = Simple interest (no compounding)
- `1` = Annual compounding
- `2` = Semiannual compounding
- `3` = Compounding three times per year
- `4` = Quarterly compounding
- `6` = Bimonthly compounding
- `12` = Monthly compounding

Data Types: `double`

`Basis` — Day count basis for the zero rates

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis for the zero rates, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

- 0 — actual/actual

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

F — Zero rates

vector

Zero rates, returned as a vector.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `getZeroRates` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`IRFunctionCurve` | `getForwardRates` | `getDiscountFactors` | `getParYields` | `toRateSpec`

Topics

“Creating an IRFunctionCurve Object” on page 9-16
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

IRBootstrapOptions

Construct specific options for bootstrapping interest-rate curve object

Description

Build an `IRBootstrapOptions` object using `IRBootstrapOptions`.

After creating an `IRBootstrapOptions` object, you can use the object with `bootstrap`.

For more detailed information on this workflow, see “Interest-Rate Curve Objects and Workflow” on page 9-2.

Creation

Syntax

```
IRBootstrapOptions_obj = IRBootstrapOptions(Name,Value)
```

Description

`IRBootstrapOptions_obj = IRBootstrapOptions(Name,Value)` sets properties on page 11-2089 and create the `IRBootstrapOptions` object to use with the `bootstrap` function. For example, `IRBootstrapOptions_obj = IRBootstrapOptions('LowerBound',-1)` creates an `IRBootstrapOptions` object. You can specify multiple name-value pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `IRBootstrapOptions_obj = IRBootstrapOptions('LowerBound',-1)`

`ConvexityAdjustment` — Controls the convexity adjustment to interest rate futures
`[]` (default) | function handle | numeric vector

Controls the convexity adjustment to interest rate futures, specified as the comma-separated pair consisting of `'ConvexityAdjustment'` and a function handle or an `N`-by-`1` numeric vector.

The function handle that takes one numeric input (time-to-maturity) and returns one numeric output, `ConvexityAdjustment`. For more information on defining a function handle, see the MATLAB Programming Fundamentals documentation.

Alternatively, you can define `ConvexityAdjustment` as an `N`-by-`1` vector of values, where `N` is the number of interest-rate futures.

In either case, the `ConvexityAdjustment` is subtracted from the futures rate.

Data Types: `double` | `function_handle`

`UpperBound` — Upper bound for rates associated with bonds or swaps
`1` (default) | scalar numeric | numeric vector

Upper bound for rates associated with bonds or swaps, specified as the comma-separated pair consisting of `'UpperBound'` and a scalar numeric or an `N`-by-`1` vector where `N` is the number of swaps and bonds. By default, `UpperBound` is `1`. Specify an upper bound that is greater than `1` when bootstrapping a discount curve.

Data Types: `double`

`LowerBound` — Lower bound for rates associated with bonds or swaps
`0` (default) | scalar numeric | numeric vector

Lower bound for rates associated with bonds or swaps, specified as the comma-separated pair consisting of `'LowerBound'` and a scalar numeric or an `N`-by-`1` vector where `N` is the number of swaps and bonds. By default, `LowerBound` is `0`.

Data Types: `double`

Properties

`ConvexityAdjustment` — Controls the convexity adjustment to interest rate futures
`[]` (default) | function handle | numeric vector

This property is read-only.

Controls the convexity adjustment to interest rate futures, returned as a function handle or an `N`-by-`1` numeric vector.

Data Types: `double` | `function_handle`

`UpperBound` — Upper bound for rates associated with bonds or swaps
`1` (default) | scalar numeric | numeric vector

Upper bound for rates associated with bonds or swaps, returned as a scalar numeric or an `N`-by-`1` vector.

Data Types: `double`

`LowerBound` — Lower bound for rates associated with bonds or swaps
`0` (default) | scalar numeric | numeric vector

Lower bound for rates associated with bonds or swaps, returned as a scalar numeric or an `N`-by-`1` vector.

Data Types: `double`

Object Functions

bootstrap Bootstrap interest-rate curve from market data

Examples

Create IRBootstrapOptionsObj to Use With the bootstrap Method

Set the `ConvexityAdjustment` to control interest-rate futures.

```
mybootoptions = IRBootstrapOptions('ConvexityAdjustment',repmat(.005,10,1))
```

```
mybootoptions =
  IRBootstrapOptions with properties:

    ConvexityAdjustment: [10x1 double]
             LowerBound: 0
             UpperBound: 1
```

Use `mybootoptions` as the optional argument, `IRBootstrapOptionsObj`, to use with the `bootstrap` method.

Create an IRBootstrapOptionsObj to Use With Negative Zero Interest-Rates

Use an `IRBootstrapOptionsObj` optional argument with the `bootstrap` method to allow for negative zero rates when solving the swap zero points.

```
Settle = datenum('15-Mar-2015');
InstrumentTypes = {'Deposit';'Deposit';'Swap';'Swap';'Swap';'Swap';};

Instruments = [Settle,datenum('15-Jun-2015'),.001; ...
Settle,datenum('15-Dec-2015'),.0005; ...
Settle,datenum('15-Mar-2016'),-.001; ...
Settle,datenum('15-Mar-2017'),-0.0005; ...
Settle,datenum('15-Mar-2018'),.0017; ...
Settle,datenum('15-Mar-2020'),.0019];

irbo = IRBootstrapOptions('LowerBound',-1);

bootModel = IRDataCurve.bootstrap('zero', Settle, InstrumentTypes,...
    Instruments,'IRBootstrapOptions',irbo);

bootModel.getZeroRates(datemnth(Settle,1:60))
```

```
ans = 60×1

    0.0012
    0.0011
    0.0010
    0.0009
    0.0008
    0.0008
    0.0007
    0.0006
    0.0005
   -0.0000
      ⋮
```

Note that `IRBootstrapOptions` optional argument for `LowerBound` is set to `-1` for negative zero rates when solving the swap zero points.

Version History

Introduced in R2008b

See Also

`IRDataCurve` | `ratecurve` | `irbootstrap`

Topics

“Creating Interest-Rate Curve Objects” on page 9-4
“Creating an IRDataCurve Object” on page 9-6
“Bootstrap IRDataCurve Based on Market Instruments” on page 9-7
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

IRDataCurve

Construct interest-rate curve object from dates and data

Description

Build an `IRDataCurve` object using `IRDataCurve`.

After creating an `IRDataCurve` object, you can use the associated object functions:

Object Function	Description
`getForwardRates`	Returns forward rates for input dates.
`getZeroRates`	Returns zero rates for input dates.
`getDiscountFactors`	Returns discount factors for input dates.
`getParYields`	Returns par yields for input dates.
`toRateSpec`	Converts to be a `RateSpec` object; this structure is identical to the `RateSpec` produced by the function `intenvset`.
`bootstrap`	Bootstraps an interest rate curve from market data.

For more detailed information on this workflow, see “Interest-Rate Curve Objects and Workflow” on page 9-2.

Creation

Syntax

```
IRDataCurve_obj = IRDataCurve(Type,Settle,Dates,Data)
IRDataCurve_obj = IRDataCurve( ___ ,Name,Value)
```

Description

`IRDataCurve_obj = IRDataCurve(Type,Settle,Dates,Data)` sets properties on page 11-2094 and creates an `IRDataCurve` object.

Note The `ratecurve` object was introduced in R2020a as part of a new object-based framework in the Financial Instruments Toolbox which supports end-to-end workflows in instrument modeling and analysis. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22. While `IRDataCurve` can be used for building and analyzing yield curves, `ratecurve` provides that functionality, but it is also seamlessly integrated with instrument pricing functionality. For example, the `ratecurve` can be used to price instruments (`Discount`) and the `irbootstrap` function takes as inputs an array of instrument objects.

`IRDataCurve_obj = IRDataCurve( ___ ,Name,Value)` sets optional properties on page 11-2094 using name-value pairs and any of the arguments in the previous syntax. For example,

`IRDataCurve_obj = IRDataCurve('Zero',CurveSettle,Dates,Data,'Compounding',4,'Basis',4)` creates an `IRDataCurve` object for a zero curve. You can specify multiple name-value pair arguments.

Input Arguments

Type — Type of interest-rate curve

string with value `"zero"`, `"forward"`, or `"discount"` | character vector with value `'zero'`, `'forward'`, or `'discount'`

Type of interest-rate curve, specified as a scalar string or character vector for one of the supported types.

Data Types: `char` | `string`

Settle — Settlement date for the curve

datetime scalar | string scalar | date character vector

Settlement date for the curve, specified as a scalar datetime, string, or date character vector.

To support existing code, `IRDataCurve` also accepts serial date numbers as inputs, but they are not recommended.

Dates — Dates corresponding to rate data

datetime array | string array | date character vector

Dates corresponding to the rate data, specified as datetime array, string array, or a vector of date character vectors.

To support existing code, `IRDataCurve` also accepts serial date numbers as inputs, but they are not recommended.

Data — Interest-rate data for curve object

numeric vector

Interest-rate data for curve object, specified as a numeric vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `IRDataCurve_obj = IRDataCurve('Zero',CurveSettle,Dates,Data,'Compounding',4,'Basis',4)`

Compounding — Compounding frequency for curve

`-1` (default) | possible values include: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

Compounding frequency for the curve, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using the supported values: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Note Simple interest can be specified by setting the `Compounding` value as `0` and is supported for "zero" and "discount" curve types only (not supported for "forward" curves).

Data Types: `double`

`Basis` — Day count basis of interest-rate curve
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of interest-rate curve, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

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

For more information, see "Basis" on page 2-228.

Data Types: `double`

`InterpMethod` — Interpolation method
`'linear'` (default) | character vector

Interpolation method, specified as the comma-separated pair consisting of `'InterpMethod'` and a character vector or string for one of the following values:

- `'linear'` — Linear interpolation (default)
- `'constant'` — Piecewise constant interpolation.
- `'pchip'` — Piecewise cubic Hermite interpolation.
- `'spline'` — Cubic spline interpolation

Data Types: `char` | `string`

Properties

Type — Type of interest-rate curve
string with value `"zero"`, `"forward"`, or `"discount"`

This property is read-only.

Instrument type, returned as a string.

Data Types: `string`

`Settle` — Settlement date

datetime

This property is read-only.

Settlement date, returned as a datetime.

Data Types: `datetime`

`Dates` — Dates corresponding to rate data

numeric vector

This property is read-only.

Dates corresponding to rate data, returned as a numeric vector.

Data Types: `double`

`Data` — Interest-rate data for the curve object

numeric

This property is read-only.

Interest-rate data for the curve object, returned as a numeric vector.

Data Types: `double`

`Compounding` — Compounding frequency for curve

`-1` (default) | possible values include: –1, `0`, 1, 2, 3, 4, 6, 12.

This property is read-only.

Compounding frequency for curve, returned as a scalar numeric.

Data Types: `double`

`Basis` — Day count basis

`0` (actual/actual) (default) | integer from `0` to `13`

This property is read-only.

Day count basis, returned as a scalar integer.

Data Types: `double`

`InterpMethod` — Interpolation method

`'linear'` (default) | character vector

This property is read-only.

Interpolation method, returned as a scalar character vector.

Data Types: `char`

Object Functions

getForwardRates	Get forward rates for input dates for IRDataCurve
getZeroRates	Get zero rates for input dates for IRDataCurve
getDiscountFactors	Get discount factors for input dates for IRDataCurve
getParYields	Get par yields for input dates for IRDataCurve
toRateSpec	Convert IRDataCurve object to RateSpec
bootstrap	Bootstrap interest-rate curve from market data

Examples

Create IRDataCurve Object

This example shows how to create an `IRDataCurve` object for an interest-rate curve.

Define the type of interest-rate curve, `Settle` date, `Dates`, and `Data`.

```
CurveSettle = datetime(2016,3,2);
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = datemnth(CurveSettle,12*[1 2 3 5 7 10 20 30]);
```

Use `IRDataCurve` to create an `IRDataCurve` object.

```
irdc = IRDataCurve('Zero',CurveSettle,Dates,Data,'Compounding',4,'Basis',4)
```

```
irdc =
             Type: Zero
           Settle: 736391 (02-Mar-2016)
      Compounding: 4
            Basis: 4 (30/360 (PSA))
     InterpMethod: linear
            Dates: [8x1 double]
             Data: [8x1 double]
```

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `IRDataCurve` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`ratecurve`

Topics
“Creating Interest-Rate Curve Objects” on page 9-4
“Creating an IRDataCurve Object” on page 9-6
“Creating an IRFunctionCurve Object” on page 9-16
“Convert RateSpec to a ratecurve Object” on page 1-49
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

IRFitOptions

Construct specific options for fitting interest-rate curve object

Description

Build an `IRFitOptions` object using `IRFitOptions`.

After creating an `IRFitOptions` object, you can use the object with `fitFunction`.

For more detailed information on this workflow, see “Interest-Rate Curve Objects and Workflow” on page 9-2.

Creation

Syntax

```
IRFitOptions_obj = IRFitOptions(InitialGuess)
IRFitOptions_obj = IRFitOptions( ___ ,Name,Value)
```

Description

`IRFitOptions_obj = IRFitOptions(InitialGuess)` sets properties on page 11-2099 and create the `IRFitOptions` object with an initial guess or with an initial guess and bounds.

`IRFitOptions_obj = IRFitOptions( ___ ,Name,Value)` sets optional properties on page 11-2099 using name-value pairs and any of the arguments in the previous syntax. For example, `IRFitOptions_obj = IRFitOptions([7 2 1 0],'FitType','yield')` creates an `IRFitOptions` object to use with `fitFunction` when building a custom fitting function. You can specify multiple name-value pair arguments.

Input Arguments

`InitialGuess` — Initial guess for the parameters of the curve function
vector

Initial guess for the parameters of the curve function, specified as vector of values for the starting point of the optimization.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `IRFitOptions_obj = IRFitOptions([7 2 1 0],'FitType','yield')`

`FitType` — Minimize by in curve fitting process
`'DurationWeightedPrice'` (default) | character vector with value of `'DurationWeightedPrice'`, `'Price'`, or `'Yield'`

Minimize by in curve fitting process, specified as the comma-separated pair consisting of `'FitType'` and a character vector.

Data Types: `char`

`UpperBound` — Upper bound for parameters of curve function
`[]` (default) | numeric

Upper bound for parameters of the curve function, specified as the comma-separated pair consisting of `'UpperBound'` and a scalar numeric.

Data Types: `double`

`LowerBound` — Lower bound for parameters of curve function
`[]` (default) | numeric

Lower bound for parameters of the curve function, specified as the comma-separated pair consisting of `'LowerBound'` and a scalar numeric.

Data Types: `double`

`OptOptions` — Optimization parameters
`[]` (default) | structure

Optimization parameters, specified as the comma-separated pair consisting of `'OptOptions'` and a structure defined by using `optimoptions` (`optimset` is also supported).

Data Types: `struct`

Properties

`FitType` — Minimize by in curve fitting process
`'DurationWeightedPrice'` (default) | character vector with value of `'DurationWeightedPrice'`, `'Price'`, or `'Yield'`

This property is read-only.

Minimize by in curve fitting process, returned as a character vector.

Data Types: `char`

`InitialGuess` — Initial guess for the parameters of the curve function
vector

This property is read-only.

Initial guess for the parameters of the curve function, returned as a vector.

Data Types: `double`

`UpperBound` — Upper bound for parameters of curve function
`[]` (default) | numeric

This property is read-only.

Upper bound for parameters of the curve function, returned as a scalar numeric.

Data Types: `double`

`LowerBound` — Lower bound for parameters of curve function
`[]` (default) | numeric

This property is read-only.

Lower bound for parameters of the curve function, returned as a scalar numeric.

Data Types: `double`

`OptOptions` — Optimization parameters
`[]` (default) | structure

This property is read-only.

Optimization parameters, returned as a structure defined by using `optimoptions` (`optimset` is also supported).

Data Types: `struct`

`A` — Inequality constraint for parameters
`[]` (default) | numeric

This property is read-only.

Inequality constraint for parameters, returned as a scalar numeric.

Data Types: `double`

`b` — Inequality constraint for parameters
`[]` (default) | numeric

This property is read-only.

Inequality constraint for parameters, returned as a scalar numeric.

Data Types: `double`

`OptimFunction` — Optimization function used to fit function
`lsqnonlin` (default) | `lsqnonlin` or `fmincon`

This property is read-only.

Optimization function used to fit function, returned as `lsqnonlin` or `fmincon`.

Data Types: `char`

Object Functions

fitFunction Custom fit interest-rate curve object to bond market data

Examples

Create IRFitOptions Object

This example shows how to create an `IRFitOptions` object with a `'yield'` `FitType`.

```
myfitoptions = IRFitOptions([7 2 1 0],'FitType','yield')
```

```
myfitoptions =
  IRFitOptions with properties:

          FitType: 'yield'
     InitialGuess: [7 2 1 0]
       UpperBound: []
       LowerBound: []
       OptOptions: []
                A: []
                b: []
    OptimFunction: 'lsqnonlin'
```

Version History

Introduced in R2008b

See Also

`IRFunctionCurve` | `parametercurve`

Topics

“Creating an IRFunctionCurve Object” on page 9-16
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

IRFunctionCurve

Construct interest-rate curve object from function handle or function and fit to market data

Description

Build a `IRFunctionCurve` object using `IRFunctionCurve`.

After you create an `IRFunctionCurve` object, you can fit the bond using the following functions.

Object Function	Description
`getForwardRates`	Returns forward rates for input dates.
`getZeroRates`	Returns zero rates for input dates.
`getDiscountFactors`	Returns discount factors for input dates.
`getParYields`	Returns par yields for input dates.
`toRateSpec`	Converts to be a `RateSpec` object. This `RateSpec` structure is identical to the `RateSpec` produced by the function `intenvset`.

Alternatively, you can create an `IRFunctionCurve` object using the following methods.

Method	Description
`fitNelsonSiegel`	Fits a Nelson-Siegel function to market data.
`fitSvensson`	Fits a Svensson function to market data.
`fitSmoothingSpline`	Fits a smoothing spline function to market data.
`fitFunction`	Fits a custom function to market data.

For more detailed information on this workflow, see “Interest-Rate Curve Objects and Workflow” on page 9-2.

Creation

Syntax

```
IRFunctionCurve_obj = IRFunctionCurve(Type,Settle,FunctionHandle)
IRFunctionCurve_obj = IRFunctionCurve( ___ ,Name,Value)
```

Description

`IRFunctionCurve_obj = IRFunctionCurve(Type,Settle,FunctionHandle)` creates an interest-rate curve object directly by specifying a function handle and sets properties on page 11-2104 and creates an `IRFunctionCurve` object. .

`IRFunctionCurve_obj = IRFunctionCurve( ___ ,Name,Value)` sets properties on page 11-2104 using optional name-value pairs and any of the arguments in the previous syntax. For example,

`IRFunctionCurve_obj = IRFunctionCurve('Forward',today,@(t) polyval([-0.0001 0.003 0.02],t))` creates an `IRFunctionCurve` object for a forward curve. You can specify multiple name-value pair arguments.

Input Arguments

Type — Type of interest-rate curve

string with value `"zero"`, `"forward"`, or `"discount"` | character vector with value `'zero'`, `'forward'`, or `'discount'`

Type of interest-rate curve, specified as a scalar string or character vector for one of the supported types.

Data Types: `char` | `string`

Settle — Settlement date for the curve

datetime scalar | string scalar | date character vector

Settlement date for the curve, specified as a scalar datetime, string, or date character vector.

To support existing code, `IRFunctionCurve` also accepts serial date numbers as inputs, but they are not recommended.

FunctionHandle — Dates corresponding to rate data

function handle

Dates corresponding to the rate data, specified as a function handle. The function handle requires one numeric input (time-to-maturity) and returns one numeric output (interest rate or discount factor). For more information on creating a function handle, see “Create Function Handle”.

Data Types: `function_handle`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `IRFunctionCurve_obj = IRFunctionCurve('Forward',today,@(t) polyval([-0.0001 0.003 0.02],t))`

Compounding — Compounding frequency per-year for curve

`-1` (default) | possible values include: –1, `0`, 1, 2, 3, 4, 6, 12.

Compounding frequency per-year for the curve, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using the supported values: –1, `0`, 1, 2, 3, 4, 6, or 12.

Data Types: `double`

Basis — Day count basis of the bond

`0` (actual/actual) (default) | integer from `0` to 13

Day count basis of the bond, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Parameters` — Curve parameters
`[ ]` (default) | numeric

Curve parameters, specified as the comma-separated pair consisting of `'Parameters'` and a numeric value.

Data Types: `double`

Properties

Type — Type of interest-rate curve
string with value `"zero"`, `"forward"`, or `"discount"`

This property is read-only.

Instrument type, returned as a string.

Data Types: `string`

`Settle` — Settlement date for the curve
datetime

This property is read-only.

Settlement date for the curve, returned as a datetime.

Data Types: `datetime`

`FunctionHandle` — Dates corresponding to rate data
function handle

This property is read-only.

Function handle that defines the interest-rate curve, returned as a scalar function handle.

Data Types: `function_handle`

`Compounding` — Compounding frequency per-year for curve
`-1` (default) | possible values include: –1, `0`, 1, 2, 3, 4, 6, 12.

This property is read-only.

Compounding frequency per-year for curve, returned as a scalar numeric.

Data Types: `double`

`Basis` — Day count basis of the bond
`0` (actual/actual) (default) | integer from `0` to `13`

This property is read-only.

Day count basis of the bond, returned as a scalar integer.

Data Types: `double`

`Parameters` — Curve parameters
`[ ]` (default) | numeric

This property is read-only.

Curve parameters, returned as a numeric value.

Data Types: `double`

Object Functions

getForwardRates	Get forward rates for input dates for IRFunctionCurve
getZeroRates	Get zero rates for input dates for IRFunctionCurve
getDiscountFactors	Get discount factors for input dates for IRFunctionCurve
getParYields	Get par yields for input dates for IRFunctionCurve
toRateSpec	Convert IRFunctionCurve object to RateSpec

Examples

Create IRFunctionCurve Object

This example shows how to create an `IRFunctionCurve` object for a forward interest-rate curve.

```
irfc = IRFunctionCurve('Forward',today,@(t) polyval([-0.0001 0.003 0.02],t))
```

```
irfc =
            Type: Forward
          Settle: 738764 (31-Aug-2022)
    Compounding: 2
           Basis: 0 (actual/actual)
```

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `IRFunctionCurve` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`parametercurve`

Topics

“Creating Interest-Rate Curve Objects” on page 9-4
“Creating an IRDataCurve Object” on page 9-6
“Creating an IRFunctionCurve Object” on page 9-16
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

liborduration

Duration of LIBOR-based interest-rate swap

Syntax

```
[PayFixDuration,GetFixDuration] = liborduration(SwapFixRate,Tenor,Settle)
```

Description

`[PayFixDuration,GetFixDuration] = liborduration(SwapFixRate,Tenor,Settle)` computes the duration of LIBOR-based interest-rate swaps.

Examples

Compute the Duration of LIBOR-Based Interest-Rate Swaps

This example shows how to compute the duration of LIBOR-based interest-rate swaps using the following data.

```
SwapFixRate = 0.0383;
Tenor = 7;
Settle = datetime(2002,10,11);

[PayFixDuration GetFixDuration] = liborduration(SwapFixRate,...
Tenor, Settle)
```

```
PayFixDuration = -4.7567

GetFixDuration = 4.7567
```

Input Arguments

`SwapFixRate` — Par swap fixed rate

vector in decimals

Par swap fixed rate (quarterly compounded), specified as an `N`-by-`1` vector in decimals. The Basis should be actual/360.

Data Types: `double`

`Tenor` — Swap tenor in years

vector

Swap tenor in years, specified as a `N`-by-`1` vector. Fractional numbers are rounded upward.

Data Types: `double`

`Settle` — Settlement date

vector

Settlement date, specified as an N-by-1 vector using serial date numbers.

Data Types: double

Output Arguments

PayFixDuration — Modified duration, in years, for the pay-fix side of the swap
vector

Modified duration, in years, for the pay-fix side of the swap, returned as a N-by-1 vector.

GetFixDuration — Modified duration, in years, for the receive-fix side of the swap
vector

Modified duration, in years, for the receive-fix side of the swap, returned as a N-by-1 vector.

Version History

Introduced before R2006a

See Also

liborfloat2fixed | liborprice

Topics

“Analysis of Bond Futures” on page 7-12
“Fitting the Diebold Li Model” on page 2-150
“Managing Present Value with Bond Futures” on page 7-14

liborfloat2fixed

Compute par fixed-rate of swap given 3-month LIBOR data

Syntax

```
[FixedSpec,ForwardDates,ForwardRates] = liborfloat2fixed(ThreeMonthRates,
Settle,Tenor)
Price = liborprice( ___ ,StartDate,Interpolation,ConvexAdj,RateParam,InArrears,
Sigma,FixedCompound,FixedBasis)
```

Description

`[FixedSpec,ForwardDates,ForwardRates] = liborfloat2fixed(ThreeMonthRates, Settle,Tenor)` computes forward rates, dates, and the swap fixed rate.

Note The `liborfloat2fixed` function assumes that floating-rate observations occur quarterly on the third Wednesday of a delivery month. The first delivery month is the month of the first third Wednesday after `Settle`. Floating-side payments occur on the third-month anniversaries of observation dates. Fixed payments start on the same date as the first floating payment, and recur on the same date as the first-coupon date (on anniversary months).

`Price = liborprice( ___ ,StartDate,Interpolation,ConvexAdj,RateParam,InArrears, Sigma,FixedCompound,FixedBasis)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Par Fixed-Rate of a Swap Given 3-Month LIBOR Data

This example shows how to compute the par fixed-rate of a swap given 3-month LIBOR data. Use the supplied `EDdata.xls` file as input to a `liborfloat2fixed` computation.

```
[EDFutData, textdata] = xlsread('EDdata.xls');
Settle                = datetime(2002,10,15);
Tenor                 = 2;

[FixedSpec, ForwardDates, ForwardRates] =...
liborfloat2fixed(EDFutData(:,1:3), Settle, Tenor)
```

```
FixedSpec = struct with fields:
      Coupon: 0.0222
      Settle: '16-Oct-2002'
    Maturity: '16-Oct-2004'
      Period: 4
       Basis: 1


ForwardDates = 8×1
```

```
      731505
      731596
      731687
      731778
      731869
      731967
      732058
      732149


ForwardRates = 8×1

    0.0177
    0.0166
    0.0170
    0.0188
    0.0214
    0.0248
    0.0279
    0.0305
```

Input Arguments

`ThreeMonthRates` — Three-month Eurodollar futures data or forward rate agreement data

matrix

Three-month Eurodollar futures data or forward rate agreement data, specified as an `N`-by-3 matrix in the form of `[month year IMMQuote]`. A forward rate agreement stipulates that a certain interest rate applies to a certain principal amount for a given future time period. The floating rate is assumed to compound quarterly and to accrue on an actual/360 basis.

Data Types: `double`

`Settle` — Settlement date of fixed-rate of swap

datetime scalar | string scalar | date character vector

Settlement date of fixed-rate of swap, specified as a scalar datetime, string, or date character vector.

To support existing code, `liborfloat2fixed` also accepts serial date numbers as inputs, but they are not recommended.

`Tenor` — Life of the swap contract

scalar integer

Life of the swap contract, specified as a scalar integer.

Data Types: `double`

`StartDate` — Reference date for valuation of forward swap

`Settle` (default) | datetime scalar | string scalar | date character vector

(Optional) Reference date for valuation of forward swap, specified as a scalar datetime, string, or date character vector. This in effect allows forward swap valuation.

To support existing code, `liborfloat2fixed` also accepts serial date numbers as inputs, but they are not recommended.

`Interpolation` — Interpolation method to determine applicable forward rate for months when no Eurodollar data is available
`'linear'` (1) (default) | scalar integer with value of `0`, `1`, or `2`

(Optional) Interpolation method to determine applicable forward rate for months when no Eurodollar data is available, specified as a scalar numeric with values of:

- `0` is `'nearest'`
- `1` is `'linear'`
- `2` is `'cubic'`

.

Data Types: `double`

`ConvexAdj` — Indicates whether futures/forward convexity adjustment is required
`0` (off) (default) | scalar logical with a value of `0` or `1`

(Optional) Indicates whether futures/forward convexity adjustment is required, specified as a scalar logical. Use `ConvexAdj` for forward rate adjustments when those rates are taken from Eurodollar futures data.

Data Types: `logical`

`RateParam` — Short-rate model's parameters (Hull-White)
`[0.05 0.015]` (default) | vector

(Optional) Short-rate model's parameters (Hull-White), specified a 1-by-2 vector to denote the parameters `[a S]`, where the short-rate process is:
$dr = [\theta(t) - ar]dt + Sdz$.

Data Types: `double`

`InArrears` — Indicates whether the swap is in arrears
`0` (off) (default) | scalar logical with a value of `0` or `1`

(Optional) Indicates whether the swap is in arrears, specified as a scalar logical.

Data Types: `logical`

`Sigma` — Overall annual volatility of caplets
scalar numeric

(Optional) Overall annual volatility of caplets, specified as a scalar numeric.

Data Types: `double`

`FixedCompound` — Compounding or frequency of payment on the fixed side
4 (quarterly) (default) | scalar numeric with possible values of 1, 2,4, or 12

(Optional) Compounding or frequency of payment on the fixed side, specified as a scalar numeric with one of the following possible values:

- 1 is annual
- 2 is semiannual
- 4 is quarterly
- 12 is monthly

Data Types: double

FixedBasis — Basis of the fixed side
0 (actual/actual) (default) | scalar numeric

(Optional) Basis of the fixed side, specified as a scalar numeric using one of the 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/365 (ISDA)

For more information, see “Basis” on page 2-228.

Data Types: double

Output Arguments

FixedSpec — Structure of the fixed-rate side of the swap
structure

Structure of the fixed-rate side of the swap, returned as a structure with the following fields:

- Coupon: Par-swap rate
- Settle: Start date
- Maturity: End date
- Period: Frequency of payment
- Basis: Accrual basis

ForwardDates — Dates corresponding to ForwardRates
numeric

Dates corresponding to `ForwardRates`, returned as datetime numbers. All of the dates are third Wednesdays of the month, spread three months apart. The first element is the third Wednesday immediately after `Settle`.

`ForwardRates` — Forward rates corresponding to the forward dates
numeric

Forward rates corresponding to the forward dates, quarterly compounded, and on the actual/360 basis, returned as numeric decimal values.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `liborfloat2fixed` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`liborduration` | `liborprice`

Topics
“Analysis of Bond Futures” on page 7-12
“Fitting the Diebold Li Model” on page 2-150
“Managing Present Value with Bond Futures” on page 7-14

liborprice

Price swap given swap rate

Syntax

```
Price = liborprice(ThreeMonthRates,Settle,Tenor,SwapRate)
Price = liborprice( ___ ,StartDate,Interpolation,ConvexAdj,RateParam,InArrears,
Sigma,FixedCompound,FixedBasis)
```

Description

`Price = liborprice(ThreeMonthRates,Settle,Tenor,SwapRate)` computes the price per $100 notional value of a swap given the swap rate. A positive result indicates that fixed side is more valuable than the floating side.

`Price = liborprice( ___ ,StartDate,Interpolation,ConvexAdj,RateParam,InArrears, Sigma,FixedCompound,FixedBasis)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Price Per $100 Notional Value of a Swap Given the Swap Rate

This example shows that a swap paying the par swap rate has a value of `0`.

```
% load the input data
[EDFutData, textdata] = xlsread('EDdata.xls');
Settle = datetime(2002,10,15);
Tenor = 2;

% compute the fixed rate from the Eurodollar data
FixedSpec = liborfloat2fixed(EDFutData(:,1:3), Settle, Tenor)
```

```
FixedSpec = struct with fields:
      Coupon: 0.0222
      Settle: '16-Oct-2002'
    Maturity: '16-Oct-2004'
      Period: 4
       Basis: 1
```

```
% compute the price of a par swap
Price = liborprice(EDFutData(:,1:3), Settle, Tenor, FixedSpec.Coupon)
```

```
Price = 3.4694e-15
```

`Price` is effectively equal to 0.

Input Arguments

`ThreeMonthRates` — Three-month Eurodollar futures data or forward rate agreement data

matrix

Three-month Eurodollar futures data or forward rate agreement data, specified as an `N`-by-`3` matrix in the form of `[month year IMMQuote]`. A forward rate agreement stipulates that a certain interest rate applies to a certain principal amount for a given future time period. The floating rate is assumed to compound quarterly and to accrue on an actual/360 basis.

Data Types: `double`

`Settle` — Settlement date of fixed-rate of swap

datetime scalar | string scalar | date character vector

Settlement date of fixed-rate of swap, specified as a scalar datetime, string, or date character vector.

To support existing code, `liborprice` also accepts serial date numbers as inputs, but they are not recommended.

`Tenor` — Life of the swap contract

scalar integer

Life of the swap contract, specified as a scalar integer.

Data Types: `double`

`SwapRate` — Swap rate

scalar decimal

Swap rate, specified as a scalar decimal.

Data Types: `double`

`StartDate` — Reference date for valuation of forward swap

`Settle` (default) | datetime scalar | string scalar | date character vector

(Optional) Reference date for valuation of forward swap, specified as a scalar datetime, string, or date character vector. This in effect allows forward swap valuation.

To support existing code, `liborprice` also accepts serial date numbers as inputs, but they are not recommended.

`Interpolation` — Interpolation method to determine applicable forward rate for months when no Eurodollar data is available

`'linear'` (`1`) (default) | scalar integer with value of `0`, `1`, or `2`

(Optional) Interpolation method to determine applicable forward rate for months when no Eurodollar data is available, specified as a scalar numeric with values of:

- `0` is `'nearest'`
- `1` is `'linear'`
- `2` is `'cubic'`

.

Data Types: `double`

`ConvexAdj` — Indicates whether futures/forward convexity adjustment is required
`0` (off) (default) | scalar logical with a value of `0` or `1`

(Optional) Indicates whether futures/forward convexity adjustment is required, specified as a scalar logical. Use `ConvexAdj` for forward rate adjustments when those rates are taken from Eurodollar futures data.

Data Types: `logical`

`RateParam` — Short-rate model's parameters (Hull-White)
`[0.05 0.015]` (default) | vector

(Optional) Short-rate model's parameters (Hull-White), specified a 1-by-2 vector to denote the parameters `[a S]`, where the short-rate process is:
$dr = [\theta(t) - ar]dt + Sdz$.

Data Types: `double`

`InArrears` — Indicates whether the swap is in arrears
`0` (off) (default) | scalar logical with a value of `0` or `1`

(Optional) Indicates whether the swap is in arrears, specified as a scalar logical.

Data Types: `logical`

`Sigma` — Overall annual volatility of caplets
scalar numeric

(Optional) Overall annual volatility of caplets, specified as a scalar numeric.

Data Types: `double`

`FixedCompound` — Compounding or frequency of payment on the fixed side
4 (quarterly) (default) | scalar numeric with possible values of 1, 2,4, or 12

(Optional) Compounding or frequency of payment on the fixed side, specified as a scalar numeric with one of the following possible values:

- 1 is annual
- 2 is semiannual
- 4 is quarterly
- 12 is monthly

Data Types: `double`

`FixedBasis` — Basis of the fixed side
`0` (actual/actual) (default) | scalar numeric

(Optional) Basis of the fixed side, specified as a scalar numeric using one of the 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/365 (ISDA)

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

`Price` — Present value of the difference between floating and fixed-rate sides of the swap per $100 notional

numeric

Present value of the difference between floating and fixed-rate sides of the swap per $100 notional, returned as a numeric value.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `liborprice` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`liborduration` | `liborfloat2fixed`

Topics

“Analysis of Bond Futures” on page 7-12

"Fitting the Diebold Li Model" on page 2-150
"Managing Present Value with Bond Futures" on page 7-14

mbscfamounts

Cash flow and time mapping for mortgage pool

Syntax

```
[CFlowAmounts,CFlowDates,TFactors,Factors,Payment,Principal,Interest,
Prepayment] = mbscfamounts(Settle,Maturity,IssueDate,GrossRate)
[CFlowAmounts,CFlowDates,TFactors,Factors,Payment,Principal,Interest,
Prepayment] = mbscfamounts( ___ ,CouponRate,Delay,PrepaySpeed,PrepayMatrix)
```

Description

`[CFlowAmounts,CFlowDates,TFactors,Factors,Payment,Principal,Interest, Prepayment] = mbscfamounts(Settle,Maturity,IssueDate,GrossRate)` computes cash flows between `Settle` and `Maturity` dates, the corresponding time factors in months from `Settle` and the mortgage factor (the fraction of loan principal outstanding).

Note Unlike `mbspassthrough`, `mbscfamounts` does not accept an original balance amount as an input. `mbscfamounts` assumes an original balance of 1.

`[CFlowAmounts,CFlowDates,TFactors,Factors,Payment,Principal,Interest, Prepayment] = mbscfamounts( ___ ,CouponRate,Delay,PrepaySpeed,PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Calculate Cash Flow Amounts and Dates, Time Factors, and Mortgage Factors for a Single Mortgage

Given a mortgage with the following characteristics, compute the cash flow amounts and dates, the time factors, and the mortgage factors.

Define the mortgage characteristics.

```
Settle      = datetime(2002,4,17);
Maturity    = datetime(2030,1,1);
IssueDate   = datetime(2000,1,1);
GrossRate   = 0.08125;
CouponRate  = 0.075;
Delay       = 14;
PrepaySpeed = 100;
```

Use `mbscfamonts` to evaluate the mortgage.

```
[CFlowAmounts, CFLowDates, TFactors, Factors] = ...
mbscfamounts(Settle, Maturity, IssueDate, GrossRate, ...
CouponRate, Delay, PrepaySpeed)
```

```
CFlowAmounts = 1×334

   -0.0033    0.0118    0.0120    0.0121    0.0120    0.0119    0.0119    0.0118    0.0117    0.0

CFLowDates = 1×334

      731323      731337      731368      731398      731429      731460      731490      731521

TFactors = 1×334

         0    0.9333    1.9333    2.9333    3.9333    4.9333    5.9333    6.9333    7.9333    8.9

Factors = 1×334

    1.0000    0.9944    0.9887    0.9828    0.9769    0.9711    0.9653    0.9595    0.9538    0.9
```

The result is contained in four 334-element row vectors.

Compute Cash Flow Amounts and Dates, Time Factors, and Mortgage Factors for a Mortgage Portfolio

Given a portfolio of mortgage-backed securities, use `mbscfamounts` to compute the cash flows and other factors from the portfolio.

Define characteristics for a mortgage portfolio.

```
Settle   = [datetime(2000,1,13) ; datetime(2002,4,17) ; datetime(2002,5,17)];
Maturity    = datetime(2030,1,1);
IssueDate   = datetime(2000,1,1);
GrossRate   = 0.08125;
CouponRate  = [0.075; 0.07875; 0.0775];
Delay       = 14;
PrepaySpeed = 100;
```

Use `mbscfamonts` to evaluate the mortgage.

```
[CFlowAmounts, CFlowDates, TFactors, Factors] = ...
mbscfamounts(Settle, Maturity, IssueDate, GrossRate, ...
CouponRate, Delay, PrepaySpeed)
```

```
CFlowAmounts = 3×361

   -0.0025    0.0071    0.0072    0.0074    0.0076    0.0077    0.0079    0.0080    0.0082    0.0
   -0.0035    0.0121    0.0123    0.0124    0.0123    0.0122    0.0122    0.0121    0.0120    0.0
   -0.0034    0.0122    0.0123    0.0123    0.0122    0.0121    0.0121    0.0120    0.0119    0.0


CFlowDates = 3×361

      730498      730517      730546      730577      730607      730638      730668      730699
      731323      731337      731368      731398      731429      731460      731490      731521
```

```
      731353      731368      731398      731429      731460      731490      731521      731551

TFactors = 3×361

         0    1.0667    2.0667    3.0667    4.0667    5.0667    6.0667    7.0667    8.0667    9.0
         0    0.9333    1.9333    2.9333    3.9333    4.9333    5.9333    6.9333    7.9333    8.9
         0    0.9333    1.9333    2.9333    3.9333    4.9333    5.9333    6.9333    7.9333    8.9

Factors = 3×361

    1.0000    0.9992    0.9982    0.9970    0.9957    0.9942    0.9925    0.9907    0.9887    0.9
    1.0000    0.9944    0.9887    0.9828    0.9769    0.9711    0.9653    0.9595    0.9538    0.9
    1.0000    0.9942    0.9883    0.9824    0.9766    0.9707    0.9649    0.9592    0.9534    0.9
```

Each output is a 3-by-361 element matrix padded with `NaN`'s wherever elements are missing.

Calculate Payment, Principal, Interest, and Prepayment for a Single Mortgage

Given a mortgage with the following characteristics, compute payments, principal, interest, and prepayment.

Define the mortgage characteristics.

```
Settle      = datetime(2002,4,17);
Maturity    = datetime(2030,1,1);
IssueDate   = datetime(2000,1,1);
GrossRate   = 0.08125;
CouponRate  = 0.075;
Delay       = 14;
PrepaySpeed = 100;
```

Use `mbscfamonts` to evaluate the mortgage.

```
[Payment, Principal, Interest, Prepayment] = ...
mbscfamounts(Settle, Maturity, IssueDate, GrossRate, ...
CouponRate, Delay, PrepaySpeed)
```

```
Payment = 1×334

   -0.0033    0.0118    0.0120    0.0121    0.0120    0.0119    0.0119    0.0118    0.0117    0.0

Principal = 1×334

      731323      731337      731368      731398      731429      731460      731490      731521

Interest = 1×334

         0    0.9333    1.9333    2.9333    3.9333    4.9333    5.9333    6.9333    7.9333    8.9

Prepayment = 1×334
```

```
    1.0000    0.9944    0.9887    0.9828    0.9769    0.9711    0.9653    0.9595    0.9538    0.9
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NMBS`-by-1 vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `mbscfamounts` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NMBS`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `mbscfamounts` also accepts serial date numbers as inputs, but they are not recommended.

`IssueDate` — Issue date

datetime array | string array | date character vector

Issue date, specified as an `NMBS`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `mbscfamounts` also accepts serial date numbers as inputs, but they are not recommended.

`GrossRate` — Gross coupon rate (including fees)

vector of decimal values

Gross coupon rate (including fees), specified as an `NMBS`-by-1 vector of decimal values.

Data Types: `double`

`CouponRate` — Net coupon rate

`GrossRate` (default) | vector of decimal values

(Optional) Net coupon rate, specified as an `NMBS`-by-1 vector of decimal values.

Data Types: `double`

`Delay` — Delay (in days) between payment from homeowner and receipt by bondholder

`0` (no delay between payment and receipt) (default) | vector

(Optional) Delay (in days) between payment from homeowner and receipt by bondholder, specified as an `NMBS`-by-1 vector.

Data Types: `double`

`PrepaySpeed` — Speed relative to PSA standard

`0` (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an NMBS-by-1 vector. The PSA standard is 100.

Note Set the PrepaySpeed to [] if you input a customized PrepayMatrix.

Data Types: double

PrepayMatrix — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a NaN-padded matrix of size max(TermRemaining)-by-NMBS. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use PrepayMatrix only when PrepaySpeed is unspecified.

Data Types: double

Output Arguments

CFlowAmounts — Cash flow amounts
matrix

Cash flows starting from Settle through end of the last month (Maturity), returned as a NMBS-by-P matrix.

CFlowDates — Cash flow dates
matrix

Cash flow dates (including at Settle), returned as a NMBS-by-P matrix.

TFactors — Time factors
matrix

Time factors (in months from Settle), returned as a NMBS-by-P matrix.

Factors — Mortgage factors (the fraction of the balance still outstanding at the end of each month)
matrix

Mortgage factors (the fraction of the balance still outstanding at the end of each month), returned as a NMBS-by-P matrix.

Payment — Total monthly payment
matrix

Total monthly payment, returned as a NMBS-by-P matrix.

Principal — Principal portion of the payment
matrix

Principal portion of the payment, returned as a NMBS-by-P matrix.

`Interest` — Interest portion of the payment
matrix

Interest portion of the payment, returned as a `NMBS`-by-`P` matrix.

`Prepayment` — Unscheduled payment of principal
matrix

Unscheduled payment of principal, returned as a `NMBS`-by-`P` matrix.

Version History

Introduced in R2012a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `mbscfamounts` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] *PSA Uniform Practices*, SF-49

See Also

`mbspassthrough` | `mbsnoprepay` | `cmoseqcf` | `cmoschedcf` | `cmosched`

Topics
“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsconvp

Convexity of mortgage pool given price

Syntax

```
Convexity = mbsconvp(Price,Settle,Maturity,IssueDate,GrossRate)
Convexity = mbsconvp( ___ ,CouponRate,Delay,PrepaySpeed,PrepayMatrix)
```

Description

`Convexity = mbsconvp(Price,Settle,Maturity,IssueDate,GrossRate)` computes mortgage-backed security convexity, given time information, price at settlement, and optionally, a prepayment model.

`Convexity = mbsconvp( ___ ,CouponRate,Delay,PrepaySpeed,PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Compute a Mortgage-Backed Security Convexity

This example shows how to compute a mortgage-backed security convexity, given a mortgage-backed security with the following characteristics.

```
Price      = 101;
Settle     = '15-Apr-2002';
Maturity   = '1 Jan 2030';
IssueDate  = '1-Jan-2000';
GrossRate  = 0.08125;
CouponRate = 0.075;
Delay = 14;
Speed = 100;

Convexity = mbsconvp(Price, Settle, Maturity, IssueDate,...
GrossRate, CouponRate, Delay, Speed)

Convexity = 71.6299
```

Input Arguments

`Price` — Clean price for every $100 face value

vector

Clean price for every $100 face value, specified as an `NMBS`-by-`1` vector.

Data Types: `double`

Settle — Settlement dates
vector

Settlement dates, specified as an NMBS-by-1 vector of serial date numbers or a cell array of character vectors.

Data Types: double | cell

Maturity — Maturity dates
vector

Maturity dates, specified as an NMBS-by-1 vector of serial date numbers or a cell array of character vectors.

Data Types: double | cell

IssueDate — Issue dates
vector

Maturity dates, specified as an NMBS-by-1 vector of serial date numbers or a cell array of character vectors.

Data Types: double | cell

GrossRate — Gross coupon rate (including fees)
vector

Gross coupon rate (including fees), specified as an NMBS-by-1 vector of numeric decimals.

Data Types: double

CouponRate — Net coupon rate
GrossRate (default) | vector

(Optional) Net coupon rate, specified as an NMBS-by-1 vector of numeric decimals.

Data Types: double

Delay — Delay in days
0 (no delay) (default) | vector

(Optional) Delay in days, specified as an NMBS-by-1 vector.

Data Types: double

PrepaySpeed — Speed relative to PSA standard
0 (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an NMBS-by-1 vector. The PSA standard is 100.

Note Set the PrepaySpeed to [] if you input a customized PrepayMatrix.

Data Types: double

PrepayMatrix — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a `NaN`-padded matrix of size `max(TermRemaining)`-by-`NMBS`. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

Output Arguments

`Convexity` — Periodic convexity of mortgage pool

scalar numeric

Periodic convexity of mortgage pool, returned as a scalar numeric.

Version History

Introduced before R2006a

References

[1] *PSA Uniform Practices*, SF-49

See Also

`mbsconvy` | `mbsdurp` | `mbspassthrough` | `mbsnoprepay` | `mbsdury`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsconvy

Convexity of mortgage pool given yield

Syntax

```
Convexity = mbsconvy(Yield,Settle,Maturity,IssueDate,GrossRate)
Convexity = mbsconvy( ___ ,CouponRate,Delay,PrepaySpeed,PrepayMatrix)
```

Description

`Convexity = mbsconvy(Yield,Settle,Maturity,IssueDate,GrossRate)` computes mortgage-backed security convexity, given time information, semiannual mortgage yield, and optionally, a prepayment model.

`Convexity = mbsconvy( ___ ,CouponRate,Delay,PrepaySpeed,PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Convexity of a Mortgage Pool Given Yield

This example shows how to compute the convexity of mortgage pool given yield for a mortgage-backed security with the following characteristics.

```
Yield      = 0.07125;
Settle     = '15-Apr-2002';
Maturity   = '1 Jan 2030';
IssueDate  = '1-Jan-2000';
GrossRate  = 0.08125;
Speed      = 100;
CouponRate = 0.075;
Delay = 14;

Convexity = mbsconvy(Yield, Settle, Maturity, IssueDate, ...
GrossRate, CouponRate, Delay, Speed)

Convexity = 72.8263
```

Input Arguments

`Yield` — Mortgage yield, compounded monthly

vector

Mortgage yield, compounded monthly, specified as an `NMBS`-by-`1` vector in decimals.

Data Types: `double`

`Settle` — Settlement dates
vector

Settlement dates, specified as an `NMBS`-by-`1` vector of serial date numbers or a cell array of character vectors. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `cell`

`Maturity` — Maturity dates
vector

Maturity dates, specified as an `NMBS`-by-`1` vector of serial date numbers or a cell array of character vectors.

Data Types: `double` | `cell`

`IssueDate` — Issue dates
vector

Maturity dates, specified as an `NMBS`-by-`1` vector of serial date numbers or a cell array of character vectors.

Data Types: `double` | `cell`

`GrossRate` — Gross coupon rate (including fees)
vector

Gross coupon rate (including fees), specified as an `NMBS`-by-`1` vector of numeric decimals.

Data Types: `double`

`CouponRate` — Net coupon rate
`GrossRate` (default) | vector

(Optional) Net coupon rate, specified as an `NMBS`-by-`1` vector of numeric decimals.

Data Types: `double`

`Delay` — Delay in days
`0` (no delay) (default) | vector

(Optional) Delay in days, specified as an `NMBS`-by-`1` vector.

Data Types: `double`

`PrepaySpeed` — Speed relative to PSA standard
`0` (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an `NMBS`-by-`1` vector. The PSA standard is `100`.

Note Set the `PrepaySpeed` to `[]` if you input a customized `PrepayMatrix`.

Data Types: `double`

`PrepayMatrix` — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a `NaN`-padded matrix of size `max(TermRemaining)`-by-`NMBS`. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

Output Arguments

`Convexity` — Periodic convexity of mortgage pool

scalar numeric

Periodic convexity of mortgage pool, returned as a scalar numeric.

Version History

Introduced before R2006a

References

[1] *PSA Uniform Practices*, SF-49

See Also

`mbsdurp` | `mbspassthrough` | `mbsnoprepay` | `mbsdury` | `mbsconvp`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsdurp

Duration of mortgage pool given price

Syntax

```
[YearDuration,ModDuration] = mbsdurp(Price,Settle,Maturity,IssueDate,
GrossRate)
[YearDuration,ModDuration] = mbsdurp( ___ ,CouponRate,Delay,PrepaySpeed,
PrepayMatrix)
```

Description

`[YearDuration,ModDuration] = mbsdurp(Price,Settle,Maturity,IssueDate, GrossRate)` computes the mortgage-backed security Macaulay (`YearDuration`) in years and modified (`ModDuration`) durations in years, given time information, price at settlement, and optionally, a prepayment model.

`[YearDuration,ModDuration] = mbsdurp( ___ ,CouponRate,Delay,PrepaySpeed, PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Find the Duration of a Mortgage Pool

This example shows how to find the duration of mortgage pool given a mortgage-backed security with the following characteristics.

```
Price = 101;
Settle = datenum('15-Apr-2002');
Maturity = datenum('1 Jan 2030');
IssueDate = datenum('1-Jan-2000');
GrossRate = 0.08125;
CouponRate = 0.075;;
Delay = 14;
Speed = 100;

[YearDuration, ModDuration] = mbsdurp(Price, Settle, Maturity,...
IssueDate, GrossRate, CouponRate, Delay, Speed)
```

```
YearDuration = 6.4380

ModDuration = 6.2080
```

Input Arguments

`Price` — Clean price for every $100 face of issue

vector

Clean price for every $100 face of issue, specified as an NMBS-by-1 vector.

Data Types: double

Settle — Settlement dates
vector

Settlement dates, specified as an NMBS-by-1 vector of serial date numbers or a cell array of character vectors. Settle must be earlier than Maturity.

Data Types: double | cell

Maturity — Maturity dates
vector

Maturity dates, specified as an NMBS-by-1 vector of serial date numbers or a cell array of character vectors.

Data Types: double | cell

IssueDate — Issue dates
vector

Maturity dates, specified as an NMBS-by-1 vector of serial date numbers or a cell array of character vectors.

Data Types: double | cell

GrossRate — Gross coupon rate (including fees)
vector

Gross coupon rate (including fees), specified as an NMBS-by-1 vector of numeric decimals.

Data Types: double

CouponRate — Net coupon rate
GrossRate (default) | vector

(Optional) Net coupon rate, specified as an NMBS-by-1 vector of numeric decimals.

Data Types: double

Delay — Delay in days
0 (no delay) (default) | vector

(Optional) Delay in days, specified as an NMBS-by-1 vector.

Data Types: double

PrepaySpeed — Speed relative to PSA standard
0 (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an NMBS-by-1 vector. The PSA standard is 100.

Note Set the PrepaySpeed to [] if you input a customized PrepayMatrix.

Data Types: double

`PrepayMatrix` — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a `NaN`-padded matrix of size `max(TermRemaining)`-by-`NMBS`. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

Output Arguments

`YearDuration` — Macaulay duration in years
scalar numeric

Macaulay duration in years, returned as a scalar numeric.

`ModDuration` — Modified duration in years
scalar numeric

Modified duration in years, returned as a scalar numeric.

Version History

Introduced before R2006a

References

[1] *PSA Uniform Practices*, SF-49

See Also

`mbspassthrough` | `mbsnoprepay` | `mbsdury` | `mbsconvp` | `mbsconvy`

Topics
“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsdury

Duration of mortgage pool given yield

Syntax

```
[YearDuration,ModDuration] = mbsdury(Yield,Settle,Maturity,IssueDate,
GrossRate)
[YearDuration,ModDuration] = mbsdury( ___ ,CouponRate,Delay,PrepaySpeed,
PrepayMatrix)
```

Description

`[YearDuration,ModDuration] = mbsdury(Yield,Settle,Maturity,IssueDate, GrossRate)` computes the mortgage-backed security Macaulay (`YearDuration`) in years and modified (`ModDuration`) durations in years, given time information, yield to maturity, and optionally, a prepayment model.

`[YearDuration,ModDuration] = mbsdury( ___ ,CouponRate,Delay,PrepaySpeed, PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Find the Duration of a Mortgage Pool Given the Yield

This example shows how to find the duration of mortgage pool given a mortgage-backed security with the following characteristics.

```
Yield = 0.07298413;
Settle = '15-Apr-2002';
Maturity = '1 Jan 2030';
IssueDate = '1-Jan-2000';
GrossRate = 0.08125;
Speed = 100;
CouponRate = 0.075;
Delay = 14;

[YearDuration, ModDuration] = mbsdury(Yield, Settle, Maturity,...
IssueDate, GrossRate, CouponRate, Delay, Speed)
```

```
YearDuration = 6.4380
```

```
ModDuration = 6.2080
```

Input Arguments

`Yield` — Mortgage yield, compounded monthly

vector

Mortgage yield, compounded monthly, specified as an NMBS-by-1 vector in decimals.

Data Types: double

Settle — Settlement dates
vector

Settlement dates, specified as an NMBS-by-1 vector of serial date numbers or a cell array of character vectors. Settle must be earlier than Maturity.

Data Types: double | cell

Maturity — Maturity dates
vector

Maturity dates, specified as an NMBS-by-1 vector of serial date numbers or a cell array of character vectors.

Data Types: double | cell

IssueDate — Issue dates
vector

Maturity dates, specified as an NMBS-by-1 vector of serial date numbers or a cell array of character vectors.

Data Types: double | cell

GrossRate — Gross coupon rate (including fees)
vector

Gross coupon rate (including fees), specified as an NMBS-by-1 vector of numeric decimals.

Data Types: double

CouponRate — Net coupon rate
GrossRate (default) | vector

(Optional) Net coupon rate, specified as an NMBS-by-1 vector of numeric decimals.

Data Types: double

Delay — Delay in days
0 (no delay) (default) | vector

(Optional) Delay in days, specified as an NMBS-by-1 vector.

Data Types: double

PrepaySpeed — Speed relative to PSA standard
0 (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an NMBS-by-1 vector. The PSA standard is 100.

Note Set the PrepaySpeed to [] if you input a customized PrepayMatrix.

Data Types: double

PrepayMatrix — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a NaN-padded matrix of size max(TermRemaining)-by-NMBS. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use PrepayMatrix only when PrepaySpeed is unspecified.

Data Types: double

Output Arguments

YearDuration — Macaulay duration in years
scalar numeric

Macaulay duration in years, returned as a scalar numeric.

ModDuration — Modified duration in years
scalar numeric

Modified duration in years, returned as a scalar numeric.

Version History

Introduced before R2006a

References

[1] *PSA Uniform Practices*, SF-49

See Also

mbspassthrough | mbsnoprepay | mbsconvp | mbsconvy | mbsdurp

Topics
“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsnoprepay

End-of-month mortgage cash flows and balances without prepayment

Syntax

```
[Balance,Interest,Payment,Principal] = mbsnoprepay(OriginalBalance,GrossRate,
Term)
```

Description

`[Balance,Interest,Payment,Principal] = mbsnoprepay(OriginalBalance,GrossRate, Term)` computes end-of-month mortgage balance, interest payments, principal payments, and cash flow payments with zero prepayment rate.

`mbsnoprepay` returns amortizing cash flows and balances over a specified term with no prepayment. When the lengths of pass-throughs are not the same, MATLAB software pads the shorter ones with `NaN`.

Examples

Compute an Amortization Schedule from Mortgage Pools

Given mortgage pools with the following characteristics, compute an amortization schedule.

```
OriginalBalance = 400000000;
CouponRate = 0.08125;
Term = [357; 355]; % Three- and five-month old mortgage pools.

[Balance, Interest, Payment, Principal] = mbsnoprepay(OriginalBalance, CouponRate, Term)
```

```
Balance = 357×2
10^8 ×

    3.9973    3.9973
    3.9946    3.9946
    3.9919    3.9918
    3.9892    3.9890
    3.9864    3.9862
    3.9837    3.9834
    3.9809    3.9806
    3.9781    3.9778
    3.9753    3.9749
    3.9724    3.9720
      ⋮


Interest = 357×2
10^6 ×

    2.7083    2.7083
    2.7065    2.7065
```

```
    2.7047    2.7046
    2.7029    2.7028
    2.7010    2.7009
    2.6992    2.6990
    2.6973    2.6971
    2.6954    2.6952
    2.6935    2.6933
    2.6916    2.6913
      ⋮

Payment = 357×2
10^6 ×

    2.9759    2.9799
    2.9759    2.9799
    2.9759    2.9799
    2.9759    2.9799
    2.9759    2.9799
    2.9759    2.9799
    2.9759    2.9799
    2.9759    2.9799
    2.9759    2.9799
    2.9759    2.9799
      ⋮

Principal = 357×2
10^6 ×

    0.2675    0.2715
    0.2693    0.2734
    0.2712    0.2752
    0.2730    0.2771
    0.2749    0.2790
    0.2767    0.2809
    0.2786    0.2828
    0.2805    0.2847
    0.2824    0.2866
    0.2843    0.2885
      ⋮
```

Input Arguments

`OriginalBalance` — Original face value in dollars

vector

Original face value in dollars, specified as a table or an `NUMOBS`-by-1 matrix.

Data Types: `double`

`GrossRate` — Gross coupon rate (including fees)

vector of decimal values

Gross coupon rate (including fees), specified as an `NUMOBS`-by-1 vector of decimal values.

Data Types: `double`

`Term` — Term of the mortgage in months
vector

Term of the mortgage in months, specified as an `NUMOBS`-by-`1` vector.

Data Types: `double`

Output Arguments

`Balance` — End-of-month balances over the life of the pass-through
vector

End-of-month balances over the life of the pass-through, returned as a `Term`-by-`1` vector.

`Interest` — End-of-month interest payments over the life of the pass-through
vector

End-of-month interest payments over the life of the pass-through, returned as a `Term`-by-`1` vector.

`Payment` — End-of-month payments over the life of the pass-through
vector

End-of-month payments over the life of the pass-through, returned as a `Term`-by-`1` vector.

`Principal` — All scheduled end-of-month principal payments over the life of the pass-through
vector

All scheduled end-of-month principal payments over the life of the pass-through, returned as a `Term`-by-`1` vector.

Version History

Introduced before R2006a

See Also

`mbspassthrough` | `mbsconvp` | `mbsconvy` | `mbsdurp`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsoas2price

Price given option-adjusted spread

Syntax

```
Price = mbsoas2price(ZeroCurve,OAS,Settle,Maturity,IssueDate,GrossRate)
Price = mbsoas2price( ___ ,CouponRate,Delay,Interpolation,PrepaySpeed,
PrepayMatrix)
```

Description

`Price = mbsoas2price(ZeroCurve,OAS,Settle,Maturity,IssueDate,GrossRate)` computes the clean price of a pass-through security for each $100 face value of outstanding principal.

`Price = mbsoas2price( ___ ,CouponRate,Delay,Interpolation,PrepaySpeed, PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Theoretical Price of a Mortgage Pool

Given an option-adjusted spread, a spot curve, and a prepayment assumption, compute theoretical price of a mortgage pool. First, create the bonds matrix:

```
Bonds = [datenum('11/21/2002')  0        100  0  2  1;
         datenum('02/20/2003')  0        100  0  2  1;
         datenum('07/31/2004')  0.03     100  2  3  1;
         datenum('08/15/2007')  0.035    100  2  3  1;
         datenum('08/15/2012')  0.04875  100  2  3  1;
         datenum('02/15/2031')  0.05375  100  2  3  1];
```

Choose a settlement date.

```
Settle = datenum('20-Aug-2002');
```

Assume the following clean prices for the bonds:

```
Prices =  [ 98.97467;
            98.58044;
           100.10534;
            98.18054;
           101.38136;
            99.25411];
```

Use the following formula to compute spot compounding for the bonds:

```
SpotCompounding = 2*ones(size(Prices));
```

Compute the zero curve.

```
[ZeroRatesP, CurveDatesP] = zbtprice(Bonds, Prices, Settle);
ZeroCurve = [CurveDatesP, ZeroRatesP, SpotCompounding]
```

```
ZeroCurve = 6×3
10^5 ×

    7.3154    0.0000    0.0000
    7.3163    0.0000    0.0000
    7.3216    0.0000    0.0000
    7.3327    0.0000    0.0000
    7.3510    0.0000    0.0000
    7.4185    0.0000    0.0000
```

Assign the following parameters:

```
OAS           = [26.0502; 28.6348; 31.2222];
Maturity      = datenum('02-Jan-2030');
IssueDate     = datenum('02-Jan-2000');
GrossRate     = 0.08125;
CouponRate    = 0.075;
Delay         = 14;
Interpolation = 1;
PrepaySpeed   = [0 50 100];
```

Calculate the theoretical price from the option-adjusted spread.

```
Price = mbsoas2price(ZeroCurve, OAS, Settle, Maturity, ...
IssueDate, GrossRate, CouponRate, Delay, Interpolation, ...
PrepaySpeed)
```

```
Price = 3×1

  124.7133
  120.9534
  118.0698
```

Input Arguments

ZeroCurve — Zero curve

matrix

Zero curve, specified as a three-column matrix, where:

- Column 1 is dates using serial date numbers.
- Column 2 is spot rates with maturities corresponding to the dates in Column 1, in decimal (for example, 0.075).
- Column 3 is the compounding value of the rates in Column 2. (This is the agency spot rate on the settlement date.) Allowable compounding values are: 1 (annual), 2 (semiannual, 3 (three times per year), 4 (quarterly), 6 (bimonthly), 12 (monthly), and -1 (continuous).

For example:

```
[datenum('11/21/2002')  0        100  0  2  1;
 datenum('02/20/2003')  0        100  0  2  1;
```

```
 datenum('07/31/2004')  0.03     100  2  3  1;
 datenum('08/15/2007')  0.035    100  2  3  1;
 datenum('08/15/2012')  0.04875  100  2  3  1;
 datenum('02/15/2031')  0.05375  100  2  3  1];
```

Data Types: `double` | `cell`

`OAS` — Option-adjusted spreads
vector

Option-adjusted spreads, in basis points, specified as an `NMBS`-by-1 vector.

Data Types: `double`

`Settle` — Settlement date
serial date number | date character vector

Settlement date, specified as an `NMBS`-by-1 vector using serial date numbers or date character vectors. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char`

`Maturity` — Maturity date
serial date number | date character vector

Maturity date, specified as an `NMBS`-by-1 vector using serial date numbers or date character vectors.

Data Types: `double` | `char`

`IssueDate` — Issue date
serial date number | date character vector

Issue date, specified as an `NMBS`-by-1 vector using serial date numbers or date character vectors.

Data Types: `double` | `char`

`GrossRate` — Gross coupon rate (including fees)
vector of decimal values

Gross coupon rate (including fees), specified as an `NMBS`-by-1 vector of decimal values.

Data Types: `double`

`CouponRate` — Net coupon rate
`GrossRate` (default) | vector of decimal values

(Optional) Net coupon rate, specified as an `NMBS`-by-1 vector of decimal values.

Data Types: `double`

`Delay` — Delay (in days) between payment from homeowner and receipt by bondholder
`0` (no delay between payment and receipt) (default) | vector

(Optional) Delay (in days) between payment from homeowner and receipt by bondholder, specified as an `NMBS`-by-1 vector.

Data Types: `double`

`Interpolation` — Interpolation method to compute the corresponding spot rates for the bond's cash flow
`1` (linear) (default) | vector

(Optional) Interpolation method to compute the corresponding spot rates for the bond's cash flow, specified as an `NMBS`-by-`1` vector. Available methods are (`0`) nearest, (`1`) linear, and (`2`) cubic spline. For more information on the supported interpolation methods, see `interp1`.

Data Types: `double`

`PrepaySpeed` — Speed relative to PSA standard
`0` (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an `NMBS`-by-`1` vector. The PSA standard is `100`.

Note Set the `PrepaySpeed` to `[]` if you input a customized `PrepayMatrix`.

Data Types: `double`

`PrepayMatrix` — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a `NaN`-padded matrix of size `max(TermRemaining)`-by-`NMBS`. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

Output Arguments

`Price` — Clean price of passthrough per $100 face of principal outstanding
vector

Clean price of passthrough per $100 face of principal outstanding, returned as a `NMBS`-by-`1` vector.

Version History

Introduced before R2006a

References

[1] *PSA Uniform Practices*, SF-49

See Also

`mbspassthrough` | `mbsprice2oas` | `mbsyield2oas`

Topics
“Generating Prepayment Vectors” on page 5-4

“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsoas2yield

Yield given option-adjusted spread

Syntax

```
[MYield,BEMBSYield] = mbsoas2yield(ZeroCurve,OAS,Settle,Maturity,IssueDate,
GrossRate)
[MYield,BEMBSYield] = mbsoas2yield( ___ ,CouponRate,Delay,Interpolation,
PrepaySpeed,PrepayMatrix)
```

Description

`[MYield,BEMBSYield] = mbsoas2yield(ZeroCurve,OAS,Settle,Maturity,IssueDate, GrossRate)` computes the mortgage and bond-equivalent yields of a pass-through security.

`[MYield,BEMBSYield] = mbsoas2yield( ___ ,CouponRate,Delay,Interpolation, PrepaySpeed,PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Theoretical Yield to Maturity of a Mortgage Pool

Given an option-adjusted spread, a spot curve, and a prepayment assumption, compute the theoretical yield to maturity of a mortgage pool. First, create the bonds matrix:

```
Bonds = [datenum('11/21/2002')  0        100  0  2  1;
         datenum('02/20/2003')  0        100  0  2  1;
         datenum('07/31/2004')  0.03     100  2  3  1;
         datenum('08/15/2007')  0.035    100  2  3  1;
         datenum('08/15/2012')  0.04875  100  2  3  1;
         datenum('02/15/2031')  0.05375  100  2  3  1];
```

Choose a settlement date.

```
Settle = datenum('20-Aug-2002');
```

Assume the following clean prices for the bonds:

```
Prices =  [ 98.97467;
            98.58044;
           100.10534;
            98.18054;
           101.38136;
            99.25411];
```

Use the following formula to compute spot compounding for the bonds:

```
SpotCompounding = 2*ones(size(Prices));
```

Compute the zero curve.

```
[ZeroRatesP, CurveDatesP] = zbtprice(Bonds, Prices, Settle);
ZeroCurve = [CurveDatesP, ZeroRatesP, SpotCompounding]
```

```
ZeroCurve = 6×3
10^5 ×

    7.3154    0.0000    0.0000
    7.3163    0.0000    0.0000
    7.3216    0.0000    0.0000
    7.3327    0.0000    0.0000
    7.3510    0.0000    0.0000
    7.4185    0.0000    0.0000
```

Assign the following parameters:

```
OAS           = [26.0502; 28.6348; 31.2222];
Maturity      = datenum('02-Jan-2030');
IssueDate     = datenum('02-Jan-2000');
GrossRate     = 0.08125;
CouponRate    = 0.075;
Delay         = 14;
Interpolation = 1;
PrepaySpeed   = [0 50 100];
```

Compute the mortgage yield and bond equivalent mortgage yield.

```
[MYield BEMBSYield] = mbsoas2yield(ZeroCurve, OAS, Settle, ...
Maturity, IssueDate, GrossRate, CouponRate, Delay, ...
Interpolation, PrepaySpeed)
```

```
MYield = 3×1

    0.0527
    0.0513
    0.0499


BEMBSYield = 3×1

    0.0533
    0.0518
    0.0504
```

Input Arguments

ZeroCurve — Zero curve

matrix

Zero curve, specified as a three-column matrix, where:

- Column 1 is dates using serial date numbers.
- Column 2 is spot rates with maturities corresponding to the dates in Column 1, in decimal (for example, 0.075).

- Column 3 is the compounding value of the rates in Column 2. (This is the agency spot rate on the settlement date.) Allowable compounding values are: 1 (annual), 2 (semiannual, 3 (three times per year), 4 (quarterly), 6 (bimonthly), 12 (monthly), and -1 (continuous).

For example:

```
[datenum('11/21/2002')  0        100  0  2  1;
 datenum('02/20/2003')  0        100  0  2  1;
 datenum('07/31/2004')  0.03     100  2  3  1;
 datenum('08/15/2007')  0.035    100  2  3  1;
 datenum('08/15/2012')  0.04875  100  2  3  1;
 datenum('02/15/2031')  0.05375  100  2  3  1];
```

Data Types: `double` | `cell`

`OAS` — Option-adjusted spreads
vector

Option-adjusted spreads, in basis points, specified as an `NMBS`-by-1 vector.

Data Types: `double`

`Settle` — Settlement date
serial date number | date character vector

Settlement date, specified as an `NMBS`-by-1 vector using serial date numbers or date character vectors. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char`

`Maturity` — Maturity date
serial date number | date character vector

Maturity date, specified as an `NMBS`-by-1 vector using serial date numbers or date character vectors.

Data Types: `double` | `char`

`IssueDate` — Issue date
serial date number | date character vector

Issue date, specified as an `NMBS`-by-1 vector using serial date numbers or date character vectors.

Data Types: `double` | `char`

`GrossRate` — Gross coupon rate (including fees)
vector of decimal values

Gross coupon rate (including fees), specified as an `NMBS`-by-1 vector of decimal values.

Data Types: `double`

`CouponRate` — Net coupon rate
`GrossRate` (default) | vector of decimal values

(Optional) Net coupon rate, specified as an `NMBS`-by-1 vector of decimal values.

Data Types: `double`

`Delay` — Delay (in days) between payment from homeowner and receipt by bondholder
`0` (no delay between payment and receipt) (default) | vector

(Optional) Delay (in days) between payment from homeowner and receipt by bondholder, specified as an `NMBS`-by-`1` vector.

Data Types: `double`

`Interpolation` — Interpolation method to compute the corresponding spot rates for the bond's cash flow
`1` (linear) (default) | vector

(Optional) Interpolation method to compute the corresponding spot rates for the bond's cash flow, specified as an `NMBS`-by-`1` vector. Available methods are (`0`) nearest, (`1`) linear, and (`2`) cubic spline. For more information on the supported interpolation methods, see `interp1`.

Data Types: `double`

`PrepaySpeed` — Speed relative to PSA standard
`0` (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an `NMBS`-by-`1` vector. The PSA standard is `100`.

Note Set the `PrepaySpeed` to `[]` if you input a customized `PrepayMatrix`.

Data Types: `double`

`PrepayMatrix` — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a `NaN`-padded matrix of size `max(TermRemaining)`-by-`NMBS`. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

Output Arguments

`MYield` — Yield to maturity of the mortgage-backed security
vector

Yield to maturity of the mortgage-backed security, returned as a `NMBS`-by-`1` vector. This yield is compounded monthly (12 times a year).

`BEMBSYield` — Bond equivalent yield of the mortgage-backed security
vector

Bond equivalent yield of the mortgage-backed security, returned as a `NMBS`-by-`1` vector. This yield is compounded semiannually (two times a year).

Version History

Introduced before R2006a

References

[1] *PSA Uniform Practices*, SF-49

See Also

`mbspassthrough` | `mbsprice2oas` | `mbsyield2oas` | `mbsoas2price`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbspassthrough

Mortgage pool cash flows and balances with prepayment

Syntax

```
[Balance,Payment,Principal,Interest,Prepayment] = mbspassthrough(
OriginalBalance,GrossRate,OriginalTerm)
[Balance,Payment,Principal,Interest,Prepayment] = mbspassthrough( ___ ,
TermRemaining,PrepaySpeed,PrepayMatrix)
```

Description

`[Balance,Payment,Principal,Interest,Prepayment] = mbspassthrough( OriginalBalance,GrossRate,OriginalTerm)` calculates mortgage pool cash flows and balances with prepayments.

If a standard (PSA) prepayment is specified, "aging" is applied to standard prepayment vector. Aging is the same amount as the age of the pool (`OriginalTerm` - `TermRemaining`).

`[Balance,Payment,Principal,Interest,Prepayment] = mbspassthrough( ___ , TermRemaining,PrepaySpeed,PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Cash Flow of Principal, Interest, and Prepayment of a Pass-Through Security

This example shows how to compute the cash flows and balances of a 3-month old mortgage pool with original term of 360 months, assuming a prepayment speed of 100.

```
OriginalBalance = 100000;
GrossRate = 0.08125;
OriginalTerm = 360;
TermRemaining = 357;
PrepaySpeed = 100;

[Balance, Payment, Principal, Interest, Prepayment] =...
mbspassthrough(OriginalBalance, GrossRate, OriginalTerm,...
TermRemaining, PrepaySpeed)
```

```
Balance = 357×1
10^4 ×

    9.9866
    9.9715
    9.9548
    9.9363
    9.9161
    9.8942
    9.8707
```

```
    9.8454
    9.8185
    9.7900
       ⋮

Payment = 357×1

  743.9671
  743.4693
  742.8468
  742.0999
  741.2285
  740.2329
  739.1132
  737.8699
  736.5034
  735.0139
      ⋮

Principal = 357×1

   66.8837
   67.2915
   67.6904
   68.0802
   68.4607
   68.8317
   69.1929
   69.5442
   69.8854
   70.2163
      ⋮

Interest = 357×1

  677.0833
  676.1777
  675.1564
  674.0196
  672.7678
  671.4012
  669.9203
  668.3257
  666.6179
  664.7976
      ⋮

Prepayment = 357×1

   66.8676
   83.5494
  100.2000
  116.8108
  133.3731
  149.8785
```

```
  166.3183
  182.6840
  198.9672
  215.1593
      ⋮
```

Input Arguments

`OriginalBalance` — Original balance value in dollars

vector

Original balance value in dollars (balance at the beginning of each `TermRemaining`), specified as an `NMBS`-by-`1` vector.

Data Types: `double`

`GrossRate` — Gross coupon rate (including fees)

vector of decimal values

Gross coupon rate (including fees), specified as an `NMBS`-by-`1` vector of decimal values.

Data Types: `double`

`OriginalTerm` — Term of the mortgage in months

vector

Term of the mortgage in months, specified as an `NMBS`-by-`1` vector.

Data Types: `double`

`TermRemaining` — Number of full months between settlement and maturity

`OriginalTerm` (default) | vector

(Optional) Number of full months between settlement and maturity, specified as an `NMBS`-by-`1` vector. For this argument, "full months" means not including fractional first term (if there is one).

Data Types: `double`

`PrepaySpeed` — Speed relative to PSA standard

`0` (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an `NMBS`-by-`1` vector. The PSA standard is `100`.

Note Set the `PrepaySpeed` to `[]` if you input a customized `PrepayMatrix`.

Data Types: `double`

`PrepayMatrix` — Customized prepayment vector

matrix

(Optional) Customized prepayment vector, specified as a `NaN`-padded matrix of size `max(TermRemaining)`-by-`NMBS`. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

Output Arguments

`Balance` — Principal balance at end of month

vector

Principal balance at end of month, returned as a `TermRemaining`-by-`1` vectors of end-of-month values.

`Payment` — Total monthly payment

vector

Total monthly payment, returned as a `TermRemaining`-by-`1` vectors of end-of-month values.

`Principal` — Principal portion of the payment

vector

Principal portion of the payment, returned as a `TermRemaining`-by-`1` vectors of end-of-month values.

`Interest` — Interest portion of the payment

vector

Interest portion of the payment, returned as a `TermRemaining`-by-`1` vectors of end-of-month values.

`Prepayment` — Unscheduled principal payment

vector

Unscheduled principal payment, returned as a `TermRemaining`-by-`1` vectors of end-of-month values.

Version History

Introduced before R2006a

See Also

`mbswal`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsprice

Mortgage-backed security price given yield

Syntax

```
[Price,AccrInt] = mbsprice(Yield,Settle,Maturity,IssueDate,GrossRate)
[Price,AccrInt] = mbsprice( ___ CouponRate,Delay,PrepaySpeed,PrepayMatrix)
```

Description

`[Price,AccrInt] = mbsprice(Yield,Settle,Maturity,IssueDate,GrossRate)` computes a mortgage-backed security price, given time information and mortgage yield at settlement.

`[Price,AccrInt] = mbsprice( ___ CouponRate,Delay,PrepaySpeed,PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Determine the Mortgage-Backed Security Price Given the Yield

This example shows how to determine the mortgage-backed security price given a mortgage-backed security with the following characteristics.

```
Yield = 0.0725;
Settle = datetime(2002,4,15);
Maturity = datetime(2030,1,1);
IssueDate = datetime(2000,1,1);
GrossRate = 0.08125;
CouponRate = 0.075;
Delay = 14;
Speed = 100;

[Price AccrInt] = mbsprice(Yield, Settle, Maturity, IssueDate,...
GrossRate, CouponRate, Delay, Speed)

Price = 101.3147

AccrInt = 0.2917
```

Determine the Mortgage-Backed Security Price Using a Customized `PrePaytMatrix`

This example shows how to determine the mortgage-backed security price, given a mortgage-backed security, and `PrePaytMatrix` with the following characteristics:

```
Yield = 0.0725;
Settle = datetime(2002,4,15);
Maturity = datetime(2030,1,1);
```

```
IssueDate = datetime(2000,1,1);
GrossRate = 0.08125;
PrepayMatrix = 0.005*ones(360,1);

[Price AccrInt] = mbsprice(Yield, Settle, Maturity, IssueDate,...
GrossRate, PrepayMatrix)
```

```
Price = 360×1

   34.8583
   34.8583
   34.8583
   34.8583
   34.8583
   34.8583
   34.8583
   34.8583
   34.8583
   34.8583
      ⋮
```

```
AccrInt = 360×1

    0.0194
    0.0194
    0.0194
    0.0194
    0.0194
    0.0194
    0.0194
    0.0194
    0.0194
    0.0194
      ⋮
```

Input Arguments

`Yield` — Mortgage yield compounded monthly

vector of decimal values

Mortgage yield compounded monthly, specified as an `NMBS`-by-`1` vector using decimal values.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NMBS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `mbsprice` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an NMBS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, mbsprice also accepts serial date numbers as inputs, but they are not recommended.

IssueDate — Issue date
datetime array | string array | date character vector

Issue date, specified as an NMBS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, mbsprice also accepts serial date numbers as inputs, but they are not recommended.

GrossRate — Gross coupon rate (including fees)
vector of decimal values

Gross coupon rate (including fees), specified as an NMBS-by-1 vector of decimal values.

Data Types: double

CouponRate — Net coupon rate
GrossRate (default) | vector of decimal values

(Optional) Net coupon rate, specified as an NMBS-by-1 vector of decimal values.

Data Types: double

Delay — Delay (in days) between payment from homeowner and receipt by bondholder
0 (no delay between payment and receipt) (default) | vector

(Optional) Delay (in days) between payment from homeowner and receipt by bondholder, specified as an NMBS-by-1 vector.

Data Types: double

PrepaySpeed — Speed relative to PSA standard
0 (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an NMBS-by-1 vector. The PSA standard is 100.

Note Set the PrepaySpeed to [] if you input a customized PrepayMatrix.

Data Types: double

PrepayMatrix — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a NaN-padded matrix of size max(TermRemaining)-by-NMBS. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use PrepayMatrix only when PrepaySpeed is unspecified.

Data Types: `double`

Output Arguments

`Price` — Clean price for every $100 face value of the securities

vector

Clean price for every $100 face value of the securities, returned as a `NMBS`-by-`1` vector.

`AccrInt` — Accrued interest of the mortgage-backed securities

vector

Accrued interest of the mortgage-backed securities, returned as a `NMBS`-by-`1` vector.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `mbsprice` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] *PSA Uniform Practices*, SF-49

See Also

`mbsyield`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsprice2oas

Option-adjusted spread given price

Syntax

```
OAS = mbsprice2oas(ZeroCurve,Price,Settle,Maturity,IssueDate,GrossRate)
OAS = mbsprice2oas( ___ ,CouponRate,Delay,Interpolation,PrepaySpeed,
PrepayMatrix)
```

Description

`OAS = mbsprice2oas(ZeroCurve,Price,Settle,Maturity,IssueDate,GrossRate)` computes the option-adjusted spread in basis points.

`OAS = mbsprice2oas( ___ ,CouponRate,Delay,Interpolation,PrepaySpeed, PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Calculate the Option-Adjusted Spread of a 30-Year Fixed-Rate Mortgage

Calculate the option-adjusted spread of a 30-year fixed-rate mortgage with about a 28-year weighted average maturity remaining, given assumptions of 0, 50, and 100 PSA prepayments. First, create the bonds matrix:

```
Bonds = [datenum('11/21/2002')  0        100  0  2  1;
         datenum('02/20/2003')  0        100  0  2  1;
         datenum('07/31/2004')  0.03     100  2  3  1;
         datenum('08/15/2007')  0.035    100  2  3  1;
         datenum('08/15/2012')  0.04875  100  2  3  1;
         datenum('02/15/2031')  0.05375  100  2  3  1];
```

Choose a settlement date.

```
Settle = datenum('20-Aug-2002');
```

Assume the following clean prices for the bonds:

```
Prices =  [ 98.97467;
            98.58044;
           100.10534;
            98.18054;
           101.38136;
            99.25411];
```

Use the following formula to compute spot compounding for the bonds:

```
SpotCompounding = 2*ones(size(Prices));
```

Compute the zero curve.

```
[ZeroRatesP, CurveDatesP] = zbtprice(Bonds, Prices, Settle);
ZeroCurve = [CurveDatesP, ZeroRatesP, SpotCompounding]
```

```
ZeroCurve = 6×3
10^5 ×

    7.3154    0.0000    0.0000
    7.3163    0.0000    0.0000
    7.3216    0.0000    0.0000
    7.3327    0.0000    0.0000
    7.3510    0.0000    0.0000
    7.4185    0.0000    0.0000
```

Assign the following parameters:

```
Price         = 95;
Maturity      = datenum('02-Jan-2030');
IssueDate     = datenum('02-Jan-2000');
GrossRate     = 0.08125;
CouponRate    = 0.075;
Delay         = 14;
Interpolation = 1;
PrepaySpeed   = [0; 50; 100];
Interpolation = 1;
```

Compute the option-adjusted spread.

```
OAS = mbsprice2oas(ZeroCurve, Price, Settle, Maturity, ...
IssueDate, GrossRate, CouponRate, Delay, Interpolation, ...
PrepaySpeed)
```

```
OAS = 3×1

   26.0508
   28.6355
   31.2232
```

Input Arguments

ZeroCurve — Zero curve

matrix

Zero curve, specified as a three-column matrix, where:

- Column 1 is dates using datetimes.
- Column 2 is spot rates with maturities corresponding to the dates in Column 1, in decimal (for example, 0.075).
- Column 3 is the compounding value of the rates in Column 2. (This is the agency spot rate on the settlement date.) Allowable compounding values are: 1 (annual), 2 (semiannual, 3 (three times per year), 4 (quarterly), 6 (bimonthly), 12 (monthly), and -1 (continuous).

For example:

```
[datetime(2003,1,1)  0.0154  12;
 datetime(2004,1,1)  0.0250  12;
```

```
 ......
 datetime(2020,1,1)  0.0675   2];
```

Data Types: `datetime` | `cell`

`Price` — Clean price for every $100 face value of bond issue
vector

Clean price for every $100 face value of bond issue, specified as an `NMBS`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date
string array | date character vector | serial date number

Settlement date, specified as an `NMBS`-by-`1` vector using a string array, date character vectors, or serial date numbers. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `string`

`Maturity` — Maturity date
string array | date character vector | serial date number

Maturity date, specified as an `NMBS`-by-`1` vector using a string array, date character vectors, or serial date numbers.

Data Types: `double` | `char` | `string`

`IssueDate` — Issue date
datetime array | string array | date character vector | serial date number

Issue date, specified as an `NMBS`-by-`1` vector using a string array, date character vectors, or serial date numbers.

Data Types: `double` | `char` | `string`

`GrossRate` — Gross coupon rate (including fees)
vector of decimal values

Gross coupon rate (including fees), specified as an `NMBS`-by-`1` vector of decimal values.

Data Types: `double`

`CouponRate` — Net coupon rate
`GrossRate` (default) | vector of decimal values

(Optional) Net coupon rate, specified as an `NMBS`-by-`1` vector of decimal values.

Data Types: `double`

`Delay` — Delay (in days) between payment from homeowner and receipt by bondholder
`0` (no delay between payment and receipt) (default) | vector

(Optional) Delay (in days) between payment from homeowner and receipt by bondholder, specified as an `NMBS`-by-`1` vector.

Data Types: `double`

`Interpolation` — Interpolation method to compute the corresponding spot rates for the bond's cash flow
`1` (linear) (default) | vector

(Optional) Interpolation method to compute the corresponding spot rates for the bond's cash flow, specified as an `NMBS`-by-`1` vector. Available methods are (`0`) nearest, (`1`) linear, and (`2`) cubic spline. For more information on the supported interpolation methods, see `interp1`.

Data Types: `double`

`PrepaySpeed` — Speed relative to PSA standard
`0` (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an `NMBS`-by-`1` vector. The PSA standard is `100`.

Note Set the `PrepaySpeed` to `[]` if you input a customized `PrepayMatrix`.

Data Types: `double`

`PrepayMatrix` — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a `NaN`-padded matrix of size `max(TermRemaining)`-by-`NMBS`. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

Output Arguments

`OAS` — Zero volatility OAS
vector

Zero volatility OAS, in basis point (bp), returned as a `NMBS`-by-`1` vector.

Version History
Introduced before R2006a

References

[1] *PSA Uniform Practices*, SF-49

See Also
`mbsoas2price` | `mbsoas2yield` | `mbsyield2oas`

Topics
“Generating Prepayment Vectors” on page 5-4

“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsprice2speed

Implied PSA prepayment speeds given price

Syntax

```
[ImpSpdOnPrc,ImpSpdOnDur,ImpSpdOnCnv] = mbsprice2speed(Price,Settle,Maturity,
IssueDate,GrossRate,PrepayMatrix)
[ImpSpdOnPrc,ImpSpdOnDur,ImpSpdOnCnv] = mbsprice2speed( ___ ,CouponRate,Delay)
```

Description

`[ImpSpdOnPrc,ImpSpdOnDur,ImpSpdOnCnv] = mbsprice2speed(Price,Settle,Maturity, IssueDate,GrossRate,PrepayMatrix)` computes PSA prepayment speeds implied by pool prices and projected (user-defined) prepayment vectors. The calculated PSA speed produces the same price, modified duration, or modified convexity, depending upon the output requested.

`[ImpSpdOnPrc,ImpSpdOnDur,ImpSpdOnCnv] = mbsprice2speed( ___ ,CouponRate,Delay)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Compute PSA Prepayment Speeds

This example shows how to compute the equivalent PSA benchmark prepayment speeds for a mortgage pool with the following characteristics and prepayment matrix.

```
Price        = 101;
Settle       = datetime(2000,1,1);
Maturity     = datetime(2030,1,1);
IssueDate    = datetime(2000,1,1);
GrossRate    = 0.08125;
PrepayMatrix = 0.005*ones(360,1);
CouponRate   = 0.075;
Delay        = 14;

[ImpSpdOnPrc, ImpSpdOnDur, ImpSpdOnCnv] = ...
mbsprice2speed(Price,Settle, Maturity, IssueDate, ...
GrossRate, PrepayMatrix, CouponRate, Delay)

ImpSpdOnPrc = 118.5980

ImpSpdOnDur = 118.3946

ImpSpdOnCnv = 109.5115
```

Input Arguments

`Price` — Clean price for every $100 face value

vector

Clean price for every $100 face value, specified as an `NMBS`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NMBS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `mbsprice2speed` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NMBS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `mbsprice2speed` also accepts serial date numbers as inputs, but they are not recommended.

`IssueDate` — Issue date

datetime array | string array | date character vector

Issue date, specified as an `NMBS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `mbsprice2speed` also accepts serial date numbers as inputs, but they are not recommended.

`GrossRate` — Gross coupon rate (including fees)

vector of decimal values

Gross coupon rate (including fees), specified as an `NMBS`-by-`1` vector of decimal values.

Data Types: `double`

`PrepayMatrix` — Customized prepayment vector

matrix

Customized prepayment vector, specified as a `NaN`-padded matrix of size `max(TermRemaining)`-by-`NMBS`. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

`CouponRate` — Net coupon rate
`GrossRate` (default) | vector of decimal values

(Optional) Net coupon rate, specified as an `NMBS`-by-`1` vector of decimal values.

Data Types: `double`

`Delay` — Delay (in days) between payment from homeowner and receipt by bondholder
`0` (no delay between payment and receipt) (default) | vector

(Optional) Delay (in days) between payment from homeowner and receipt by bondholder, specified as an `NMBS`-by-`1` vector.

Data Types: `double`

Output Arguments

`ImpSpdOnPrc` — Equivalent PSA benchmark prepayment speed for the pass-through to carry the same price
vector

Equivalent PSA benchmark prepayment speed for the pass-through to carry the same price, returned as a `NMBS`-by-`1` vector.

`ImpSpdOnDur` — Equivalent PSA benchmark prepayment speed for the pass-through to carry the same modified duration
vector

Equivalent PSA benchmark prepayment speed for the pass-through to carry the same modified duration, returned as a `NMBS`-by-`1` vector.

`ImpSpdOnCnv` — Equivalent PSA benchmark prepayment speed for the pass-through to carry the same modified convexity
vector

Equivalent PSA benchmark prepayment speed for the pass-through to carry the same modified convexity, returned as a `NMBS`-by-`1` vector.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `mbsprice2speed` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] *PSA Uniform Practices,* SF-49

See Also

`mbsprice` | `mbsyield2speed`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbswal

Weighted average life of mortgage pool

Syntax

```
WAL = mbswal(Settle,Maturity,IssueDate,GrossRate)
WAL = mbswal( ___ CouponRate,Delay,PrepaySpeed,PrepayMatrix)
```

Description

`WAL = mbswal(Settle,Maturity,IssueDate,GrossRate)` computes the weighted average life, in number of years, of a mortgage pool, as measured from the settlement date.

`WAL = mbswal( ___ CouponRate,Delay,PrepaySpeed,PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Determine the Weighted Average Life of a Mortgage Pool

This example shows how to determine the weighted average life of a mortgage pool, given a pass-through security with the following characteristics.

```
Settle = datetime(2002,4,15);
Maturity = datetime(2030,1,1);
IssueDate = datetime(2000,1,1);
GrossRate = 0.08125;
CouponRate = 0.075;
Delay = 14;
Speed = 100;

WAL = mbswal(Settle, Maturity, IssueDate, GrossRate, ...
CouponRate, Delay, Speed)
```

```
WAL = 10.5477
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an NMBS-by-1 vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `mbswal` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an NMBS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, mbswal also accepts serial date numbers as inputs, but they are not recommended.

IssueDate — Issue date
datetime array | string array | date character vector

Issue date, specified as an NMBS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, mbswal also accepts serial date numbers as inputs, but they are not recommended.

GrossRate — Gross coupon rate (including fees)
vector of decimal values

Gross coupon rate (including fees), specified as an NMBS-by-1 vector of decimal values.

Data Types: double

CouponRate — Net coupon rate
GrossRate (default) | vector of decimal values

(Optional) Net coupon rate, specified as an NMBS-by-1 vector of decimal values.

Data Types: double

Delay — Delay (in days) between payment from homeowner and receipt by bondholder
0 (no delay between payment and receipt) (default) | vector

(Optional) Delay (in days) between payment from homeowner and receipt by bondholder, specified as an NMBS-by-1 vector.

Data Types: double

PrepaySpeed — Speed relative to PSA standard
0 (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an NMBS-by-1 vector. The PSA standard is 100.

Note Set the PrepaySpeed to [] if you input a customized PrepayMatrix.

Data Types: double

PrepayMatrix — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a NaN-padded matrix of size max(TermRemaining)-by-NMBS. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use PrepayMatrix only when PrepaySpeed is unspecified.

Data Types: `double`

Output Arguments

`WAL` — Weighted Average Life of MBS, in number of years

vector

Weighted Average Life (WAL) of MBS, in number of years, returned as a `NMBS`-by-`1` vector.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `mbswal` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] *PSA Uniform Practices*, SF-49

See Also

`mbspassthrough`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsyield

Mortgage-backed security yields given price

Syntax

```
[MYield,BEMBSYield] = mbsyield(Price,Settle,Maturity,IssueDate,GrossRate)
[MYield,BEMBSYield] = mbsyield( ___ CouponRate,Delay,PrepaySpeed,PrepayMatrix)
```

Description

`[MYield,BEMBSYield] = mbsyield(Price,Settle,Maturity,IssueDate,GrossRate)` computes a mortgage-backed security yield to maturity and the bond equivalent yield, given time information, and price at settlement.

`[MYield,BEMBSYield] = mbsyield( ___ CouponRate,Delay,PrepaySpeed,PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Determine a Mortgage-Backed Security Yield Given the Price

This example shows how to determine the mortgage-backed security yield, given a mortgage-backed security with the following characteristics.

```
Price = 102;
Settle = datetime(2002,4,15);
Maturity = datetime(2030,1,1);
IssueDate = datetime(2000,1,1);
GrossRate = 0.08125;
CouponRate = 0.075;
Delay = 14;
Speed = 100;

[MYield, BEMBSYield] = mbsyield(Price, Settle, Maturity, ...
IssueDate, GrossRate, CouponRate, Delay,  Speed)

MYield = 0.0715

BEMBSYield = 0.0725
```

Determine Multiple Mortgage-Backed Securities Yields Given the Price

This example shows how to determine multiple mortgage-backed securities yields, given a portfolio of mortgage-backed securities with the following characteristics.

```
Price = 102;
Settle = [datetime(2000,2,13) ; datetime(2002,4,17) ; datetime(2002,5,17) ; datetime(2000,1,13)]
```

```
Maturity  = datetime(2030,1,1);
IssueDate = datetime(2000,1,1);
GrossRate = 0.08125;
CouponRate = [0.075; 0.07875; 0.0775; 0.08125];
Delay = 14;
Speed = 100;

[MYield, BEMBSYield] = mbsyield(Price, Settle, Maturity,...
IssueDate, GrossRate, CouponRate, Delay,  Speed)
```

```
MYield = 4×1

    0.0717
    0.0751
    0.0739
    0.0779


BEMBSYield = 4×1

    0.0728
    0.0763
    0.0750
    0.0791
```

Input Arguments

`Price` — Clean price for every $100 face value

vector

Clean price for every $100 face value, specified as an NMBS-by-1 vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an NMBS-by-1 vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `mbsyield` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an NMBS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `mbsyield` also accepts serial date numbers as inputs, but they are not recommended.

`IssueDate` — Issue date

datetime array | string array | date character vector

Issue date, specified as an NMBS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `mbsyield` also accepts serial date numbers as inputs, but they are not recommended.

`GrossRate` — Gross coupon rate (including fees)
vector of decimal values

Gross coupon rate (including fees), specified as an NMBS-by-1 vector of decimal values.

Data Types: `double`

`CouponRate` — Net coupon rate
`GrossRate` (default) | vector of decimal values

(Optional) Net coupon rate, specified as an NMBS-by-1 vector of decimal values.

Data Types: `double`

`Delay` — Delay (in days) between payment from homeowner and receipt by bondholder
`0` (no delay between payment and receipt) (default) | vector

(Optional) Delay (in days) between payment from homeowner and receipt by bondholder, specified as an NMBS-by-1 vector.

Data Types: `double`

`PrepaySpeed` — Speed relative to PSA standard
`0` (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an NMBS-by-1 vector. The PSA standard is `100`.

Note Set the `PrepaySpeed` to `[]` if you input a customized `PrepayMatrix`.

Data Types: `double`

`PrepayMatrix` — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a NaN-padded matrix of size `max(TermRemaining)`-by-NMBS. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

Output Arguments

`MYield` — Yield to maturity of the mortgage-backed security
vector

Yield to maturity of the mortgage-backed security, returned as a `NMBS`-by-`1` vector. This yield is compounded monthly (12 times a year).

`BEMBSYield` — Bond equivalent yield of the mortgage-backed security
vector

Bond equivalent yield of the mortgage-backed security, returned as a `NMBS`-by-`1` vector. This yield is compounded semiannually (two times a year).

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `mbsyield` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] *PSA Uniform Practices*, SF-49

See Also

`mbsprice`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsyield2oas

Option-adjusted spread given yield

Syntax

```
OAS = mbsyield2oas(ZeroCurve,Yield,Settle,Maturity,IssueDate,GrossRate)
OAS = mbsyield2oas( ___ ,CouponRate,Delay,Interpolation,PrepaySpeed,
PrepayMatrix)
```

Description

`OAS = mbsyield2oas(ZeroCurve,Yield,Settle,Maturity,IssueDate,GrossRate)` computes option-adjusted spread in basis points.

`OAS = mbsyield2oas( ___ ,CouponRate,Delay,Interpolation,PrepaySpeed, PrepayMatrix)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Calculate the Option-Adjusted Spread of a 30-Year Fixed-Rate Mortgage Pool

Calculate the option-adjusted spread of a 30-year, fixed-rate mortgage pool with about 28-year weighted average maturity left, given assumptions of 0, 50, and 100 PSA prepayments. First, create the bonds matrix:

```
Bonds = [datenum('11/21/2002')  0        100  0  2  1;
         datenum('02/20/2003')  0        100  0  2  1;
         datenum('07/31/2004')  0.03     100  2  3  1;
         datenum('08/15/2007')  0.035    100  2  3  1;
         datenum('08/15/2012')  0.04875  100  2  3  1;
         datenum('02/15/2031')  0.05375  100  2  3  1];
```

Choose a settlement date.

```
Settle = datenum('20-Aug-2002');
```

Assume the following clean prices for the bonds:

```
Prices =  [ 98.97467;
            98.58044;
           100.10534;
            98.18054;
           101.38136;
            99.25411];
```

Use the following formula to compute spot compounding for the bonds:

```
SpotCompounding = 2*ones(size(Prices));
```

Compute the zero curve.

```
[ZeroRatesP, CurveDatesP] = zbtprice(Bonds, Prices, Settle);
ZeroCurve = [CurveDatesP, ZeroRatesP, SpotCompounding]
```

```
ZeroCurve = 6×3
10^5 ×

    7.3154    0.0000    0.0000
    7.3163    0.0000    0.0000
    7.3216    0.0000    0.0000
    7.3327    0.0000    0.0000
    7.3510    0.0000    0.0000
    7.4185    0.0000    0.0000
```

Assign the following parameters:

```
Price         = 95;
Maturity      = datenum('02-Jan-2030');
IssueDate     = datenum('02-Jan-2000');
GrossRate     = 0.08125;
CouponRate    = 0.075;
Delay         = 14;
Interpolation = 1;
PrepaySpeed   = [0 50 100];
```

Compute the yield, and from the yield, compute the option-adjusted spread.

```
[mbsyld, beyld] = mbsyield(Price, Settle, ...
Maturity, IssueDate, GrossRate, CouponRate, Delay, PrepaySpeed);

OAS = mbsyield2oas(ZeroCurve, mbsyld, Settle, ...
Maturity, IssueDate, GrossRate, CouponRate, Delay, ...
Interpolation, PrepaySpeed)
```

```
OAS = 3×1

   26.0508
   28.6355
   31.2232
```

Input Arguments

`ZeroCurve` — Zero curve

matrix

Zero curve, specified as a three-column matrix, where:

- Column 1 is serial date numbers.
- Column 2 is spot rates with maturities corresponding to the dates in Column 1, in decimal (for example, 0.075).
- Column 3 is the compounding value of the rates in Column 2. (This is the agency spot rate on the settlement date.) Allowable compounding values are: 1 (annual), 2 (semiannual, 3 (three times per year), 4 (quarterly), 6 (bimonthly), 12 (monthly), and -1 (continuous).

For example:

```
[datenum('1-Jan-2003')  0.0154  12;
 datenum('1-Jan-2004')  0.0250  12;
 ......
 datenum('1-Jan-2020')  0.0675   2];
```

Data Types: `double` | `char` | `cell`

`Yield` — Mortgage yield, compounded monthly
vector in decimals

Mortgage yield, compounded monthly, specified as an `NMBS`-by-`1` vector in decimals.

Data Types: `double`

`Settle` — Settlement date
serial date number | date character vector

Settlement date, specified as an `NMBS`-by-`1` vector using serial date numbers or date character vectors. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char`

`Maturity` — Maturity date
serial date number | date character vector

Maturity date, specified as an `NMBS`-by-`1` vector using serial date numbers or date character vectors.

Data Types: `double` | `char`

`IssueDate` — Issue date
serial date number | date character vector

Issue date, specified as an `NMBS`-by-`1` vector using serial date numbers or date character vectors.

Data Types: `double` | `char`

`GrossRate` — Gross coupon rate (including fees)
vector of decimal values

Gross coupon rate (including fees), specified as an `NMBS`-by-`1` vector of decimal values.

Data Types: `double`

`CouponRate` — Net coupon rate
`GrossRate` (default) | vector of decimal values

(Optional) Net coupon rate, specified as an `NMBS`-by-`1` vector of decimal values.

Data Types: `double`

`Delay` — Delay (in days) between payment from homeowner and receipt by bondholder
`0` (no delay between payment and receipt) (default) | vector

(Optional) Delay (in days) between payment from homeowner and receipt by bondholder, specified as an `NMBS`-by-`1` vector.

Data Types: `double`

`Interpolation` — Interpolation method to compute the corresponding spot rates for the bond's cash flow
`1` (linear) (default) | vector

(Optional) Interpolation method to compute the corresponding spot rates for the bond's cash flow, specified as an `NMBS`-by-`1` vector. Available methods are (`0`) nearest, (`1`) linear, and (`2`) cubic spline. For more information on the supported interpolation methods, see `interp1`.

Data Types: `double`

`PrepaySpeed` — Speed relative to PSA standard
`0` (no prepayment) (default) | vector

(Optional) Speed relative to PSA standard, specified as an `NMBS`-by-`1` vector. The PSA standard is `100`.

Note Set the `PrepaySpeed` to `[]` if you input a customized `PrepayMatrix`.

Data Types: `double`

`PrepayMatrix` — Customized prepayment vector
matrix

(Optional) Customized prepayment vector, specified as a `NaN`-padded matrix of size `max(TermRemaining)`-by-`NMBS`. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

Output Arguments

`OAS` — Zero volatility OAS
vector

Zero volatility OAS, in basis point (bp), returned as a `NMBS`-by-`1` vector.

Version History
Introduced before R2006a

References

[1] *PSA Uniform Practices*, SF-49

See Also
`mbsoas2price` | `mbsoas2yield` | `mbsprice2oas`

Topics
“Generating Prepayment Vectors” on page 5-4

“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

mbsyield2speed

Implied PSA prepayment speeds given yield

Syntax

```
[ImpSpdOnPrc,ImpSpdOnDur,ImpSpdOnCnv] = mbsyield2speed(Yield,Settle,Maturity,
IssueDate,GrossRate,PrepayMatrix)
[ImpSpdOnPrc,ImpSpdOnDur,ImpSpdOnCnv] = mbsyield2speed( ___ ,CouponRate,Delay)
```

Description

`[ImpSpdOnPrc,ImpSpdOnDur,ImpSpdOnCnv] = mbsyield2speed(Yield,Settle,Maturity, IssueDate,GrossRate,PrepayMatrix)` computes PSA prepayment speeds implied by pool yields and projected (user-defined) prepayment vectors. The calculated PSA speed produces the same yield, modified duration, or modified convexity, depending upon the output requested.

`[ImpSpdOnPrc,ImpSpdOnDur,ImpSpdOnCnv] = mbsyield2speed( ___ ,CouponRate,Delay)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Compute PSA Prepayment Speeds

Calculate the equivalent PSA benchmark prepayment speeds for a security with these characteristics and prepayment matrix.

```
Yield        = 0.065;
Settle       = datetime(2000,1,1);
Maturity     = datetime(2030,1,1);
IssueDate    = datetime(2000,1,1);
GrossRate    = 0.08125;
PrepayMatrix = 0.005*ones(360,1);
CouponRate   = 0.075;
Delay        = 14;

[ImpSpdOnYld, ImpSpdOnDur, ImpSpdOnCnv] = ...
mbsyield2speed(Yield, Settle, Maturity, IssueDate, GrossRate, ...
PrepayMatrix, CouponRate, Delay)

ImpSpdOnYld = 117.7644

ImpSpdOnDur = 116.7436

ImpSpdOnCnv = 108.3309
```

Input Arguments

`Yield` — Mortgage yield, compounded monthly

vector in decimals

Mortgage yield, compounded monthly, specified as an `NMBS`-by-`1` vector in decimals.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NMBS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `mbsyield2speed` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NMBS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `mbsyield2speed` also accepts serial date numbers as inputs, but they are not recommended.

`IssueDate` — Issue date

datetime array | string array | date character vector

Issue date, specified as an `NMBS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `mbsyield2speed` also accepts serial date numbers as inputs, but they are not recommended.

`GrossRate` — Gross coupon rate (including fees)

vector of decimal values

Gross coupon rate (including fees), specified as an `NMBS`-by-`1` vector of decimal values.

Data Types: `double`

`PrepayMatrix` — Customized prepayment vector

matrix

Customized prepayment vector, specified as a `NaN`-padded matrix of size `max(TermRemaining)`-by-`NMBS`. Each column corresponds to each mortgage-backed security, and each row corresponds to each month after settlement.

Note Use `PrepayMatrix` only when `PrepaySpeed` is unspecified.

Data Types: `double`

`CouponRate` — Net coupon rate
`GrossRate` (default) | vector of decimal values

(Optional) Net coupon rate, specified as an `NMBS`-by-`1` vector of decimal values.

Data Types: `double`

`Delay` — Delay (in days) between payment from homeowner and receipt by bondholder
`0` (no delay between payment and receipt) (default) | vector

(Optional) Delay (in days) between payment from homeowner and receipt by bondholder, specified as an `NMBS`-by-`1` vector.

Data Types: `double`

Output Arguments

`ImpSpdOnPrc` — Equivalent PSA benchmark prepayment speed for the pass-through to carry the same price
vector

Equivalent PSA benchmark prepayment speed for the pass-through to carry the same price, returned as a `NMBS`-by-`1` vector.

`ImpSpdOnDur` — Equivalent PSA benchmark prepayment speed for the pass-through to carry the same modified duration
vector

Equivalent PSA benchmark prepayment speed for the pass-through to carry the same modified duration, returned as a `NMBS`-by-`1` vector.

`ImpSpdOnCnv` — Equivalent PSA benchmark prepayment speed for the pass-through to carry the same modified convexity
vector

Equivalent PSA benchmark prepayment speed for the pass-through to carry the same modified convexity, returned as a `NMBS`-by-`1` vector.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `mbsyield2speed` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] *PSA Uniform Practices,* SF-49

See Also

`mbsyield` | `mbsprice2speed`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

psaspeed2default

Benchmark default

Syntax

```
[ADRPSA,MDRPSA] = psaspeed2default(DefaultSpeed)
```

Description

`[ADRPSA,MDRPSA] = psaspeed2default(DefaultSpeed)` computes the benchmark default on the performing balance of mortgage-backed securities per PSA benchmark speed.

Examples

Compute the Benchmark Default Rates on the Performing Balance of Mortgage-Backed Securities Per PSA Benchmark Speed

This example shows how to compute the benchmark default rates on the performing balance of mortgage-backed securities per PSA benchmark speed, given a mortgage-backed security with annual speed set at the PSA default benchmark.

```
DefaultSpeed = 100;

[ADRPSA, MDRPSA] = psaspeed2default(DefaultSpeed)
```

```
ADRPSA = 360×1

    0.0002
    0.0004
    0.0006
    0.0008
    0.0010
    0.0012
    0.0014
    0.0016
    0.0018
    0.0020
      ⋮


MDRPSA = 360×1
10^-3 ×

    0.0167
    0.0333
    0.0500
    0.0667
    0.0834
    0.1001
    0.1167
    0.1334
```

```
    0.1501
    0.1668
      ⋮
```

Input Arguments

`DefaultSpeed` — Annual speed relative to the benchmark

vector in decimals

Annual speed relative to the benchmark, specified as an `NDEF`-by-`1` vector. The PSA benchmark is `100`.

Data Types: `double`

Output Arguments

`ADRPSA` — PSA default rate

vector

PSA default rate, returned as a `360`-by-`NDEF` vector in decimals.

`MDRPSA` — PSA monthly default rate

vector

PSA monthly default rate, returned as a `360`-by-`NDEF` vector in decimals.

Version History

Introduced before R2006a

References

[1] *PSA Uniform Practices*, SF-49

See Also

`psaspeed2rate`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

psaspeed2rate

Single monthly mortality rate given PSA speed

Syntax

```
[CPRPSA,SMMPSA] = psaspeed2rate(PSASpeed)
```

Description

`[CPRPSA,SMMPSA] = psaspeed2rate(PSASpeed)` calculates vectors of PSA prepayments, each containing 360 prepayment elements, to represent the 360 months in a 30-year mortgage pool.

Examples

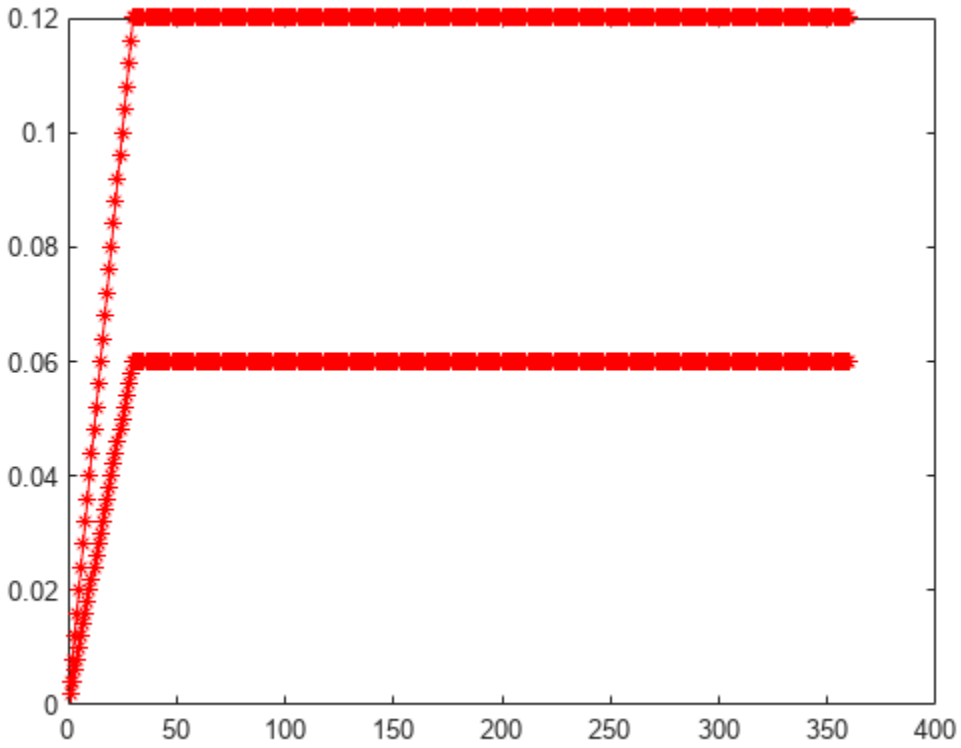
Compute the Prepayment and Mortality Rates

This example shows how to compute the prepayment and mortality rates, given a mortgage-backed security with annual speed set at the PSA default benchmark.

```
PSASpeed = [100 200];

[CPRPSA, SMMPSA]= psaspeed2rate(PSASpeed)
```

```
CPRPSA = 360×2

    0.0020    0.0040
    0.0040    0.0080
    0.0060    0.0120
    0.0080    0.0160
    0.0100    0.0200
    0.0120    0.0240
    0.0140    0.0280
    0.0160    0.0320
    0.0180    0.0360
    0.0200    0.0400
      ⋮


SMMPSA = 360×2

    0.0002    0.0003
    0.0003    0.0007
    0.0005    0.0010
    0.0007    0.0013
    0.0008    0.0017
    0.0010    0.0020
    0.0012    0.0024
    0.0013    0.0027
    0.0015    0.0031
    0.0017    0.0034
      ⋮
```

```
% view the plot of the output
psaspeed2rate(PSASpeed)
```



Input Arguments

PSASpeed — Annual speed relative to the benchmark

vector in decimals

Annual speed relative to the benchmark, specified as any value > `0` using an `NSPD`-by-`1` vector. The PSA benchmark is `100`.

Data Types: `double`

Output Arguments

CPRPSA — PSA conditional prepayment rate

vector

PSA conditional prepayment rate, returned as a `360`-by-`NSPD` vector in decimals.

SMMPSA — PSA single monthly mortality rate

vector

PSA monthly default rate, returned as a `360`-by-`NSPD` vector in decimals.

Version History

Introduced before R2006a

References

[1] *PSA Uniform Practices*, SF-49

See Also

`psaspeed2default`

Topics

“Generating Prepayment Vectors” on page 5-4
“Mortgage Prepayments” on page 5-5
“Prepayment Modeling with a Two Factor Hull White Model and a LIBOR Market Model” on page 5-16
“What Are Mortgage-Backed Securities?” on page 5-2

stepcpncfamounts

Cash flow amounts and times for bonds and stepped coupons

Syntax

```
[CFlows,CDates,CTimes] = stepcpncfamounts(Settle,Maturity,ConvDates,
CouponRates)
[CFlows,CDates,CTimes] = stepcpncfamounts( ___ ,Period,Basis,EndMonthRule,Face)
```

Description

`[CFlows,CDates,CTimes] = stepcpncfamounts(Settle,Maturity,ConvDates, CouponRates)` returns matrices of cash flow amounts, cash flow dates, and time factors for a portfolio of `NUMBONDS` stepped-coupon bonds.

`[CFlows,CDates,CTimes] = stepcpncfamounts( ___ ,Period,Basis,EndMonthRule,Face)` adds additional optional arguments.

Examples

Generate Stepped Cash Flows for Three Different Bonds

This example generates stepped cash flows for three different bonds, all paying interest semiannually. The life span of the bonds is about 18–19 years each:

- Bond A has two conversions, but the first one occurs on the settlement date and immediately expires.
- Bond B has three conversions, with conversion dates exactly on the coupon dates.
- Bond C has three conversions, with some conversion dates not on the coupon dates. It has the longest maturity. This case illustrates that only cash flows for full periods after conversion dates are affected, as illustrated here:

The following table illustrates the interest-rate characteristics of this bond portfolio.

Bond A Dates	Bond A Rates	Bond B Dates	Bond B Rates	Bond C Dates	Bond C Rates
Settle (02-Aug-92)	7.5%	Settle (02-Aug-92)	7.5%	Settle (02-Aug-92)	2.5%
First Conversion (02-Aug-92)	8.875%	First Conversion (15-Jun-97))	8.875%	First Conversion (14-Jun-97))	5.0%
Second Conversion (15-Jun-03)	9.25%	Second Conversion (15-Jun-01)	9.25%	Second Conversion (14-Jun-01)	7.5%
Maturity (15-Jun-10)	NaN	Third Conversion (15-Jun-05)	10.0%	Third Conversion (14-Jun-05)	10.0%
		Maturity (15-Jun-10)	NaN	Maturity (15-Jun-11)	NaN

Define the bond specifications.

```
Settle   = datenum('02-Aug-1992');

ConvDates = [datenum('02-Aug-1992'), datenum('15-Jun-2003'),...
             nan;
            datenum('15-Jun-1997'), datenum('15-Jun-2001'),...
            datenum('15-Jun-2005');
            datenum('14-Jun-1997'), datenum('14-Jun-2001'),...
            datenum('14-Jun-2005')];

Maturity = [datenum('15-Jun-2010');
            datenum('15-Jun-2010');
            datenum('15-Jun-2011')];

CouponRates = [0.075 0.08875 0.0925 nan;
               0.075 0.08875 0.0925 0.1;
               0.025 0.05    0.0750 0.1];
Basis = 1;
Period = 2;
EndMonthRule = 1;
Face = 100;
```

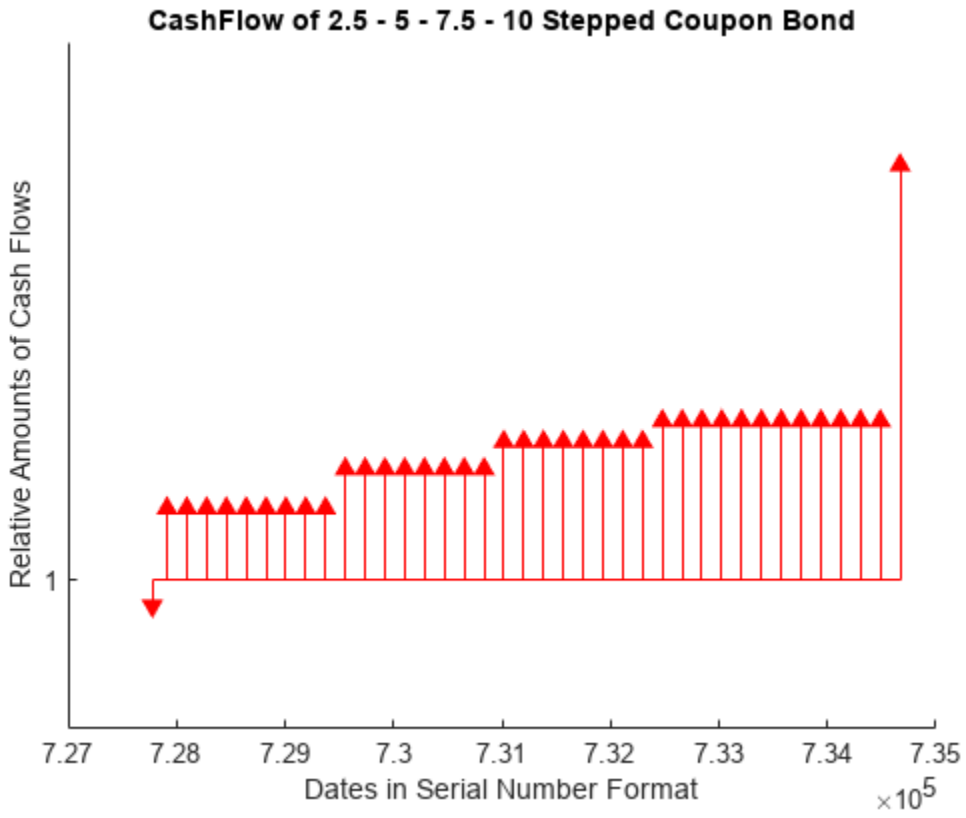
Use `stepcpncfamounts` to compute cash flows and timings.

```
[CFlows, CDates, CTimes] = stepcpncfamounts(Settle, Maturity, ConvDates, CouponRates)
```

```
CFlows = 3×39

   -1.1639    4.4375    4.4375    4.4375    4.4375    4.4375    4.4375    4.4375    4.4375    4.4
   -0.9836    3.7500    3.7500    3.7500    3.7500    3.7500    3.7500    3.7500    3.7500    3.7
   -0.3279    1.2500    1.2500    1.2500    1.2500    1.2500    1.2500    1.2500    1.2500    1.2


CDates = 3×39

      727778      727913      728095      728278      728460      728643      728825      729008
      727778      727913      728095      728278      728460      728643      728825      729008
      727778      727913      728095      728278      728460      728643      728825      729008


CTimes = 3×39

         0    0.7377    1.7377    2.7377    3.7377    4.7377    5.7377    6.7377    7.7377    8.7
         0    0.7377    1.7377    2.7377    3.7377    4.7377    5.7377    6.7377    7.7377    8.7
         0    0.7377    1.7377    2.7377    3.7377    4.7377    5.7377    6.7377    7.7377    8.7
```

Visualize the third bond's cash flows (2.5 - 5 - 7.5 - 10) using the `cfplot` function.

```
cfplot(CDates(3,:),CFlows(3,:));
xlabel('Dates in Serial Number Format')
ylabel('Relative Amounts of Cash Flows')
title('CashFlow of 2.5 - 5 - 7.5 - 10 Stepped Coupon Bond')
```



Input Arguments

`Settle` — Settlement date

serial date number | date character vector

Settlement date, specified either as a scalar or `NSTP`-by-`1` vector using serial date numbers or date character vectors.

`Settle` must be earlier than `Maturity`.

Data Types: `double` | `char`

`Maturity` — Maturity date

serial date number | date character vector

Maturity date, specified as a scalar or an `NSTP`-by-`1` vector using serial date numbers or date character vectors that represent the maturity date for each bond.

Data Types: `double` | `char` | `string` | `datetime`

`ConvDates` — Conversion dates

serial date number | date character vector

Conversion dates, specified as a `NSTP`-by-max(`NCONV`) matrix using serial date numbers or date character vectors containing conversion dates after `Settle`. The size of the matrix is equal to the number of instruments by the maximum number of conversions. Fill unspecified entries with `NaN`.

Data Types: `double` | `char`

`CouponRates` — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an `NSTP`-by-max(`NCONV+1`) matrix containing coupon rates for each bond in the portfolio in decimal form. The matrix size is equal to the number of instruments by maximum number of conversions + 1. First column of this matrix contains rates applicable between `Settle` and the first conversion date (date in the first column of `ConvDates`). Fill unspecified entries with `NaN`

`ConvDates` has the same number of rows as `CouponRates` to reflect the same number of bonds. However, `ConvDates` has one less column than `CouponRates`. This situation is illustrated by

```
Settle---------ConvDate1-----------ConvDate2------------Maturity

        Rate1               Rate2                Rate3
```

Data Types: `double`

`Period` — Coupons per year
2 per year (default) | vector

(Optional) Coupons per year, specified as an `NSTP`-by-`1` vector. Values for `Period` are `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis of each instrument, specified as an `NSTP`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

EndMonthRule — End-of-month rule flag for generating dates when Maturity is end-of-month date for month having 30 or fewer days
1 (in effect) (default) | nonnegative integer [0,1]

(Optional) End-of-month rule flag for generating dates when Maturity is an end-of-month date for a month having 30 or fewer days, specified for each bond as a nonnegative integer [0, 1] using a NSTP-by-1 vector.

- 0 = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- 1 = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: logical

Face — Face value
100 (default) | vector of nonnegative values

(Optional) Face value, specified for each bond as an NSTP-by-1 vector of nonnegative face values.

Data Types: double

Output Arguments

CFlows — Cash flow amounts
vector

Cash flow amounts, returned as a vector where the first entry in each row vector is the (negative) accrued interest due at settlement. If no accrued interest is due, the first column is zero.

CDates — Cash flow dates
vector

Cash flow dates, returned as vector in serial date number form. At least two columns are always present: one for settlement and one for maturity.

CTimes — Time factor
vector

Time factor for the SIA semiannual price/yield conversion, returned as a vector. The SIA semi-annual price/yield conversion is DiscountFactor = (1 + Yield/2).^(-TFactor). Time factors are in units of semiannual coupon periods. For ISMA conventions: DiscountFactor = (1 + Yield).^(-TFactor). Time factors are in units of annual coupon periods. In computing time factors, use SIA actual/actual conventions for all time factor calculations.

Note For bonds with fixed coupons, use cfamounts. If you use a fixed-coupon bond with stepcpncfamounts, MATLAB software generates an error.

Version History

Introduced before R2006a

See Also

stepcpnprice | stepcpnyield | cfplot

Topics

“Cash Flows from Stepped-Coupon Bonds” on page 6-8
“Price and Yield of Stepped-Coupon Bonds” on page 6-9
“Managing Present Value with Bond Futures” on page 7-14

stepcpnprice

Price bond with stepped coupons

Syntax

```
[Price,AccruedInterest] = stepcpnprice(Yield,Settle,Maturity,ConvDates,
CouponRates)
[Price,AccruedInterest] = stepcpnprice( ___ ,Period,Basis,EndMonthRule,Face)
```

Description

`[Price,AccruedInterest] = stepcpnprice(Yield,Settle,Maturity,ConvDates, CouponRates)` computes the price of bonds with stepped coupons given the yield to maturity. The function supports any number of conversion dates.

`[Price,AccruedInterest] = stepcpnprice( ___ ,Period,Basis,EndMonthRule,Face)` adds additional optional arguments.

Examples

Compute Bond Prices with Stepped Coupons

Compute the price and accrued interest due on a portfolio of stepped-coupon bonds having a yield of 7.221%, given three conversion scenarios:

- Bond A has two conversions, the first one falling on the settle date and immediately expiring.
- Bond B has three conversions, with conversion dates exactly on the coupon dates.
- Bond C has three conversions, with one or more conversion dates not on coupon dates. This case illustrates that only cash flows for full periods after conversion dates are affected, as illustrated below:

The following table illustrates the interest-rate characteristics of this bond portfolio.

Bond A Dates	Bond A Rates	Bond B Dates	Bond B Rates	Bond C Dates	Bond C Rates
Settle (02-Aug-92)	7.5%	Settle (02-Aug-92)	7.5%	Settle (02-Aug-92)	7.5%
First Conversion (02-Aug-92)	8.875%	First Conversion (15-Jun-97)	8.875%	First Conversion (14-Jun-97)	8.875%
Second Conversion (15-Jun-03)	9.25%	Second Conversion (15-Jun-01)	9.25%	Second Conversion (14-Jun-01)	9.25%
Maturity (15-Jun-10)	NaN	Third Conversion (15-Jun-05)	10.0%	Third Conversion (14-Jun-05)	10.0%
		Maturity (15-Jun-10)	NaN	Maturity (15-Jun-10)	NaN

Define the specifications for the bonds.

```
Yield = 0.07221;
Settle   = datenum('02-Aug-1992');
```

```
ConvDates = [datenum('02-Aug-1992'), datenum('15-Jun-2003'),...
             nan;
            datenum('15-Jun-1997'), datenum('15-Jun-2001'),...
            datenum('15-Jun-2005');
            datenum('14-Jun-1997'), datenum('14-Jun-2001'),...
            datenum('14-Jun-2005')];
Maturity = datenum('15-Jun-2010');

CouponRates = [0.075 0.08875 0.0925 nan;
               0.075 0.08875 0.0925 0.1;
               0.075 0.08875 0.0925 0.1];
Basis = 1;
Period = 2;
EndMonthRule = 1;
Face = 100;
```

Use `stepcpnprice` to compute the bond prices with stepped coupons.

```
[Price, AccruedInterest] = stepcpnprice(Yield, Settle, Maturity, ConvDates, CouponRates, Period,
```

```
Price = 3×1

  117.3874
  113.4387
  114.1759


AccruedInterest = 3×1

    1.1587
    0.9792
    0.9792
```

Input Arguments

`Yield` — Yield to maturity

numeric

Yield to maturity, specified as a scalar or `NUMBONDS`-by-1 vector of numeric values.

Data Types: `double`

`Settle` — Settlement date

serial date number | date character vector

Settlement date, specified either as a scalar or `NUMBONDS`-by-1 vector using serial date numbers or date character vectors.

`Settle` must be earlier than `Maturity`.

Data Types: `double` | `char`

`Maturity` — Maturity date

serial date number | date character vector

Maturity date, specified as a scalar or an NUMBONDS-by-1 vector using serial date numbers or date character vectors that represent the maturity date for each bond.

Data Types: double | char

ConvDates — Conversion dates
serial date number | date character vector

Conversion dates, specified as a NSTP-by-max(NCONV) matrix using serial date numbers or date character vectors that contain conversion dates after Settle. The size of the matrix is equal to the number of instruments by the maximum number of conversions. Fill unspecified entries with NaN.

Data Types: double | char

CouponRates — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an NSTP-by-max(NCONV+1) matrix containing coupon rates for each bond in the portfolio in decimal form. The matrix size is equal to the number of instruments by maximum number of conversions + 1. First column of this matrix contains rates applicable between Settle and the first conversion date (date in the first column of ConvDates). Fill unspecified entries with NaN

ConvDates has the same number of rows as CouponRates to reflect the same number of bonds. However, ConvDates has one less column than CouponRates. This situation is illustrated by

```
Settle---------ConvDate1-----------ConvDate2------------Maturity

        Rate1               Rate2                Rate3
```

Data Types: double

Period — Coupons per year
2 per year (default) | vector

(Optional) Coupons per year, specified as an NUMBONDS-by-1 vector. Values for Period are 1, 2, 3, 4, 6, and 12.

Data Types: double

Basis — Day-count basis
0 (actual/actual) (default) | integer from 0 to 13

(Optional) Day-count basis of each instrument, specified as an NUMBONDS-by-1 vector.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

(Optional) End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified for each bond as a nonnegative integer [`0`, `1`] using a `NUMBONDS`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Face — Face value
`100` (default) | vector of nonnegative values

(Optional) Face value, specified for each bond as an `NUMBONDS`-by-`1` vector of nonnegative face values.

Data Types: `double`

Output Arguments

`Price` — Clean price
vector

Clean price, returned as a `NUMBONDS`-by-`1` vector.

Note For bonds with fixed coupons, use `bndprice`. If you use a fixed-coupon bond with `stepcpnprice`, you receive the error: `incorrect number of inputs`.

`AccruedInterest` — Accrued interest payable at settlement dates
vector

accrued interest payable at settlement dates, returned as a `NUMBONDS`-by-`1` vector.

Version History

Introduced before R2006a

See Also

`stepcpnyield` | `tbillprice` | `stepcpncfamounts` | `cdprice` | `bndprice`

Topics

“Cash Flows from Stepped-Coupon Bonds” on page 6-8
“Price and Yield of Stepped-Coupon Bonds” on page 6-9
“Managing Present Value with Bond Futures” on page 7-14

stepcpnyield

Yield to maturity of bond with stepped coupons

Syntax

```
Yield = stepcpnyield(Price,Settle,Maturity,ConvDates,CouponRates)
Yield = stepcpnyield( ___ ,Period,Basis,EndMonthRule,Face)
```

Description

`Yield = stepcpnyield(Price,Settle,Maturity,ConvDates,CouponRates)` computes the yield to maturity of bonds with stepped coupons given the price. The function supports any number of conversion dates.

`Yield = stepcpnyield( ___ ,Period,Basis,EndMonthRule,Face)` adds additional optional arguments.

Examples

Compute Yield to Maturity for Bonds with Stepped Coupons

Find the yield to maturity of three stepped-coupon bonds of known price, given three conversion scenarios:

- Bond A has two conversions, the first one falling on the settle date and immediately expiring.
- Bond B has three conversions, with conversion dates exactly on the coupon dates.
- Bond C has three conversions, with one or more conversion dates not on coupon dates. This case illustrates that only cash flows for full periods after conversion dates are affected, as illustrated below.

The following table illustrates the interest-rate characteristics of this bond portfolio.

Bond A Dates	Bond A Rates	Bond B Dates	Bond B Rates	Bond C Dates	Bond C Rates
Settle (02-Aug-92)	7.5%	Settle (02-Aug-92)	7.5%	Settle (02-Aug-92)	7.5%
First Conversion (02-Aug-92)	8.875%	First Conversion (15-Jun-97)	8.875%	First Conversion (14-Jun-97)	8.875%
Second Conversion (15-Jun-03)	9.25%	Second Conversion (15-Jun-01)	9.25%	Second Conversion (14-Jun-01)	9.25%
Maturity (15-Jun-10)	NaN	Third Conversion (15-Jun-05)	10.0%	Third Conversion (14-Jun-05)	10.0%
		Maturity (15-Jun-10)	NaN	Maturity (15-Jun-10)	NaN

Define the specifications for the bonds.

```
format long
Price = [117.3824; 113.4339; 113.4339];
```

```
Settle = datenum('02-Aug-1992');

ConvDates = [datenum('02-Aug-1992'), datenum('15-Jun-2003'), nan;
datenum('15-Jun-1997'), datenum('15-Jun-2001'), datenum('15-Jun-2005');
datenum('14-Jun-1997'), datenum('14-Jun-2001'), datenum('14-Jun-2005')];

Maturity = datenum('15-Jun-2010');

CouponRates = [0.075 0.08875 0.0925 nan;
               0.075 0.08875 0.0925 0.1;
               0.075 0.08875 0.0925 0.1];
Basis = 1;
Period = 2;
EndMonthRule = 1;
Face = 100;
```

Use `stepcpnyield` to compute the yield to maturity for the bonds with stepped coupons.

```
Yield = stepcpnyield(Price, Settle, Maturity, ConvDates, CouponRates, Period, Basis, EndMonthRule
```

```
Yield = 3×1

   0.072214402049150
   0.072214267800360
   0.072864799557221
```

Input Arguments

`Price` — Price of bond

numeric

Price of bond, specified as a scalar or `NUMBONDS`-by-1 vector of numeric values.

Data Types: `double`

`Settle` — Settlement date

serial date number | date character vector

Settlement date, specified either as a scalar or `NUMBONDS`-by-1 vector using serial date numbers or date character vectors.

`Settle` must be earlier than `Maturity`.

Data Types: `double` | `char`

`Maturity` — Maturity date

serial date number | date character vector

Maturity date, specified as a scalar or an `NUMBONDS`-by-1 vector using serial date numbers or date character vectors that represent the maturity date for each bond.

Data Types: `double` | `char`

`ConvDates` — Conversion dates

serial date number | date character vector

Conversion dates, specified as a NSTP-by-max(NCONV) matrix using serial date numbers or date character vectors that contain conversion dates after `Settle`. The size of the matrix is equal to the number of instruments by the maximum number of conversions. Fill unspecified entries with `NaN`.

Data Types: `double` | `char`

`CouponRates` — Bond coupon rate
positive decimal value

Bond coupon rate, specified as an NSTP-by-max(NCONV+1) matrix containing coupon rates for each bond in the portfolio in decimal form. The matrix size is equal to the number of instruments by maximum number of conversions + 1. First column of this matrix contains rates applicable between `Settle` and the first conversion date (date in the first column of `ConvDates`). Fill unspecified entries with `NaN`

`ConvDates` has the same number of rows as `CouponRates` to reflect the same number of bonds. However, `ConvDates` has one less column than `CouponRates`. This situation is illustrated by

```
Settle---------ConvDate1-----------ConvDate2------------Maturity

        Rate1              Rate2                Rate3
```

Data Types: `double`

`Period` — Coupons per year
2 per year (default) | vector

(Optional) Coupons per year, specified as an NUMBONDS-by-1 vector. Values for `Period` are `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Day-count basis of each instrument, specified as an NUMBONDS-by-1 vector.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

(Optional) End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, specified for each bond as a nonnegative integer [`0`, `1`] using a `NUMBONDS`-by-`1` vector.

- `0` = Ignore rule, meaning that a payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Face — Face value
`100` (default) | vector of nonnegative values

(Optional) Face value, specified for each bond as an `NUMBONDS`-by-`1` vector of nonnegative face values.

Data Types: `double`

Output Arguments

`Yield` — Yield to maturity
vector

Yield to maturity, returned as a `NUMBONDS`-by-`1` vector in decimal form.

Note For bonds with fixed coupons, use `bndyield`. You receive the error `incorrect number of inputs` if you use a fixed-coupon bond with `stepcpnyield`.

Version History

Introduced before R2006a

See Also

`stepcpnprice` | `bndprice` | `cdprice` | `stepcpncfamounts` | `stepcpnprice` | `tbillprice` | `zeroprice`

Topics
“Cash Flows from Stepped-Coupon Bonds” on page 6-8
“Price and Yield of Stepped-Coupon Bonds” on page 6-9
“Managing Present Value with Bond Futures” on page 7-14

tfutbyprice

Future prices of Treasury bonds given spot price

Syntax

```
[QtdFutPrice,AccrInt] = tfutbyprice(SpotCurve,Price,SettleFut,MatFut,
ConvFactor,CouponRate,Maturity)
[QtdFutPrice,AccrInt] = tfutbyprice( ___ ,Interpolation)
```

Description

`[QtdFutPrice,AccrInt] = tfutbyprice(SpotCurve,Price,SettleFut,MatFut, ConvFactor,CouponRate,Maturity)` computes future prices of Treasury notes and bonds given the spot price.

In addition, you can use the Financial Instruments Toolbox method `getZeroRates` for an `IRDataCurve` object with a `Dates` property to create a vector of dates and data acceptable for `tfutbyprice`. For more information, see “Converting an IRDataCurve or IRFunctionCurve Object” on page 9-30.

Note Alternatively, you can use the `BondFuture` object to price bond future instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`[QtdFutPrice,AccrInt] = tfutbyprice( ___ ,Interpolation)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Determine the Future Prices of Treasury Bonds Given the Spot Price

This example shows how to determine the future price of two Treasury bonds based upon a spot rate curve constructed from data for November 14, 2002.

```
% construct spot curve from Nov 14, data
Bonds = [datenum('02/13/2003'),        0;
         datenum('05/15/2003'),        0;
         datenum('10/31/2004'),  0.02125;
         datenum('11/15/2007'),     0.03;
         datenum('11/15/2012'),     0.04;
         datenum('02/15/2031'),  0.05375];

Yields  = [1.20; 1.25; 1.86; 2.99; 4.02; 4.93]/100;

Settle = datenum('11/15/2002');

[ZeroRates, CurveDates] = ...
zbtyield(Bonds, Yields, Settle);
```

```
SpotCurve  = [CurveDates, ZeroRates];

% calculate a particular bond's future quoted price
RefDate    = [datenum('1-Dec-2002'); datenum('1-Mar-2003')];
MatFut     = [datenum('15-Dec-2002'); datenum('15-Mar-2003')];
Maturity   = [datenum('15-Aug-2009');datenum('15-Aug-2010')];
CouponRate = [0.06;0.0575];
ConvFactor = convfactor(RefDate, Maturity, CouponRate);
Price = [114.416; 113.171];
Interpolation = 1;

[QtdFutPrice, AccrInt] = tfutbyprice(SpotCurve, Price, Settle, ...
MatFut, ConvFactor, CouponRate, Maturity, Interpolation)
```

```
QtdFutPrice = 2×1

  114.0409
  113.4029


AccrInt = 2×1

    1.9891
    0.4448
```

Input Arguments

SpotCurve — Treasury spot curve

matrix

Treasury spot curve, specified as a number of futures using one of the following forms:

- NFUT-by-2 matrix in the form of `[SpotDates SpotRates]` and these spot rates must be quoted as semiannual compounding (2) when the third column is not supplied.
- NFUT-by-3 matrix in the form of `[SpotDates SpotRates Compounding]`, where allowed `Compounding` values for the third column are −1, 1, 2 (default), 3, 4, and 12, where −1 is continuous compounding.

Data Types: `double`

Price — Prices of Treasury bonds or notes per $100 notional at settlement date

scalar numeric | vector

Prices of Treasury bonds or notes per $100 notional at settlement date, specified as a scalar numeric or an NINST-by-1 vector. Use `bndprice` for theoretical value of bond.

Data Types: `double`

SettleFut — Settlement date of futures contract

serial date number | date character vector

Settlement date of futures contract, specified as a scalar or an NINST-by-1 vector using serial date numbers or date character vectors.

Data Types: `double` | `char`

`MatFut` — Maturity dates (or anticipated delivery dates) of futures contract
serial date number | date character vector

Maturity dates (or anticipated delivery dates) of futures contract, specified as a scalar or an `NINST`-by-`1` vector using serial date numbers or character vectors.

Data Types: `double` | `char`

`ConvFactor` — Conversion factor
numeric

Conversion factor, specified using `convfactor`.

Data Types: `double` | `char` | `cell`

`CouponRate` — Underlying bond annual coupon
scalar numeric in decimal | vector in decimals

Underlying bond annual coupon, specified as a scalar numeric decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Maturity` — Underlying bond maturity date
serial date number | date character vector

Underlying bond maturity date, specified as a scalar or an `NINST`-by-`1` vector using serial date numbers or date character vectors.

Data Types: `double` | `char`

`Interpolation` — Interpolation method to compute the corresponding spot rates for the bond's cash flow
`1` (linear) (default) | vector

(Optional) Interpolation method to compute the corresponding spot rates for the bond's cash flow, specified as an `NMBS`-by-`1` vector. Available methods are (`0`) nearest, (`1`) linear, and (`2`) cubic spline. For more information on the supported interpolation methods, see `interp1`.

Data Types: `double`

Output Arguments

`QtdFutPrice` — Quoted futures price, per $100 notional
vector

Quoted futures price, per $100 notional, returned as a `NINST`-by-`1` vector.

`AccrInt` — Accrued Interest due at delivery date, per $100 notional
vector

Accrued Interest due at delivery date, per $100 notional, returned as a `NINST`-by-`1` vector.

Version History

Introduced before R2006a

See Also

`convfactor` | `tfutbyyield` | `BondFuture`

Topics

“Computing Treasury Bill Price and Yield”
“Select Cheapest-to-Deliver Bond Using BondFuture Instrument” on page 2-212
“Treasury Bills Defined”

tfutbyyield

Future prices of Treasury bonds given current yield

Syntax

```
[QtdFutPrice,AccrInt] = tfutbyyield(SpotCurve,Yield,SettleFut,MatFut,
ConvFactor,CouponRate,Maturity)
[QtdFutPrice,AccrInt] = tfutbyyield( ___ ,Interpolation)
```

Description

`[QtdFutPrice,AccrInt] = tfutbyyield(SpotCurve,Yield,SettleFut,MatFut, ConvFactor,CouponRate,Maturity)` computes prices of Treasury bond futures given a spot curve and bond yields at settlement.

In addition, you can use the Financial Instruments Toolbox method `getZeroRates` for an `IRDataCurve` object with a `Dates` property to create a vector of dates and data acceptable for `tfutbyyield`. For more information, see “Converting an IRDataCurve or IRFunctionCurve Object” on page 9-30.

`[QtdFutPrice,AccrInt] = tfutbyyield( ___ ,Interpolation)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Determine Future Prices of Treasury Bonds Given the Current Yield

This example shows how to determine the future price of two Treasury bonds based upon a spot rate curve constructed from data for November 14, 2002.

```
% construct spot curve from Nov 14, data
Bonds = [datenum('02/13/2003'),        0;
         datenum('05/15/2003'),        0;
         datenum('10/31/2004'),  0.02125;
         datenum('11/15/2007'),     0.03;
         datenum('11/15/2012'),     0.04;
         datenum('02/15/2031'),  0.05375];

Yields  = [1.20; 1.25; 1.86; 2.99; 4.02; 4.93]/100;

Settle = datenum('11/15/2002');

[ZeroRates, CurveDates] = ...
zbtyield(Bonds, Yields, Settle);

SpotCurve  = [CurveDates, ZeroRates];

% calculate a particular bond's future quoted price
RefDate    = [datenum('1-Dec-2002'); datenum('1-Mar-2003')];
MatFut     = [datenum('15-Dec-2002'); datenum('15-Mar-2003')];
```

```
Maturity   = [datenum('15-Aug-2009');datenum('15-Aug-2010')];
CouponRate = [0.06;0.0575];
ConvFactor = convfactor(RefDate, Maturity, CouponRate);
Yield = [0.03576; 0.03773];
Interpolation = 1;

[QtdFutPrice, AccrInt] = tfutbyyield(SpotCurve, Yield, Settle, ...
MatFut, ConvFactor, CouponRate, Maturity, Interpolation)

QtdFutPrice = 2×1

  114.0416
  113.4034


AccrInt = 2×1

    1.9891
    0.4448
```

Input Arguments

SpotCurve — Treasury spot curve

matrix

Treasury spot curve, specified as a number of futures using one of the following forms:

- NFUT-by-2 matrix in the form of `[SpotDates SpotRates]` and these spot rates must be quoted as semiannual compounding (2) when the third column is not supplied.
- NFUT-by-3 matrix in the form of `[SpotDates SpotRates Compounding]`, where allowed `Compounding` values for the third column are −1, 1, 2 (default), 3, 4, and 12, where −1 is continuous compounding.

Data Types: `double`

Yield — Yield to maturities at settlement date

scalar numeric | vector

Yield to maturities at settlement date, specified as a scalar numeric or an NINST-by-1 vector. Use `bndyield` for theoretical value of bond yield.

Data Types: `double`

SettleFut — Settlement date of futures contract

serial date number | date character vector

Settlement date of futures contract, specified as a scalar or an NINST-by-1 vector using serial date numbers or date character vectors.

Data Types: `double` | `char`

MatFut — Maturity dates (or anticipated delivery dates) of futures contract

serial date number | date character vector

Maturity dates (or anticipated delivery dates) of futures contract, specified as a scalar or an `NINST`-by-`1` vector using serial date numbers or character vectors.

Data Types: `double` | `char`

`ConvFactor` — Conversion factor
numeric

Conversion factor, specified using `convfactor`.

Data Types: `double` | `char` | `cell`

`CouponRate` — Underlying bond annual coupon
scalar numeric in decimal | vector in decimals

Underlying bond annual coupon, specified as a scalar numeric decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Maturity` — Underlying bond maturity date
serial date number | date character vector

Underlying bond maturity date, specified as a scalar or an `NINST`-by-`1` vector using serial date numbers or date character vectors.

Data Types: `double` | `char`

`Interpolation` — Interpolation method to compute the corresponding spot rates for the bond's cash flow
`1` (linear) (default) | vector

(Optional) Interpolation method to compute the corresponding spot rates for the bond's cash flow, specified as an `NMBS`-by-`1` vector. Available methods are (`0`) nearest, (`1`) linear, and (`2`) cubic spline. For more information on the supported interpolation methods, see `interp1`.

Data Types: `double`

Output Arguments

`QtdFutPrice` — Quoted futures price, per $100 notional
vector

Quoted futures price, per $100 notional, returned as a `NINST`-by-`1` vector.

`AccrInt` — Accrued Interest due at delivery date, per $100 notional
vector

Accrued Interest due at delivery date, per $100 notional, returned as a `NINST`-by-`1` vector.

Version History

Introduced before R2006a

See Also

`convfactor` | `tfutbyprice`

Topics

tfutimprepo

Implied repo rates for Treasury bond future given price

Syntax

```
ImpliedRepo = tfutimprepo(ReinvestData,Price,QtdFutPrice,Settle,MatFut,
ConvFactor,CouponRate,Maturity)
```

Description

`ImpliedRepo = tfutimprepo(ReinvestData,Price,QtdFutPrice,Settle,MatFut, ConvFactor,CouponRate,Maturity)` computes the implied repo rate that prevents arbitrage of Treasury bond futures, given the clean price at the settlement and delivery dates.

Examples

Compute the Implied Repo Rates for Treasury Bond Futures Given the Price

This example shows how to compute the implied repo rate given the following set of data.

```
ReinvestData = [0.018  3];
Price = [114.4160; 113.1710];
QtdFutPrice = [114.1201; 113.7090];
Settle = datetime(2002,11,15);
MatFut = [datetime(2002,12,15) ; datetime(2003,3,15)];
ConvFactor = [1; 0.9854];
CouponRate = [0.06; 0.0575];
Maturity = [datetime(2009,8,15) ; datetime(2010,8,15)];

ImpliedRepo = tfutimprepo(ReinvestData, Price, QtdFutPrice, ...
Settle, MatFut, ConvFactor, CouponRate, Maturity)
```

```
ImpliedRepo = 2×1

    0.0200
    0.0200
```

Input Arguments

`ReinvestData` — Reinvestment of intervening coupons

matrix

Reinvestment of intervening coupons, specified as a number of futures `NFUT`-by-2 matrix of rates and bases in the form of `[ReinvestRate ReinvestBasis]`.

`ReinvestRate` is the simple reinvestment rate, in decimal. Specify `ReinvestBasis` as `0` = not reinvested, `2` = actual/360, or `3` = actual/365.

Data Types: `double`

`Price` — Current bond price per $100 notional
scalar numeric | vector

Current bond price per $100 notional, specified as a scalar numeric or an `NINST`-by-`1` vector.

Data Types: `double`

`QtdFutPrice` — Quoted bond futures price per $100 notional
scalar numeric | vector

Quoted bond futures price per $100 notional, specified as a scalar numeric or an `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement/valuation date of futures contract
datetime array | string array | date character vector

Settlement/valuation date of futures contract, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tfutimprepo` also accepts serial date numbers as inputs, but they are not recommended.

`MatFut` — Maturity dates (or anticipated delivery dates) of futures contract
datetime array | string array | date character vector

Maturity dates (or anticipated delivery dates) of futures contract, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tfutimprepo` also accepts serial date numbers as inputs, but they are not recommended.

`ConvFactor` — Conversion factor
numeric

Conversion factor, specified using `convfactor`.

Data Types: `double` | `char` | `cell`

`CouponRate` — Underlying bond annual coupon
scalar numeric in decimal | vector in decimals

Underlying bond annual coupon, specified as a scalar numeric decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Maturity` — Underlying bond maturity date
datetime array | string array | date character vector

Underlying bond maturity date, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tfutimprepo` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`ImpliedRepo` — Implied annual repo rate with an actual/360 basis
vector in decimals

Implied annual repo rate (in decimals) with an actual/360 basis, returned as a `NINST`-by-`1` vector.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `tfutimprepo` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`tfutpricebyrepo` | `tfutyieldbyrepo`

Topics

“Computing Treasury Bill Price and Yield”
“Treasury Bills Defined”

tfutpricebyrepo

Calculates Treasury bond futures price given the implied repo rates

Syntax

```
[QtdFutPrice,AccrInt] = tfutpricebyrepo(RepoData,ReinvestData,Price,Settle,
MatFut,ConvFactor,CouponRate,Maturity)
```

Description

`[QtdFutPrice,AccrInt] = tfutpricebyrepo(RepoData,ReinvestData,Price,Settle,MatFut,ConvFactor,CouponRate,Maturity)` computes the theoretical futures bond price given the settlement price, the repo/funding rates, and the reinvestment rate.

Examples

Compute Treasury Bond Futures Price Given the Implied Repo Rates

This example shows how to compute the quoted futures price and accrued interest due on the target delivery date, given the following data.

```
RepoData     = [0.020  2];
ReinvestData = [0.018  3];
Price        = [114.416; 113.171];
Settle       = datetime(2002,11,15);
MatFut       = [datetime(2002,12,15) ; datetime(2003,3,15)];
ConvFactor   = [1 ; 0.9854];
CouponRate   = [0.06;0.0575];
Maturity     = [datetime(2009,8,15) ; datetime(2010,8,15)];

[QtdFutPrice AccrInt] = tfutpricebyrepo(RepoData, ...
ReinvestData, Price, Settle, MatFut, ConvFactor, CouponRate, ...
Maturity)
```

```
QtdFutPrice = 2×1

  114.1201
  113.7090


AccrInt = 2×1

    1.9891
    0.4448
```

Input Arguments

RepoData — Simple term repo/funding rates

matrix

Simple term repo/funding rates, specified as a number of futures `NFUT`-by-`2` matrix of rates in decimal and their bases in the form of `[RepoRate RepoBasis]`.

Specify `RepoBasis` as `2` = actual/360 or `3` = actual/365.

Data Types: `double`

`ReinvestData` — Reinvestment of intervening coupons
matrix

Reinvestment of intervening coupons, specified as a number of futures `NFUT`-by-`2` matrix of rates and bases in the form of `[ReinvestRate ReinvestBasis]`.

`ReinvestRate` is the simple reinvestment rate, in decimal. Specify `ReinvestBasis` as `0` = not reinvested, `2` = actual/360, or `3` = actual/365.

Data Types: `double`

`Price` — Current bond price per $100 notional
scalar numeric | vector

Current bond price per $100 notional, specified as a scalar numeric or an `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement/valuation date of futures contract
datetime array | string array | date character vector

Settlement/valuation date of futures contract, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tfutpricebyrepo` also accepts serial date numbers as inputs, but they are not recommended.

`MatFut` — Maturity dates (or anticipated delivery dates) of futures contract
datetime array | string array | date character vector

Maturity dates (or anticipated delivery dates) of futures contract, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tfutpricebyrepo` also accepts serial date numbers as inputs, but they are not recommended.

`ConvFactor` — Conversion factor
numeric

Conversion factor, specified using `convfactor`.

Data Types: `double` | `char` | `cell`

`CouponRate` — Underlying bond annual coupon
scalar numeric in decimal | vector in decimals

Underlying bond annual coupon, specified as a scalar numeric decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Maturity` — Underlying bond maturity date
datetime array | string array | date character vector

Underlying bond maturity date, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tfutpricebyrepo` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`QtdFutPrice` — Quoted futures price, per $100 notional
vector

Quoted futures price, per $100 notional, returned as a `NINST`-by-`1` vector.

`AccrInt` — Accrued Interest due at delivery date, per $100 notional
vector

Accrued Interest due at delivery date, per $100 notional, returned as a `NINST`-by-`1` vector.

Version History

Introduced before R2006a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `tfutpricebyrepo` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`tfutimprepo` | `tfutyieldbyrepo`

Topics
“Computing Treasury Bill Price and Yield”
“Treasury Bills Defined”

tfutyieldbyrepo

Calculates Treasury bond futures yield given the implied repo rates

Syntax

```
FwdYield = tfutyieldbyrepo(RepoData,ReinvestData,Yield,Settle,MatFut,
ConvFactor,CouponRate,Maturity)
```

Description

`FwdYield = tfutyieldbyrepo(RepoData,ReinvestData,Yield,Settle,MatFut, ConvFactor,CouponRate,Maturity)` computes the theoretical futures bond yield given the settlement yield, the repo/funding rate, and the reinvestment rate.

Examples

Compute the Treasury Bond Futures Yield Given the Implied Repo Rates

This example shows how to compute the quoted futures bond yield, given the following data.

```
RepoData     = [0.020  2];
ReinvestData = [0.018  3];
Yield        = [0.0215; 0.0257];
Settle       = datetime(2002,11,15);
MatFut       = [datetime(2002,12,15) ; datetime(2003,3,15)];
ConvFactor   = [1; 0.9854];
CouponRate   = [0.06; 0.0575];
Maturity     = [datetime(2009,8,15) ; datetime(2010,8,15)];

FwdYield = tfutyieldbyrepo(RepoData, ReinvestData, Yield,...
    Settle, MatFut, ConvFactor, CouponRate, Maturity)
```

```
FwdYield = 2×1

    0.0221
    0.0282
```

Input Arguments

RepoData — Simple term repo/funding rates

matrix

Simple term repo/funding rates, specified as a number of futures `NFUT`-by-2 matrix of rates in decimal and their bases in the form of `[RepoRate RepoBasis]`.

Specify `RepoBasis` as 2 = actual/360 or 3 = actual/365.

Data Types: `double`

`ReinvestData` — Reinvestment of intervening coupons
matrix

Reinvestment of intervening coupons, specified as a number of futures `NFUT`-by-`2` matrix of rates and bases in the form of `[ReinvestRate ReinvestBasis]`.

`ReinvestRate` is the simple reinvestment rate, in decimal. Specify `ReinvestBasis` as `0` = not reinvested, `2` = actual/360, or `3` = actual/365.

Data Types: `double`

`Yield` — Yield to maturity of Treasury bonds per $100 notional at `Settle`
scalar numeric | vector

Yield to maturity of Treasury bonds per $100 notional at `Settle`, specified as a scalar numeric or an `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement/valuation date of futures contract
datetime array | string array | date character vector

Settlement/valuation date of futures contract, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tfutyieldbyrepo` also accepts serial date numbers as inputs, but they are not recommended.

`MatFut` — Maturity dates (or anticipated delivery dates) of futures contract
datetime array | string array | date character vector

Maturity dates (or anticipated delivery dates) of futures contract, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tfutyieldbyrepo` also accepts serial date numbers as inputs, but they are not recommended.

`ConvFactor` — Conversion factor
numeric

Conversion factor, specified using `convfactor`.

Data Types: `double` | `char` | `cell`

`CouponRate` — Underlying bond annual coupon
scalar numeric in decimal | vector in decimals

Underlying bond annual coupon, specified as a scalar numeric decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Maturity` — Underlying bond maturity date
datetime array | string array | date character vector

Underlying bond maturity date, specified as a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tfutyieldbyrepo` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`FwdYield` — Forward yield to maturity compounded semiannually

vector in decimals

Forward yield to maturity, in decimals, compounded semiannually, returned as a `NINST`-by-`1` vector.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `tfutyieldbyrepo` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`tfutimprepo` | `tfutpricebyrepo`

Topics

“Computing Treasury Bill Price and Yield”
“Treasury Bills Defined”

toRateSpec

Convert `IRDataCurve` object to `RateSpec`

Syntax

```
F = toRateSpec(CurveObj,InpDates)
```

Description

`F = toRateSpec(CurveObj,InpDates)` computes `RateSpec` object for input dates for an `IRDataCurve` object. The `RateSpec` object that is identical to the `RateSpec` structure created by the function `intenvset`.

Examples

Convert an IRDataCurve Object to a RateSpec

This example shows how to convert an `IRDataCurve` object to a `RateSpec`. First, an `IRDataCurve` object is created using the function `IRDataCurve` constructor with `Dates` and `Data`, then this object is converted to a `RateSpec` structure using the `toRateSpec` method.

```
CurveSettle = datetime(2016,3,2);
Data = [2.09 2.47 2.71 3.12 3.43 3.85 4.57 4.58]/100;
Dates = datemnth(CurveSettle,12*[1 2 3 5 7 10 20 30]);
irdc = IRDataCurve('Forward',CurveSettle,Dates,Data);
toRateSpec(irdc, CurveSettle+30:30:CurveSettle+365)
```

```
ans = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [12x1 double]
            Rates: [12x1 double]
         EndTimes: [12x1 double]
       StartTimes: [12x1 double]
         EndDates: [12x1 double]
       StartDates: 736391
    ValuationDate: 736391
            Basis: 0
     EndMonthRule: 1
```

Input Arguments

CurveObj — Interest-rate curve object

object

Interest-rate curve object, specified by using `IRDataCurve`.

Data Types: `object`

`InpDates` — Input dates
datetime array | string array | date character vector

Input dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The input dates must be after the `Settle` date of `IRDataCurve`.

To support existing code, `toRateSpec` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`F` — Rate spec
object

Rate spec, returned as an object. The `RateSpec` object that is identical to the `RateSpec` structure created by the function `intenvset`.

Alternatively, you can convert the `RateSpec` object to a `ratecurve` object (see “Convert RateSpec to a ratecurve Object” on page 1-49) and then use the Financial Instruments Toolbox object-based framework for pricing instruments.

Version History

Introduced in R2008b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `toRateSpec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`ratecurve` | `IRDataCurve` | `getForwardRates` | `getZeroRates` | `getDiscountFactors` | `getParYields`

Topics
“Creating Interest-Rate Curve Objects” on page 9-4
“Creating an IRDataCurve Object” on page 9-6
“Using the toRateSpec Function” on page 9-30
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Convert RateSpec to a ratecurve Object” on page 1-49

“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

toRateSpec

Convert `IRFunctionCurve` object to `RateSpec`

Syntax

```
F = toRateSpec(CurveObj,InpDates)
```

Description

`F = toRateSpec(CurveObj,InpDates)` computes `RateSpec` object for input dates for an `IRFunctionCurve` object. The `RateSpec` object that is identical to the `RateSpec` structure created by the function `intenvset`.

Examples

Convert an `IRFunctionCurve` Object to a `RateSpec`

This example shows how to convert an `IRFunctionCurve` object to a `RateSpec`. First, an `IRFunctionCurve` object is created using the function `IRFunctionCurve` constructor, then a `RateSpec` structure is created using the `toRateSpec` method.

```
irfc = IRFunctionCurve('Forward',today,@(t) polyval([-0.0001 0.003 0.02],t));
toRateSpec(irfc, today+30:30:today+365)
```

```
ans = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [12x1 double]
            Rates: [12x1 double]
         EndTimes: [12x1 double]
       StartTimes: [12x1 double]
         EndDates: [12x1 double]
       StartDates: 738764
    ValuationDate: 738764
            Basis: 0
     EndMonthRule: 1
```

Input Arguments

`CurveObj` — Interest-rate curve object

object

Interest-rate curve object, specified by using `IRFunctionCurve`.

Data Types: `object`

`InpDates` — Input dates

datetime array | string array | date character vector

Input dates, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The input dates must be after the `Settle` date of `IRFunctionCurve`.

To support existing code, `toRateSpec` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

F — Rate spec

object

Rate spec, returned as an object. The `RateSpec` object that is identical to the `RateSpec` structure created by the function `intenvset`.

Alternatively, you can convert the `RateSpec` object to a `ratecurve` object (see “Convert RateSpec to a ratecurve Object” on page 1-49) and then use the Financial Instruments Toolbox object-based framework for pricing instruments.

Version History

Introduced in R2008b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `toRateSpec` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`IRFunctionCurve` | `ratecurve` | `getForwardRates` | `getZeroRates` | `getDiscountFactors` | `getParYields`

Topics

“Creating an IRFunctionCurve Object” on page 9-16
“Using the toRateSpec Function” on page 9-30
“Interest-Rate Curve Objects and Workflow” on page 9-2
“Convert RateSpec to a ratecurve Object” on page 1-49
“Mapping Financial Instruments Toolbox Curve Functions to Object-Based Framework” on page 1-95

zeroprice

Price zero-coupon instruments given yield

Syntax

```
Price = zeroprice(Yield,Settle,Maturity)
Price = zeroprice( ___ ,Period,Basis,EndMonthRule)
```

Description

`Price = zeroprice(Yield,Settle,Maturity)` prices zero-coupon instruments given a yield. `zeroprice` calculates the prices for a portfolio of general short and long-term zero-coupon instruments given the yield of reference bonds. In other words, if the zero-coupon computed with this yield is used to discount the reference bond, the value of that reference bond is equal to its price.

`Price = zeroprice( ___ ,Period,Basis,EndMonthRule)` adds optional arguments for `Period`, `Basis`, and `EndMonthRule`.

Examples

Compute the Price of a Short-Term Zero-Coupon Instrument

This example shows how to compute the price of a short-term zero-coupon instrument.

```
Settle = datetime(1993,6,24);
Maturity = datetime(1993,11,1);
Period = 2;
Basis = 0;
Yield = 0.04;

Price = zeroprice(Yield, Settle, Maturity, Period, Basis)
```

```
Price = 98.6066
```

Compute the Prices of a Portfolio of Two Zero-Coupon Instruments

This example shows how to compute the prices of a portfolio of two zero-coupon instruments, one short-term, and the other long-term.

```
Settle = datetime(1993,6,24);
Maturity = [datetime(1993,11,1) ; datetime(2024,1,15)];
Basis = [0; 1];
Yield = [0.04; 0.1];

Price = zeroprice(Yield, Settle, Maturity, [], Basis)
```

```
Price = 2×1
```

```
   98.6066
    5.0697
```

Input Arguments

`Yield` — Reference bond yield

scalar | vector

Reference bond yield, specified as a scalar or a `NZERO`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a `NZERO`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `zeroprice` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NZERO`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `zeroprice` also accepts serial date numbers as inputs, but they are not recommended.

`Period` — Number of coupons in one year

`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

(Optional) Number of coupons in one year, specified as a positive integer for the values `1,2,4,6,12` in a `NZERO`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of bond

`0` (actual/actual) (default) | vector of positive integers of the set `[1...13]`

(Optional) Day-count basis of the bond, specified as a positive integer using a `NZERO`-by-`1` vector.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
1 (in effect) (default) | nonnegative integer with value of `0` or `1`

(Optional) End-of-month rule flag, specified as a nonnegative integer with a value of `0` or `1` using a `NZERO`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

Output Arguments

`Price` — Price for each zero-coupon instrument
vector

Price for each zero-coupon instrument (per $100 notional), returned as a column vector.

Algorithms

To compute the price when `Period` is `1` or `0` for the quasi-coupon periods to redemption, `zeroprice` uses the formula

$$Price = \frac{RV}{1 + \left(\frac{DSR}{E} \times \frac{Y}{M}\right)}$$

.

Quasi-coupon periods are the coupon periods that would exist if the bond were paying interest at a rate other than zero.

When there is more than one quasi-coupon period to the redemption date, `zeroprice` uses the formula

$$Price = \frac{RV}{\left(1 + \frac{Y}{M}\right)^{Nq - 1 + \frac{DSC}{E}}}$$

.

The elements of the equations are defined as follows.

Variable	Definition
DSC	Number of days from settlement date to next quasi-coupon date as if the security paid periodic interest.
DSR	Number of days from settlement date to the redemption date (call date, put date, and so on).
E	Number of days in quasi-coupon period.
M	Number of quasi-coupon periods per year (standard for the particular security involved).
Nq	Number of quasi-coupon periods between settlement date and redemption date. If this number contains a fractional part, raise it to the next whole number.
Price	Dollar price per $100 par value.
RV	Redemption value.
Y	Annual yield (decimal) when held to redemption.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `zeroprice` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Mayle, Jan. *Standard Securities Calculation Methods.* 3rd Edition, Vol. 1, Securities Industry Association, Inc., New York, 1993, ISBN 1-882936-01-9. Vol. 2, 1994, ISBN 1-882936-02-7.

See Also

`bndprice` | `cdprice` | `tbillprice` | `zeroyield`

Topics

“Computing Treasury Bill Price and Yield”

“Pricing Treasury Notes” on page 6-5
“Pricing Corporate Bonds” on page 6-7
“Measuring Zero-Coupon Bond Function Quality” on page 6-5

zeroyield

Yield of zero-coupon instruments given price

Syntax

```
Yield = zeroyield(Price,Settle,Maturity)
Yield = zeroyield( ___ ,Period,Basis,EndMonthRule)
```

Description

`Yield = zeroyield(Price,Settle,Maturity)` computes the yield of zero-coupon instruments given price. `zeroyield` calculates the bond-equivalent yield for a portfolio of general short and long-term zero-coupon instruments given the price of the instruments. In other words, if the zero-coupon computed with this yield is used to discount the reference bond, the value of that reference bond is equal to its price

`Yield = zeroyield( ___ ,Period,Basis,EndMonthRule)` adds optional arguments for `Period`, `Basis`, and `EndMonthRule`.

Examples

Compute the Yield of a Short-Term Zero-Coupon Instrument

This example shows how to compute the yield of a short-term zero-coupon instrument.

```
Settle   = datetime(1993,6,24);
Maturity = datetime(1993,11,1);
Basis    = 0;
Price    = 95;

Yield = zeroyield(Price, Settle, Maturity, [], Basis)
```

```
Yield = 0.1490
```

Compute the Yield of a Short-Term Zero-Coupon Instrument Using a Day-Count Basis of 30/360 (SIA)

This example shows how to compute the yield of a short-term zero-coupon instrument using a day-count basis of 30/360 (SIA).

```
Settle   = datetime(1993,6,24);
Maturity = datetime(1993,11,1);
Basis    = 1;
Price    = 95;

Yield = zeroyield(Price, Settle, Maturity, [], Basis)
```

```
Yield = 0.1492
```

Compute the Yield of a Long-Term Zero-Coupon Instrument

This example shows how to compute the yield of a long-term zero-coupon instrument.

```
Settle   = datetime(1993,6,24);
Maturity = datetime(2024,6,15);
Basis    = 0;
Price    = 9;

Yield = zeroyield(Price, Settle, Maturity, [], Basis)
```

```
Yield = 0.0793
```

Input Arguments

`Price` — Reference bond price

scalar | vector

Reference bond price, specified as a scalar or a `NZERO`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a `NZERO`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `zeroyield` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NZERO`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `zeroyield` also accepts serial date numbers as inputs, but they are not recommended.

`Period` — Number of coupons in one year

`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

(Optional) Number of coupons in one year, specified as a positive integer for the values `1,2,4,6,12` in a `NZERO`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis of bond

`0` (actual/actual) (default) | vector of positive integers of the set `[1...13]`

(Optional) Day-count basis of the bond, specified as a positive integer using a `NZERO`-by-`1` vector.

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

For more information, see “Basis” on page 2-228.

Note When the `Maturity` date is fewer than 182 days away and the `Basis` is `actual/365`, the `zeroyield` uses a simple-interest algorithm. If `Maturity` is more than 182 days away, `zeroyield` uses present value calculations.

When the `Basis` is `actual/360`, the simple interest algorithm gives the money-market yield for short (1–6 months to maturity) Treasury bills.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer with value of `0` or `1`

(Optional) End-of-month rule flag, specified as a nonnegative integer with a value of `0` or `1` using a `NZERO`-by-`1` 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 coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

Output Arguments

`Yield` — Bond-equivalent yield for each zero-coupon instrument
vector

Bond-equivalent yield for each zero-coupon instrument, returned as a column vector.

Algorithms

To compute the yield when there is zero or one quasi-coupon period to redemption, `zeroyield` uses the formula

$$Yield = \left(\frac{RV - P}{P}\right) \times \left(\frac{M \times E}{DSR}\right)$$

.

Quasi-coupon periods are the coupon periods which would exist if the bond was paying interest at a rate other than zero. The first term calculates the yield on invested dollars. The second term converts this yield to a per annum basis.

When there is more than one quasi-coupon period to the redemption date, `zeroyield` uses the formula

$$Yield = \left(\left(\frac{RV}{P}\right)^{\frac{1}{Nq - 1 + \frac{DSC}{E}}} - 1\right) \times M$$

The elements of the equations are defined as follows.

Variable	Definition
DSC	Number of days from the settlement date to next quasi-coupon date as if the security paid periodic interest.
DSR	Number of days from the settlement date to redemption date (call date, put date, and so on).
E	Number of days in quasi-coupon period.
M	Number of quasi-coupon periods per year (standard for the particular security involved).
Nq	Number of quasi-coupon periods between the settlement date and redemption date. If this number contains a fractional part, raise it to the next whole number.
P	Dollar price per $100 par value.
RV	Redemption value.
Yield	Annual yield (decimal) when held to redemption.

Version History

Introduced before R2006a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `zeroyield` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Mayle, Jan. *Standard Securities Calculation Methods.* 3rd Edition, Vol. 1, Securities Industry Association, Inc., New York, 1993, ISBN 1-882936-01-9. Vol. 2, 1994, ISBN 1-882936-02-7.

See Also

`bndyield` | `cdyield` | `tbillyield` | `zeroprice`

Topics

“Computing Treasury Bill Price and Yield”
“Pricing Treasury Notes” on page 6-5
“Pricing Corporate Bonds” on page 6-7
“Measuring Zero-Coupon Bond Function Quality” on page 6-5

touchbybls

Price one-touch and no-touch binary options using Black-Scholes option pricing model

Syntax

```
Price = touchybls(RateSpec,StockSpec,Settle,Maturity,BarrierSpec,Barrier,
Payoff)
```

Description

`Price = touchybls(RateSpec,StockSpec,Settle,Maturity,BarrierSpec,Barrier, Payoff)` calculates one-touch and no-touch binary options using the Black-Scholes option pricing model.

Note Alternatively, you can use the `Touch` object to price one touch options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Price a One-Touch Option

Compute the price of a one-touch option using the following data:

```
AssetPrice = 105;
Rate = 0.1;
Volatility = 0.2;
Settle = datetime(2018,1,1);
Maturity = datetime(2018,6,1);
```

Define the `RateSpec` using `intenvset`.

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

Define the `StockSpec` using `stockspec`.

```
DividendType = "Continuous";
DividendYield = Rate - 0.1;
StockSpec = stockspec(Volatility, AssetPrice, DividendType, DividendYield);
```

Calculate the price of a one-touch binary option.

```
BarrierSpec = "OT";
Barrier = 100;
Payoff = 15;

Price = touchbybls(RateSpec, StockSpec, Settle, Maturity, BarrierSpec, Barrier, Payoff)
```

```
Price = 9.4102
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities, the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the touch option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `touchbybls` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date

datetime array | string array | date character vector

Maturity date for the touch option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `touchbybls` also accepts serial date numbers as inputs, but they are not recommended.

BarrierSpec — Barrier option type

character vector with values `'OT'` or `'NT'`

Barrier option type, specified as an `NINST`-by-`1` cell array of character vectors with the following values:

- `'OT'` — One-touch

  The one-touch option provides a payoff if the underlying asset ever trades at or beyond the `Barrier` level. Otherwise, the `Payoff` is zero.

- `'NT'` — No-touch

  The no-touch option provides a `Payoff` if the underlying asset never trades at or beyond the `Barrier` level. Otherwise, the `Payoff` is zero.

Data Types: `char` | `cell`

`Barrier` — Barrier value
numeric

Barrier value, specified as an `NINST`-by-`1` matrix of numeric values.

Data Types: `double`

`Payoff` — Payoff value
numeric

Payoff value, specified as an `NINST`-by-`1` matrix of numeric values.

Note The payoff value is calculated for the point in time that the `Barrier` value is reached. The payoff is either cash or nothing. If a no-touch option is specified using the `BarrierSpec`, the payoff is at the `Maturity` of the option.

Data Types: `double`

Output Arguments

`Price` — Expected prices for one-touch options
matrix

Expected prices for one-touch options at time 0, returned as an `NINST`-by-`1` matrix.

More About

Touch and No-Touch Options

The one-touch and no-touch options provide a payoff if the underlying spot either ever or never trades at or beyond the barrier level. Otherwise, the payoff is zero.

Only two outcomes are possible with a one-touch option if a trader holds the contract all the way through expiration:

- The target price (`Barrier`) is reached and the trader collects the full premium.
- The target price (`Barrier`) is not reached and the trader loses the amount originally paid to open the trade.

Version History

Introduced in R2019b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `touchbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Haug, E. *The Complete Guide to Option Pricing Formulas.* McGraw-Hill Education, 2007.

[2] Wystup, U. *FX Options and Structured Products.* Wiley Finance, 2007.

See Also

`touchsensbybls` | `dbltouchbybls` | `dbltouchsensbybls` | `Touch`

Topics

“One-Touch and Double One-Touch Options” on page 3-30
“Supported Equity Derivative Functions” on page 3-19

touchsensbybls

Calculate price or sensitivities for one-touch and no-touch binary options using Black-Scholes option pricing model

Syntax

```
PriceSens = touchsensbybls(RateSpec,StockSpec,Settle,Maturity,BarrierSpec,
Barrier,Payoff)
PriceSens = touchsensbybls( ___ ,Name,Value)
```

Description

`PriceSens = touchsensbybls(RateSpec,StockSpec,Settle,Maturity,BarrierSpec, Barrier,Payoff)` calculates the price and sensitivities for one-touch and no-touch binary options using the Black-Scholes option pricing model.

Note Alternatively, you can use the `Touch` object to calculate price or sensitivities for one touch options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = touchsensbybls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Calculate the Price and Sensitivities for a One-Touch Option

Compute the price and sensitivities for a one-touch option using the following data:

```
AssetPrice = 105;
Rate = 0.1;
Volatility = 0.2;
Settle = datetime(2018,1,1);
Maturity = datetime(2018,6,1);
```

Define the `RateSpec` using `intenvset`.

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

Define the `StockSpec` using `stockspec`.

```
DividendType = "Continuous";
DividendYield = Rate - 0.1;
StockSpec = stockspec(Volatility, AssetPrice, DividendType, DividendYield);
```

Define the sensitivities.

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

Calculate the price and sensitivities for a one-touch binary option.

```
BarrierSpec = "OT";
Barrier = 100;
Payoff = 15;

[Price, Delta, Gamma] = touchsensbybls(RateSpec, StockSpec, Settle, Maturity, BarrierSpec, Barrie

Price = 9.4102

Delta = -0.9415

Gamma = 0.0685
```

Input Arguments

RateSpec — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

StockSpec — Stock specification for underlying asset

structure

Stock specification for the underlying asset. For information on the stock specification, see `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities, the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

Settle — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the touch option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `touchsensbybls` also accepts serial date numbers as inputs, but they are not recommended.

Maturity — Maturity date

datetime array | string array | date character vector

Maturity date for the touch option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `touchsensbybls` also accepts serial date numbers as inputs, but they are not recommended.

BarrierSpec — Barrier option type

character vector with values `'OT'` or `'NT'`

Barrier option type, specified as an NINST-by-1 cell array of character vectors with the following values:

- 'OT' — One-touch. The one-touch option provides a payoff if the underlying spot ever trades at or beyond the Barrier level and the payoff is zero otherwise.
- 'NT' — No-touch. The no-touch option provides a Payoff if the underlying spot ever never trades at or beyond the Barrier level and the Payoff is zero otherwise.

Data Types: char | cell

Barrier — Barrier value
numeric

Barrier value, specified as an NINST-by-1 matrix of numeric values.

Data Types: double

Payoff — Payoff value
numeric

Payoff value, specified as an NINST-by-1 matrix of numeric values.

Note The payoff value is calculated for the point in time that the Barrier value is reached. The payoff is either cash or nothing. If a no-touch option is specified using the BarrierSpec, the payoff is at the Maturity of the option.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: Price = touchsensbybls(RateSpec,StockSpec,Settle,Maturity,BarrierSpec,Barrier,Payoff, 'OutSpec','Delta')

OutSpec — Define outputs
{'Price'} (default) | character vector with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All' | cell array of character vectors with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'

Define outputs, specified as the comma-separated pair consisting of 'OutSpec' and an NOUT- by-1 or a 1-by-NOUT cell array of character vectors with possible values of 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All'.

OutSpec = {'All'} specifies that the output is Delta, Gamma, Vega, Lambda, Rho, Theta, and Price, in that order. This is the same as specifying OutSpec to include each sensitivity.

Example: OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}

Data Types: char | cell

Output Arguments

`PriceSens` — Expected prices or sensitivities for one-touch options

matrix

Expected prices at time 0 or sensitivities (defined using `OutSpec`) for one-touch options, returned as an `NINST`-by-`1` matrix.

More About

Touch and No-Touch Options

The one-touch and no-touch options provide a payoff if the underlying spot either ever or never trades at or beyond the barrier level. Otherwise, the payoff is zero.

Only two outcomes are possible with a one-touch option if a trader holds the contract all the way through expiration:

- The target price (`Barrier`) is reached and the trader collects the full premium.
- The target price (`Barrier`) is not reached and the trader loses the amount originally paid to open the trade.

Version History

Introduced in R2019b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `touchsensbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Haug, E. *The Complete Guide to Option Pricing Formulas.* McGraw-Hill Education, 2007.

[2] Wystup, U. *FX Options and Structured Products.* Wiley Finance, 2007.

See Also

`touchbybls` | `dbltouchbybls` | `dbltouchsensbybls` | `Touch`

Topics

“One-Touch and Double One-Touch Options” on page 3-30
“Supported Equity Derivative Functions” on page 3-19

dbltouchbybls

Price double one-touch and double no-touch binary options using Black-Scholes option pricing model

Syntax

```
Price = dbltouchbybls(RateSpec,StockSpec,Settle,Maturity,BarrierSpec,Barrier,
Payoff)
```

Description

`Price = dbltouchbybls(RateSpec,StockSpec,Settle,Maturity,BarrierSpec,Barrier, Payoff)` calculates double one-touch and double no-touch binary options using Black-Scholes option pricing model.

Note Alternatively, you can use the `DoubleTouch` object to price double touch options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

Examples

Price a Double No-Touch Option

Compute the price of a double no-touch option using the following data:

```
AssetPrice = 105;
Rate = 0.1;
Volatility = 0.2;
Settle = datetime(2018,1,1);
Maturity = datetime(2018,6,1);
```

Define the `RateSpec` using `intenvset`.

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

Define the `StockSpec` using `stockspec`.

```
DividendType = "Continuous";
DividendYield = Rate - 0.03;
StockSpec = stockspec(Volatility, AssetPrice, DividendType, DividendYield);
```

Calculate the price of a double no-touch binary option.

```
BarrierSpec = "DNT";
Barrier = [120 80];
Payoff = 10;

Price = dbltouchbybls(RateSpec, StockSpec, Settle, Maturity, BarrierSpec, Barrier, Payoff)
```

```
Price = 6.3082
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset, specified by the `StockSpec` obtained from `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities, the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the double touch option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `dbltouchbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date for the double touch option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `dbltouchbybls` also accepts serial date numbers as inputs, but they are not recommended.

`BarrierSpec` — Double barrier option type

cell array of character vectors with values of `'DOT'` or `'DNT'` | string array with values of `"DOT"` or `"DNT"`

Double barrier option type, specified as an `NINST`-by-`1` cell array of character vectors or string array with the following values:

- `'DOT'` — Double one-touch. The double one-touch option defines two `Barrier` levels. A double one-touch option provides a `Payoff` if the underlying asset ever touches either the upper or lower `Barrier` levels.
- `'DNT'` — Double no-touch. The double no-touch option defines two `Barrier` levels. A double no-touch option provides a `Payoff` if the underlying asset ever never touches either the upper or lower `Barrier` levels.

Data Types: `char` | `cell` | `string`

`Barrier` — Double barrier value

numeric

Double barrier value, specified as an `NINST`-by-`2` matrix of numeric values, where the first column is Upper Barrier(1)(UB) and the second column is Lower Barrier(2)(LB). Barrier(1) must be greater than Barrier(2).

Data Types: `double`

`Payoff` — Payoff value

numeric

Payoff value, specified as an `NINST`-by-`1` matrix of numeric values, where each element is a `1`-by-`2` vector in which the first column is Barrier(1)(UB) and the second column is Barrier(2)(LB). Barrier(1) must be greater than Barrier(2).

Note The payoff value is calculated for the point in time that the `Barrier` value is reached. The payoff is either cash or nothing. If you specify a double no-touch option using `BarrierSpec`, the payoff is at the `Maturity` of the option.

Data Types: `double`

Output Arguments

`Price` — Expected prices for double one-touch options

matrix

Expected prices for double one-touch options at time 0, returned as an `NINST`-by-`1` matrix.

More About

Double One-Touch and Double No-Touch Options

Double one-touch options and double no-touch options work the same way as one-touch options, except that there are two barriers.

A double one-touch or double no-touch option provides a payoff if the underlying spot either ever or never touches either the upper or lower `Barrier` levels. If neither barrier level is breached prior to expiration, the option expires worthless and the trader loses all the premium paid to the broker for setting up the trade. For example, if the current USD/EUR rate is 1.15, and the trader believes that this rate will change significantly over the next 15 days, the trader can use a double one-touch option with barriers at 1.10 and 1.20. The trader can profit if the rate moves beyond either of the two barriers.

Version History

Introduced in R2019b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `dbltouchbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Haug, E. *The Complete Guide to Option Pricing Formulas.* McGraw-Hill Education, 2007.

[2] Wystup, U. *FX Options and Structured Products.* Wiley Finance, 2007.

See Also

`touchbybls` | `touchsensbybls` | `dbltouchsensbybls` | `DoubleTouch`

Topics

“One-Touch and Double One-Touch Options” on page 3-30
“Supported Equity Derivative Functions” on page 3-19

dbltouchsensbybls

Calculate prices and sensitivities for double one-touch and double no-touch binary options using Black-Scholes option pricing model

Syntax

```
PriceSens = dbltouchsensbybls(RateSpec,StockSpec,Settle,Maturity,BarrierSpec,
Barrier,Payoff)
PriceSens = dbltouchsensbybls( ___ ,Name,Value)
```

Description

`PriceSens = dbltouchsensbybls(RateSpec,StockSpec,Settle,Maturity,BarrierSpec, Barrier,Payoff)` calculates prices and sensitivities for double one-touch and double no-touch binary options using the Black-Scholes option pricing model.

Note Alternatively, you can use the `DoubleTouch` object to calculate price or sensitivities for double touch options. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

`PriceSens = dbltouchsensbybls( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Calculate the Price and Sensitivities for a Double No-Touch Option

Compute the price and sensitivities for a double no-touch option using the following data:

```
AssetPrice = 105;
Rate = 0.1;
Volatility = 0.2;
Settle = datetime(2018,1,1);
Maturity = datetime(2018,6,1);
```

Define the `RateSpec` using `intenvset`.

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

Define the `StockSpec` using `stockspec`.

```
DividendType = "Continuous";
DividendYield = Rate - 0.03;
StockSpec = stockspec(Volatility, AssetPrice, DividendType, DividendYield);
```

Define the sensitivities.

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

Calculate the price and sensitivities for a double no-touch binary option.

```
BarrierSpec = "DNT";
Barrier = [120 80];
Payoff = 10;

[Price, Delta, Gamma] = dbltouchsensbybls(RateSpec, StockSpec, Settle, Maturity, BarrierSpec, Ba

Price = 6.3082

Delta = -0.2770

Gamma = -0.0311
```

Input Arguments

`RateSpec` — Interest-rate term structure

structure

Interest-rate term structure (annualized and continuously compounded), specified by the `RateSpec` obtained from `intenvset`. For information on the interest-rate specification, see `intenvset`.

Data Types: `struct`

`StockSpec` — Stock specification for underlying asset

structure

Stock specification for the underlying asset, specified by the `StockSpec` obtained from `stockspec`.

`stockspec` handles several types of underlying assets. For example, for physical commodities, the price is `StockSpec.Asset`, the volatility is `StockSpec.Sigma`, and the convenience yield is `StockSpec.DividendAmounts`.

Data Types: `struct`

`Settle` — Settlement or trade date

datetime array | string array | date character vector

Settlement or trade date for the double touch option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `dbltouchsensbybls` also accepts serial date numbers as inputs, but they are not recommended.

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date for the double touch option, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `dbltouchsensbybls` also accepts serial date numbers as inputs, but they are not recommended.

`BarrierSpec` — Double barrier option type

cell array of character vectors with values of `'DOT'` or `'DNT'` | string array with values of `"DOT"` or `"DNT"`

Double barrier option type, specified as an NINST-by-1 cell array of character vectors or string array with the following values:

- 'DOT' — Double one-touch. The double one-touch option defines two Barrier levels. A double one-touch option provides a Payoff if the underlying asset ever touches either the upper or lower Barrier levels.
- 'DNT' — Double no-touch. The double no-touch option defines two Barrier levels. A double no-touch option provides a Payoff if the underlying asset ever never touches either the upper or lower Barrier levels.

Data Types: char | cell | string

Barrier — Double barrier value
numeric

Double barrier value, specified as an NINST-by-2 matrix of numeric values, where the first column is Upper Barrier(1)(UB) and the second column is Lower Barrier(2)(LB). Barrier(1) must be greater than Barrier(2).

Data Types: double

Payoff — Payoff value
numeric

Payoff value, specified as an NINST-by-1 matrix of numeric values, where each element is a 1-by-2 vector in which the first column is Barrier(1)(UB) and the second column is Barrier(2)(LB). Barrier(1) must be greater than Barrier(2).

Note The payoff value is calculated for the point in time that the Barrier value is reached. The payoff is either cash or nothing. If you specify a double no-touch option using BarrierSpec, the payoff is at the Maturity of the option.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: PriceSens = dbltouchsensbybls(RateSpec,StockSpec,OptSpec,Strike,Settle,Maturity,BarrierSpec,Barrier,'OutSpec','Delta')

OutSpec — Define outputs
{'Price'} (default) | character vector with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All' | cell array of character vectors with values 'Price', 'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', and 'All' | string array with values "Price", "Delta", "Gamma", "Vega", "Lambda", "Rho", "Theta", and "All"

Define outputs, specified as the comma-separated pair consisting of `'OutSpec'` and an `NOUT`- by-`1` or a `1`-by-`NOUT` cell array of character vectors with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`.

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

Example: `OutSpec = {'delta','gamma','vega','lambda','rho','theta','price'}`

Data Types: `char` | `cell`

Output Arguments

`PriceSens` — Expected prices or sensitivities for double one-touch options

matrix

Expected prices at time 0 or sensitivities (defined using `OutSpec`) for double one-touch options, returned as an `NINST`-by-`1` matrix.

More About

Double One-Touch and Double No-Touch Options

Double one-touch options and double no-touch options work the same way as one-touch options, except that there are two barriers.

A double one-touch or double no-touch option provides a payoff if the underlying spot either ever or never touches either the upper or lower `Barrier` levels. If neither barrier level is breached prior to expiration, the option expires worthless and the trader loses all the premium paid to the broker for setting up the trade. For example, if the current USD/EUR rate is 1.15, and the trader believes that this rate will change significantly over the next 15 days, the trader can use a double one-touch option with barriers at 1.10 and 1.20. The trader can profit if the rate moves beyond either of the two barriers.

Version History

Introduced in R2019b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `dbltouchsensbybls` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Haug, E. *The Complete Guide to Option Pricing Formulas.* McGraw-Hill Education, 2007.

[2] Wystup, U. *FX Options and Structured Products.* Wiley Finance, 2007.

See Also

`touchbybls` | `touchsensbybls` | `dbltouchbybls` | `DoubleTouch`

Topics

“One-Touch and Double One-Touch Options” on page 3-30
“Supported Equity Derivative Functions” on page 3-19

fininstrument

Create specified instrument object type

Syntax

```
Instrument = fininstrument(InstrumentType,Name,Value)
```

Description

`Instrument = fininstrument(InstrumentType,Name,Value)` creates an instrument object for one or more instruments specified by the `InstrumentType` and specifies options using one or more name-value pair arguments. The available name-value pair arguments depend on the `InstrumentType` you specify.

For more information on the workflow for creating an instrument object, a model object, and a pricer object, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods, see “Choose Instruments, Models, and Pricers” on page 1-53.

Examples

Use `fininstrument` to Create `OptionEmbeddedFixedBond` Instrument

Use `fininstrument` to create an `OptionEmbeddedFixedBond` instrument object.

```
CallDates = datetime(2025,9,15) + calyears([0 1 2]');
CallStrikes = [101 103 105]';
CallSchedule = timetable(CallDates,CallStrikes);
OptionEmbedFixedBOption = fininstrument("OptionEmbeddedFixedBond",'Maturity',"15-Sep-2031",'Coupo
```

```
OptionEmbedFixedBOption =
  OptionEmbeddedFixedBond with properties:

                 CouponRate: 0.0300
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
              FirstCouponDate: NaT
               LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2031
                   CallDates: [3x1 datetime]
                    PutDates: [0x1 datetime]
                CallSchedule: [3x1 timetable]
```

```
      PutSchedule: [0x0 timetable]
CallExerciseStyle: "bermudan"
 PutExerciseStyle: [0x0 string]
             Name: "option_embedded_fixedbond"
```

Input Arguments

`InstrumentType` — Type of instrument

character vector | string

Type of instrument, specified as a scalar string or character vector.

Use these options for interest-rate instruments:

- `"Deposit"` — For more information, see `Deposit`.
- `"FRA"` — For more information, see `FRA`.
- `"FixedBond"` — For more information, see `FixedBond`.
- `"FixedBondOption"` — For more information, see `FixedBondOption`.
- `"FloatBond"` — For more information, see `FloatBond`.
- `"FloatBondOption"` — For more information, see `FloatBondOption`.
- `"OptionEmbeddedFixedBond"` — For more information, see `OptionEmbeddedFixedBond`.
- `"OptionEmbeddedFloatBond"` — For more information, see `OptionEmbeddedFloatBond`.
- `"Swap"` — For more information, see `Swap`.
- `"Cap"` — For more information, see `Cap`.
- `"Floor"` — For more information, see `Floor`.
- `"Swaption"` — For more information, see `Swaption`.
- `"STIRFuture"` — For more information, see `STIRFuture`.
- `"OISFuture"` — For more information, see `OISFuture`.
- `"OvernightIndexSwap"` — For more information, see `OvernightIndexedSwap`.
- `"BondFuture"` — For more information, see `BondFuture`.

Use these options for inflation instruments:

- `"InflationBond"` — For more information, see `InflationBond`.
- `"YearYearInflationSwap"` — For more information, see `YearYearInflationSwap`.
- `"ZeroCouponInflationSwap"` — For more information, see `ZeroCouponInflationSwap`.
- `"ConvertibleBond"` — For more information, see `ConvertibleBond`.

Use these options for equity, commodity, FX, or energy instruments:

- `"Vanilla"` — For more information, see `Vanilla`.
- `"Lookback"` — For more information, see `Lookback`.
- `"PartialLookback"` — For more information, see `PartialLookback`.
- `"Barrier"` — For more information, see `Barrier`.

- `"DoubleBarrier"` — For more information, see `DoubleBarrier`.
- `"Asian"` — For more information, see `Asian`.
- `"Spread"` — For more information, see `Spread`.
- `"VarianceSwap"` — For more information, see `VarianceSwap`.
- `"Touch"` — For more information, see `Touch`.
- `"DoubleTouch"` — For more information, see `DoubleTouch`.
- `"Cliquet"` — For more information, see `Cliquet`.
- `"Binary"` — For more information, see `Binary`.
- `"CommodityFuture"` — For more information, see `CommodityFuture`.
- `"EquityIndexFuture"` — For more information, see `EquityIndexFuture`.
- `"FXFuture"` — For more information, see `FXFuture`.
- `"ConvertibleBond"` — For more information, see `ConvertibleBond`.

Use these options for credit derivative instruments:

- `"CDS"` — For more information, see `CDS`.
- `"CDSOption"` — For more information, see `CDSOption`.

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Instrument = fininstrument("Cap",Name,Value)`

The available name-value pair arguments depend on the value you specify for `InstrumentType`.

Name-Value Pair Arguments for Interest-Rate Instruments

- `Deposit` — For more information, see “Deposit Name-Value Pair Arguments” on page 11-2994.
- `FRA` — For more information, see “FRA Name-Value Pair Arguments” on page 11-2835.
- `FixedBond` — For more information, see “FixedBond Name-Value Pair Arguments” on page 11-2750.
- `FixedBondOption` — For more information, see “FixedBondOption Name-Value Pair Arguments” on page 11-2770.
- `FloatBond` — For more information, see “FloatBond Name-Value Pair Arguments” on page 11-2786.
- `FloatBondOption` — For more information, see “FloatBondOption Name-Value Pair Arguments” on page 11-2804.
- `OptionEmbeddedFixedBond` — For more information, see “OptionEmbeddedFixedBond Name-Value Pair Arguments” on page 11-2860.
- `OptionEmbeddedFloatBond` — For more information, see “OptionEmbeddedFloatBond Name-Value Pair Arguments” on page 11-2884.

- `Swap` — For more information, see “Swap Name-Value Pair Arguments” on page 11-2917.
- `Cap` — For more information, see “Cap Name-Value Pair Arguments” on page 11-2710.
- `Floor` — For more information, see “Floor Name-Value Pair Arguments” on page 11-2818.
- `Swaption` — For more information, see “Swaption Name-Value Pair Arguments” on page 11-2946.
- `STIRFuture` — For more information, see “STIRFuture Name-Value Arguments” on page 11-2506.
- `OvernightIndexedSwap` — For more information, see “OvernightIndexedSwap Name-Value Arguments” on page 11-2529.
- `OISFuture` — For more information, see “OISFuture Name-Value Pair Arguments” on page 11-2495.
- `BondFuture` — For more information, see “BondFuture Name-Value Arguments” on page 11-3003.
- `ConvertibleBond` — For more information, see “ConvertibleBond Name-Value Pair Arguments” on page 11-2553.

Name-Value Pair Arguments for Inflation Instruments

- `InflationBond` — For more information, see “InflationBond Name-Value Pair Arguments” on page 11-2569.
- `YearYearInflationSwap` — For more information, see “YearYearInflationSwap Name-Value Pair Arguments” on page 11-2582.
- `ZeroCouponInflationSwap` — For more information, see “ZeroCouponInflationSwap Name-Value Pair Arguments” on page 11-2592.

Name-Value Pair Arguments for Equity Instruments

- `Vanilla` — For more information, see “Vanilla Name-Value Pair Arguments” on page 11-2966.
- `Lookback` — For more information, see “Lookback Name-Value Pair Arguments” on page 11-2844.
- `PartialLookback` — For more information, see “PartialLookback Name-Value Arguments” on page 11-2541.
- `Barrier` — For more information, see “Barrier Name-Value Pair Arguments” on page 11-2637.
- `DoubleBarrier` — For more information, see “DoubleBarrier Name-Value Pair Arguments” on page 11-2655.
- `Asian` — For more information, see “Asian Name-Value Pair Arguments” on page 11-2618.
- `Spread` — For more information, see “Spread Name-Value Pair Arguments” on page 11-2904.
- `VarianceSwap` — For more information, see “VarianceSwap Name-Value Pair Arguments” on page 11-2936.
- `Touch` — For more information, see “Touch Name-Value Pair Arguments” on page 11-2672.
- `DoubleTouch` — For more information, see “DoubleTouch Name-Value Pair Arguments” on page 11-2684.
- `Cliquet` — For more information, see “Cliquet Name-Value Arguments” on page 11-2515.
- `Binary` — For more information, see “Binary Name-Value Pair Arguments” on page 11-2696.
- `CommodityFuture` — For more information, see “CommodityFuture Name-Value Arguments” on page 11-3012.
- `EquityIndexFuture` — For more information, see “EquityIndexFuture Name-Value Arguments” on page 11-3023.

- `FXFuture` — For more information, see “FXFuture Name-Value Arguments” on page 11-3032.
- `ConvertibleBond` — For more information, see “ConvertibleBond Name-Value Pair Arguments” on page 11-2553.

Name-Value Pair Arguments for Credit Derivative Instruments

- `CDS` — For more information, see “CDS Name-Value Pair Arguments” on page 11-2727.
- `CDSOption` — For more information, see “CDSOption Name-Value Pair Arguments” on page 11-2736.

Output Arguments

`Instrument` — Instrument
instrument object

Instrument, returned as an instrument object.

Version History

Introduced in R2020a

See Also

`finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

finmodel

Create specified model object type

Syntax

```
Model = finmodel(ModelType,Name,Value)
```

Description

`Model = finmodel(ModelType,Name,Value)` creates a `Model` object based on `ModelType` creates a model object specified by `ModelType` and specifies model options using one or more name-value pair arguments.

For more information on the workflow for creating an instrument object, a model object, and a pricer object, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods, see “Choose Instruments, Models, and Pricers” on page 1-53.

Examples

Use `finmodel` to Create SABR Model

Use `finmodel` to create a SABR model object.

```
SabrModel = finmodel("SABR",'Alpha',0.032,'Beta',0.04, 'Rho', .08, 'Nu', 0043,'Shift',0.002)
```

```
SabrModel =
  SABR with properties:

             Alpha: 0.0320
              Beta: 0.0400
               Rho: 0.0800
                Nu: 43
             Shift: 0.0020
    VolatilityType: "black"
```

Input Arguments

`ModelType` — Model type

character vector | string

Model type, specified as a scalar string or character vector.

These options are available for interest-rate instruments:

- `"Black"` — For more information, see `Black`.
- `"HullWhite"` — For more information, see `HullWhite`.
- `"BlackKarasinski"` — For more information, see `BlackKarasinski`.
- `"BlackDermanToy"` — For more information, see `BlackDermanToy`.
- `"Normal"` — For more information, see `Normal`.
- `"SABR"` — For more information, see `SABR`.
- `"BraceGatarekMusiela"` — For more information, see `BraceGatarekMusiela`.
- `"SABRBraceGatarekMusiela"` — For more information, see `SABRBraceGatarekMusiela`.
- `"LinearGaussian2F"` — For more information, see `LinearGaussian2F`.

These options are available for equity instruments:

- `"BlackScholes"` — For more information, see `BlackScholes`.
- `"Bachelier"` — For more information, see `Bachelier`.
- `"Heston"` — For more information, see `Heston`.
- `"Bates"` — For more information, see `Bates`.
- `"Merton"` — For more information, see `Merton`.
- `"Dupire"` — For more information, see `Dupire`.

These options are available for credit derivative instruments:

- `"CDSBlack"` — For more information, see `CDSBlack`.

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Model = finmodel("Black",Name,Value)`

The available name-value pair arguments depend on the value you specify for `ModelType`.

Name-Value Pair Arguments for Interest-Rate Models

- `Black` — For more information, see “Black Name-Value Pair Arguments” on page 11-3048.
- `HullWhite` — For more information, see “HullWhite Name-Value Pair Arguments” on page 11-3115.
- `BlackKarasinski` — For more information, see “BlackKarasinski Name-Value Pair Arguments” on page 11-3096.
- `BlackDermanToy` — For more information, see “BlackDermanToy Name-Value Pair Arguments” on page 11-3103.
- `Normal` — For more information, see “Normal Name-Value Pair Arguments” on page 11-3125.
- `Sabr` — For more information, see “SABR Name-Value Pair Arguments” on page 11-3139

- `BraceGatarekMusiela` — For more information, see “BraceGatarekMusiela Name-Value Arguments” on page 11-3065
- `SABRBraceGatarekMusiela` — For more information, see “SABRBraceGatarekMusiela Name-Value Arguments” on page 11-3071
- `LinearGaussian2F` — For more information, see “LinearGaussian2F Name-Value Arguments” on page 11-3090

Name-Value Pair Arguments for Equity Models

- `BlackScholes` — For more information, see “BlackScholes Name-Value Pair Arguments” on page 11-3057.
- `Bachelier` — For more information, see “Bachelier Name-Value Pair Arguments” on page 11-3130.
- `Heston` — For more information, see “Heston Name-Value Pair Arguments” on page 11-3108.
- `Bates` — For more information, see “Bates Name-Value Pair Arguments” on page 11-3040.
- `Merton` — For more information, see “Merton Name-Value Pair Arguments” on page 11-3119.
- `Dupire` — For more information, see “Dupire Name-Value Pair Arguments” on page 11-3133.

Name-Value Pair Arguments for Credit Derivative Models

- `CDSBlack` — For more information, see “CDSBlack Name-Value Pair Arguments” on page 11-3052.

Output Arguments

`Model` — Model

model object

Model, returned as a model object.

Version History

Introduced in R2020a

See Also

`fininstrument` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

finpricer

Create pricing method

Syntax

```
Pricer = finpricer(PricerType,Name,Value)
```

Description

`Pricer = finpricer(PricerType,Name,Value)` creates a `Pricer` object based on `PricerType` creates a pricer object and specifies pricing options using one or more name-value pair arguments. The available name-value pair arguments depend on the `PricerType` you specify.

For more information on the workflow for creating an instrument object, a model object, and a pricer object, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods, see “Choose Instruments, Models, and Pricers” on page 1-53.

Examples

Use `finpricer` to Create `ConzeViswanathan` Pricer

This example shows the workflow to create a `BlackScholes` model and `ratecurve` object to use with a `ConzeViswanathan` pricing method.

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',.358)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3580
    Correlation: 1
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `ConzeViswanathan` Pricer Object

Use `finpricer` to create a `ConzeViswanathan` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',950,
```

```
outPricer =
  ConzeViswanathan with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 950
    DividendValue: 2.5000
     DividendType: "continuous"
```

Input Arguments

`PricerType` — Pricer type

character vector | string

Pricer type, specified as a scalar string or character vector.

These options are available for interest-rate instruments:

- `"Discount"` — For more information, see `Discount`.
- `"Future"` — For more information, see `Future`.
- `"IRTree"` — For more information, see `IRTree`.
- `"IRMonteCarlo"` — For more information, see `IRMonteCarlo`.
- `"HullWhite"` — For more information, see `HullWhite`.
- `"Analytic"` — The `"Analytic"` pricer can be any one of the following types of pricing methods:
  - `SABR` — For more information, see `SABR`.
  - `Normal` — For more information, see `Normal`.
  - `Black` — For more information, see `Black`.

These options are available for inflation instruments:

- `"Inflation"` — For more information, see `Inflation`.

These options are available for equity instruments:

- `"Analytic"` — The `"Analytic"` pricer can be any one of the following types of pricing methods:
  - `BlackScholes` — For more information, see `BlackScholes`.
  - `IkedaKunitomo` — For more information, see `IkedaKunitomo`.
  - `HeynenKat` — For more information, see `HeynenKat`.
  - `Heston` — For more information, see `Heston`.
  - `Levy` — For more information, see `Levy`.
  - `KemnaVorst` — For more information, see `KemnaVorst`.
  - `TurnbullWakeman` — For more information, see `TurnbullWakeman`.
  - `ConzeViswanathan` — For more information, see `ConzeViswanathan`.
  - `GoldmanSosinGatto` — For more information, see `GoldmanSosinGatto`.
  - `RollGeskeWhaley` — For more information, see `RollGeskeWhaley`.
  - `Rubinstein` — For more information, see `Rubinstein`.
  - `Kirk` — For more information, see `Kirk`.
  - `BjerksundStensland` — For more information, see `BjerksundStensland`.
- `"AssetTree"` — For more information, see `AssetTree`.
- `"AssetMonteCarlo"` — For more information, see `AssetMonteCarlo`.
- `"FiniteDifference"` — For more information, see `FiniteDifference`.
- `"FFT"` — For more information, see `FFT`.
- `"NumericalIntegration"` — For more information, see `NumericalIntegration`.
- `"VannaVolga"` — For more information, see `VannaVolga`.
- `"ReplicatingVarianceSwap"` — For more information, see `ReplicatingVarianceSwap`.
- `"Future"` — For more information, see `Future`.

These options are available for credit derivative instruments:

- `"Credit"` — For more information, see `Credit`.
- `"Analytic"` — The `"Analytic"` pricer can be any one of the following types of pricing methods:
  - `CDSBlack` — For more information, see `CDSBlack`.

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Pricer = finpricer("Black",Name,Value)`

Depending on the `PricerType`, the associated name-value pair arguments are different.

Name-Value Pair Arguments for Interest-Rate Pricers

- `IRTree` — For more information, see “IRTree Name-Value Pair Arguments” on page 11-3288.
- `IRMonteCarlo` — For more information, see “IRMonteCarlo Name-Value Arguments” on page 11-3276.
- `Black` — For more information, see “Black Name-Value Pair Arguments” on page 11-3199.
- `HullWhite` — For more information, see “HullWhite Name-Value Pair Arguments” on page 11-3272.
- `Normal` — For more information, see “Normal Name-Value Pair Arguments” on page 11-3332.
- `Sabr` — For more information, see “SABR Name-Value Pair Arguments” on page 11-3342.
- `Discount` — For more information, see “Discount Name-Value Pair Arguments” on page 11-3244.
- `Future` — For more information, see “Future Name-Value Pair Arguments” on page 11-3248.

Name-Value Pair Arguments for Inflation Pricers

- `Inflation` — For more information, see “Inflation Name-Value Pair Arguments” on page 11-2602.

Name-Value Pair Arguments for Equity Pricers

- `Levy` — For more information, see “Levy Name-Value Pair Arguments” on page 11-3324.
- `KemnaVorst` — For more information, see “KemnaVorst Name-Value Pair Arguments” on page 11-3312.
- `TurnbullWakeman` — For more information, see “TurnbullWakeman Name-Value Pair Arguments” on page 11-3354.
- `BlackScholes` — For more information, see “BlackScholes Name-Value Pair Arguments” on page 11-3203.
- `IkedaKunitomo` — For more information, see “IkedaKunitomo Name-Value Pair Arguments” on page 11-3164.
- `HeynenKat` — For more information, see “HeynenKat Name-Value Arguments” on page 11-3158.
- `Heston` — For more information, see “Heston Name-Value Pair Arguments” on page 11-3177.
- `ConzeViswanathan` — For more information, see “ConzeViswanathan Name-Value Pair Arguments” on page 11-3211.
- `GoldmanSosinGatto` — For more information, see “GoldmanSosinGatto Name-Value Pair Arguments” on page 11-3266.
- `Rubinstein` — For more information, see “Rubinstein Name-Value Pair Arguments” on page 11-2607.
- `RollGeskeWhaley` — For more information, see “RollGeskeWhaley Name-Value Pair Arguments” on page 11-3336.
- `Kirk` — For more information, see “Kirk Name-Value Pair Arguments” on page 11-3318.
- `BjerksundStensland` — For more information, see “BjerksundStensland Name-Value Pair Arguments” on page 11-3191.
- `AssetMonteCarlo` — For more information, see “AssetMonteCarlo Name-Value Pair Arguments” on page 11-3145.
- `AssetTree` — For more information, see “AssetTree Name-Value Pair Arguments” on page 11-3302.
- `FiniteDifference` — For more information, see “FiniteDifference Name-Value Pair Arguments” on page 11-3347.

- `FFT` — For more information, see “FFT Name-Value Pair Arguments” on page 11-3252.
- `NumericalIntegration` — For more information, see “NumericalIntegration Name-Value Pair Arguments” on page 11-3230.
- `VannaVolga` — For more information, see “Required VannaVolga Name-Value Pair Arguments” on page 11-3170.
- `ReplicatingVarianceSwap` — For more information, see “ReplicatingVarianceSwap Name-Value Pair Arguments” on page 11-3183.
- `Future` — For more information, see “Future Name-Value Pair Arguments” on page 11-3248.

Name-Value Pair Arguments for Credit Derivative Pricers

- `Credit` — For more information, see “Credit Name-Value Pair Arguments” on page 11-3216.
- `CDSBlack` — For more information, see “CDSBlack Name-Value Pair Arguments” on page 11-3221.

Output Arguments

`Pricer` — Pricer

pricer object

Pricer, returned as a pricer object.

Version History

Introduced in R2020a

See Also

`fininstrument` | `finmodel`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

irbootstrap

Bootstrap interest-rate curve from market data

Syntax

```
outCurve = irbootstrap(BootInstruments,Settle)
outCurve = irbootstrap( ___ ,Name,Value)
```

Description

`outCurve = irbootstrap(BootInstruments,Settle)` creates a data structure for storing interest-rate term structure data. The `outCurve` output is a `ratecurve` object.

`outCurve = irbootstrap( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any of the input argument combinations in the previous syntax. For example, `OutCurve = irbootstrap(Settle,BootInstruments,'Type',"zero",'Compounding',2,'Basis',5,'InterpMethod',"cubic")` bootstraps a zero curve from `BootInstruments`.

Examples

Bootstrap `ratecurve` Object from Market Data

Define Deposit and Swap Parameters

```
Settle = datetime(2018,3,21);
DepRates = [.0050769 .0054934 .0061432 .0072388 .0093263]';
DepTimes = [1 2 3 6 12]';
DepDates = datemnth(Settle,DepTimes);
nDeposits = length(DepTimes);

SwapRates = [.0112597 0;.0128489 0;.0138917 0;.0146135 0;.0151175 0;...
      .0155184 0;.0158536 0;.0161435 0];
SwapTimes = (2:9)';
SwapDates = datemnth(Settle,12*SwapTimes);
nSwaps = length(SwapTimes);

nInst = nDeposits + nSwaps;
```

Create a Vector of Market Swap Instruments

Use `fininstrument` to create a vector of market `Deposit` and `Swap` instrument objects.

```
BootInstruments(nInst,1) = fininstrument.FinInstrument;
for ii=1:length(DepDates)
     BootInstruments(ii) = fininstrument("deposit",'Maturity',DepDates(ii),'Rate',DepRates(ii))
end

for ii=1:length(SwapDates)
     BootInstruments(ii+nDeposits) = fininstrument("swap",'Maturity',SwapDates(ii),'LegRate',[Sw
end
```

Create `ratecurve` Object for Zero-Rate Curve

Use `irbootstrap` to create a `ratecurve` object for the zero-rate curve.

```
ZeroCurve = irbootstrap(BootInstruments,Settle)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [13x1 datetime]
               Rates: [13x1 double]
              Settle: 21-Mar-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Zero Curve Using `DiscountCurve` Input

This example shows how to create a `ratecurve` object using `irbootstrap` with `Deposit` and `Swap` instruments and a `DiscountCurve` for discounting cash flows.

```
Settle = datetime(2021,4,15);
crvDates = Settle + [calmonths([1 2 3 6]) calyears([1 2 3 5 7 10 20 30])]';
crvRates = [0.0004 0.0004 0.0005 0.0006 0.0008 0.0018 0.0037 0.0088 0.013 0.0165 0.022 0.0233]';
DiscountCurve = ratecurve("zero",Settle,crvDates,crvRates);

DepRates = [0.002 0.0021 0.0023 0.0024 .0028]';
DepDates = Settle + calmonths([1 2 3 6 12]');

SwapRates = [0.0041 0;0.0057 0;0.017 0;0.0193 0;0.024 0;0.027 0];
SwapTimes = [2 3 5 10 20 30]';
SwapDates = datemnth(Settle,12*SwapTimes);

BootInstruments = [fininstrument("deposit","Maturity",DepDates,"Rate",DepRates); ...
    fininstrument("swap","Maturity",SwapDates,"LegRate",SwapRates)];

ZeroCurve = irbootstrap(BootInstruments,Settle,'DiscountCurve',DiscountCurve)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [11x1 datetime]
               Rates: [11x1 double]
              Settle: 15-Apr-2021
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Bootstrap ratecurve Object from BootInstruments for OIS Futures and Overnight Indexed Swaps

Create a `BootInstruments` variable as an input argument to `irbootstrap` to create a ratecurve object. The `BootInstruments` variable has `OISFuture` instrument objects for one-month SOFR Futures and three-month SOFR Futures, and an `OvernightIndexedSwap` instrument object.

Create Instruments

Use `fininstrument` to create an `OISFuture` instrument object for one-month SOFR Futures.

```
Settle = datetime(2021,3,4);
HFDates = datetime(2021,3,1) + caldays(0:3)';
HistFixing = timetable(HFDates,[0.02;0.04;0.04;0.02]);

% Data from the following: https://www.cmegroup.com/trading/interest-rates/stir/one-month-sofr_qu
Prices_1M = [99.97 99.96 99.95]';
Maturity_1M = lbusdate(2021,[3 4 5]',[],[],'datetime');
StartDate_1M = fbusdate(2021,[3 4 5]',[],[],'datetime');
FutInstruments_1M = fininstrument("OISFuture","Maturity",Maturity_1M ,"QuotedPrice",Prices_1M,"St
    'HistoricalFixing',HistFixing,'Name',"1MonthSOFRFuture")
```

```
FutInstruments_1M=3×1 object
  3x1 OISFuture array with properties:

    QuotedPrice
    Method
    Basis
    StartDate
    Maturity
    Notional
    BusinessDayConvention
    Holidays
    ProjectionCurve
    HistoricalFixing
    Name
```

Use `fininstrument` to create an `OISFuture` instrument object for three-month SOFR Futures.

```
% Data from the following: https://www.cmegroup.com/trading/interest-rates/stir/three-month-sofr_
Prices_3M = [99.92 99.895 99.84 99.74]';
Dates_3M_Maturity = thirdwednesday([6 9 12 3]',[2021 2021 2021 2022]','datetime');
Dates_3M_Start = thirdwednesday([3 6 9 12]',2021,'datetime');
FutInstruments_3M = fininstrument("OISFuture","Maturity",Dates_3M_Maturity,...
    "QuotedPrice",Prices_3M,"StartDate",Dates_3M_Start,'HistoricalFixing',HistFixing,'Name',"3Mon
```

Use `fininstrument` to create an `OvernightIndexedSwap` instrument object.

```
SOFRSwapRates = [.0023 0;.0064 0;.013 0;.017 0;.0175 0];
SOFRSwapTimes = [3 5 10 20 30];
SOFRSwapDates = datemnth(Settle,12*SOFRSwapTimes)';
SOFRSwapInstruments = fininstrument("OvernightIndexedSwap","Maturity",SOFRSwapDates,"LegRate",SOF
```

```
SOFRSwapInstruments=5×1 object
  5x1 OvernightIndexedSwap array with properties:
```

```
    LegRate
    LegType
    Reset
    Basis
    Notional
    HistoricalFixing
    ResetOffset
    ProjectionCurve
    BusinessDayConvention
    Holidays
    EndMonthRule
    StartDate
    Maturity
    Name
```

Define `BootInstruments` for the three types of instruments.

```
BootInstruments = [FutInstruments_1M;FutInstruments_3M;SOFRSwapInstruments]
```

```
BootInstruments=12×1 object
  12x1 heterogeneous FinInstrument (OISFuture, OvernightIndexedSwap) array with properties:

    Name
```

Create `ratecurve` Object

Use `irbootstrap` to create a `ratecurve` object.

```
SOFRCurve = irbootstrap(BootInstruments,Settle)
```

```
SOFRCurve =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: [12x1 datetime]
             Rates: [12x1 double]
            Settle: 04-Mar-2021
      InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Bootstrap `ratecurve` Object from `STIRFuture`, `Deposit`, and `Swap` `BootInstruments`

Create `BootInstruments` for multiple `Deposit`, `STIRFuture`, and `Swap` instruments and then use `irbootstrap` to create and display a `ratecurve` object.

Create Instruments

Use `fininstrument` to create a `Deposit` instrument object.

```
Settle = datetime(2021,6,15);
DepRates = [0.0016 0.0017 .00175]';
```

```
DepDates = Settle + calmonths([1 2 3]');

Deposits = fininstrument("Deposit","Maturity",DepDates,"Rate",DepRates,'Name',"deposit_instrumen

Deposits=3×1 object
  3x1 Deposit array with properties:

    Rate
    Period
    Basis
    Maturity
    Principal
    BusinessDayConvention
    Holidays
    Name
```

Use `fininstrument` to create a `STIRFuture` instrument object.

```
FutureRates = [0.002 0.0025 0.0035]';
FutMat = [datetime(2021,9,15) datetime(2021,12,15) datetime(2022,3,16)]';
FutEndDates = [datetime(2021,12,15) datetime(2022,3,15) datetime(2022,6,15)]';

Futures = fininstrument("STIRFuture","Maturity",FutMat,"RateEndDate",FutEndDates,"QuotedPrice",10

Futures=3×1 object
  3x1 STIRFuture array with properties:

    QuotedPrice
    Basis
    RateEndDate
    Maturity
    Notional
    BusinessDayConvention
    Holidays
    ProjectionCurve
    Name
```

Use `fininstrument` to create a `Swap` instrument object.

```
SwapRates = [.0063 0;.0108 0;.013 0;.015 0;0.017 0;.018 0;0.019 0];
SwapTimes = [2 5 7 10 15 20 30]';
SwapDates = datemnth(Settle,12*SwapTimes);

Swaps = fininstrument("Swap","Maturity",SwapDates,"LegRate",SwapRates,'Name',"swap_instrument")

Swaps=7×1 object
  7x1 Swap array with properties:

    LegRate
    LegType
    Reset
    Basis
    Notional
    LatestFloatingRate
    ResetOffset
    DaycountAdjustedCashFlow
    ProjectionCurve
```

```
    BusinessDayConvention
    Holidays
    EndMonthRule
    StartDate
    Maturity
    Name
```

Define `BootInstruments` for the three instruments.

```
BootInstruments = [Deposits;Futures;Swaps];
```

Create `ratecurve` Object Using `irbootstrap`

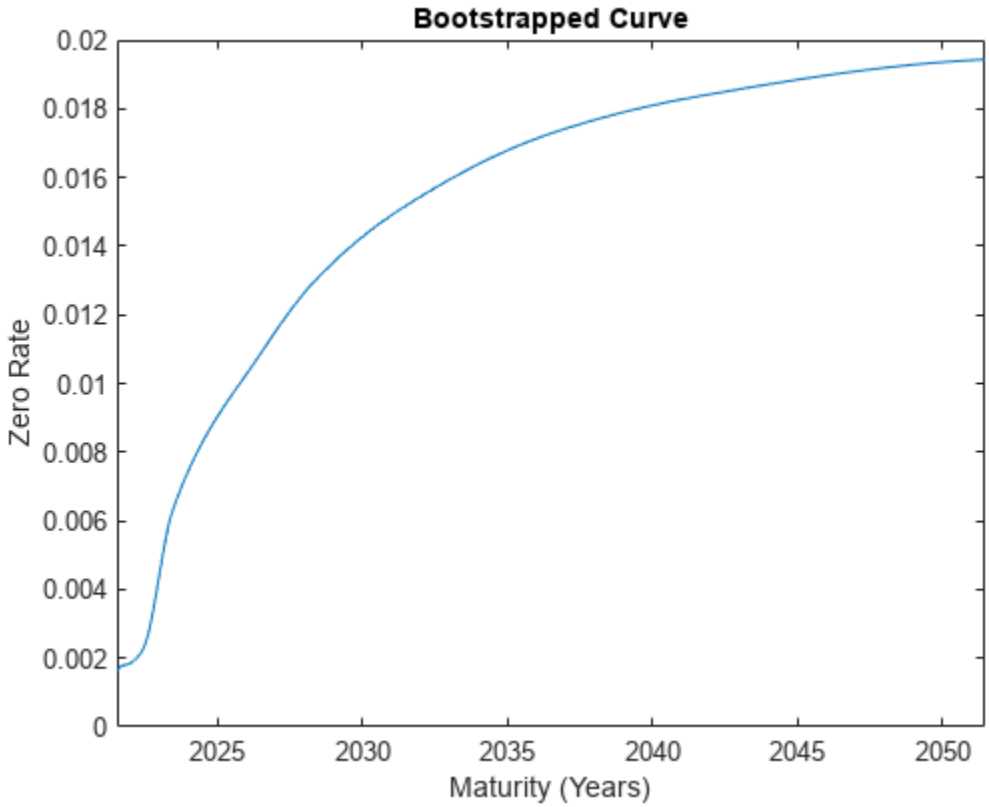
Use `irbootstrap` to create a `ratecurve` object.

```
ConvexityAdj = (1:3)'/10000;
ZeroCurve = irbootstrap(BootInstruments,Settle,'ConvexityAdjustment',ConvexityAdj,'InterpMethod'
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [13x1 datetime]
               Rates: [13x1 double]
              Settle: 15-Jun-2021
        InterpMethod: "pchip"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Plot Bootstrapped Curve

```
PlottingDates = Settle + calmonths(1:360);
plot(PlottingDates,zerorates(ZeroCurve,PlottingDates))
xlabel('Maturity (Years)')
ylabel('Zero Rate')
title('Bootstrapped Curve')
```



Input Arguments

`BootInstruments` — Collection of instruments

array of instrument objects

Collection of instruments, specified as an array of instrument objects. The collection of instruments can include `Deposit`, `Swap`, `FRA`, `STIRFuture`, `OISFuture`, and `OvernightIndexedSwap` instruments.

Data Types: `object`

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `irbootstrap` also accepts serial date numbers as inputs, but they are not recommended.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `OutCurve = irbootstrap(Settle,BootInstruments,'Type',"zero",'Compounding',2,'Basis',5,'InterpMethod',"cubic")`

Type — Type of interest-rate curve
`"zero"` (default) | string with value `"discount"`, `"forward"`, or `"zero"` | character vector with value `'discount'`, `'forward'`, or `'zero'`

Type of interest-rate curve, specified as the comma-separated pair consisting of `'Type'` and a scalar string or character vector.

Note When you use `irbootstrap`, the value you specify for `Type` can impact the curve construction because it affects the type of data that is interpolated on (that is, forward rates, zero rates, or discount factors) during the bootstrapping process.

Data Types: `char` | `string`

`Compounding` — Compounding frequency
`Compounding` for the `ratecurve` object (default) | possible values include: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

Compounding frequency, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using the supported values: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`InterpMethod` — Interpolation method
`"linear"` (default) | string with value `"linear"`, `"cubic"`, `"next"`, `"previous"`, `"pchip"`, `"v5cubic"`, `"makima"`, or `"spline"` | character vector with value `'linear'`, `'cubic'`, `'next'`, `'previous'`, `'pchip'`, `'v5cubic'`, `'makima'`, or `'spline'`

Interpolation method, specified as the comma-separated pair consisting of `'InterpMethod'` and a scalar string or character vector using a supported value. For more information on interpolation methods, see `interp1`.

Data Types: `char` | `string`

`ShortExtrapMethod` — Extrapolation method for data before first data
`"next"` (default) | string with value `"linear"`, `"next"`, `"previous"`, `"pchip"`, `"cubic"`,`"v5cubic"`, `"makima"`, or `"spline"` | character vector with value `'linear'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`,`'v5cubic'`, `'makima'`, or `'spline'`

Extrapolation method for data before first data, specified as the comma-separated pair consisting of `'ShortExtrapMethod'` and a scalar string or character vector using a supported value. For more information on interpolation methods, see `interp1`.

Data Types: `char` | `string`

`LongExtrapMethod` — Extrapolation method for data after last data
`"previous"` (default) | string with value `"linear"`, `"next"`, `"previous"`, `"pchip"`, `"cubic"`,`"v5cubic"`, `"makima"`, or `"spline"` | character vector with value `'linear'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`,`'v5cubic'`, `'makima'`, or `'spline'`

Extrapolation method for data after last data, specified as the comma-separated pair consisting of `'LongExtrapMethod'` and a scalar string or character vector using a supported value. For more information on interpolation methods, see `interp1`.

Data Types: `char` | `string`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve.empty` (default) | scalar `ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a previously created `ratecurve` object.

Data Types: `object`

`ConvexityAdjustment` — Convexity adjustment for `STIRFuture` instrument
`0` (default) | numeric vector

Convexity adjustment for one or more `STIRFuture` instruments, specified as the comma-separated pair consisting of `'ConvexityAdjustment'` and an `NFutures`-by-`1` vector of numeric values.

Note You can only use `ConvexityAdjustment` when using `irbootstrap` with a `STIRFuture` instrument. Also, the length of the `ConvexityAdjustment` vector must match the number of `STIRFuture` instruments.

Data Types: `double`

Output Arguments

`outCurve` — Rate curve

`ratecurve` object

Rate curve, returned as a `ratecurve` object. The object has the following properties:

- `Type`
- `Settle`
- `Compounding`
- `Basis`
- `Dates`
- `Rates`
- `InterpMethod`
- `ShortExtrapMethod`
- `LongExtrapMethod`

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `irbootstrap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`ratecurve` | `forwardrates` | `discountfactors` | `zerorates`

Topics

“Compute LIBOR Fallback” on page 2-192
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

fitNelsonSiegel

Fit Nelson-Siegel model to bond market data

Syntax

```
outCurve = fitNelsonSiegel(Settle,Instruments,CleanPrice)
```

Description

`outCurve = fitNelsonSiegel(Settle,Instruments,CleanPrice)` fits a Nelson-Siegel model to bond data.

Examples

Fit Nelson-Siegel Model to Bond Market Data

Define the bond data and use `fininstrument` to create `FixedBond` instrument objects.

```
 Settle = datetime(2017,9,15);
  Maturity = [datetime(2019,9,15);datetime(2021,9,15);...
      datetime(2023,9,15);datetime(2026,9,7);...
      datetime(2035,9,15);datetime(2047,9,15)];

  CleanPrice = [100.1;100.1;100.8;96.6;103.3;96.3];
  CouponRate = [0.0400;0.0425;0.0450;0.0400;0.0500;0.0425];

  nInst = numel(CouponRate);
Bonds(nInst,1) = fininstrument.FinInstrument;
for ii=1:nInst
    Bonds(ii) = fininstrument("FixedBond",'Maturity',Maturity(ii),...
        'CouponRate',CouponRate(ii));
end
```

Use `fitNelsonSiegel` to create a `parametercurve` object.

```
NSModel = fitNelsonSiegel(Settle,Bonds,CleanPrice)
```

```
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

NSModel =
  parametercurve with properties:

              Type: "zero"
            Settle: 15-Sep-2017
       Compounding: -1
             Basis: 0
    FunctionHandle: @(t)fitF(Params,t)
        Parameters: [3.5246e-08 0.0363 0.0900 16.5823]
```

Input Arguments

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `fitNelsonSiegel` also accepts serial date numbers as inputs, but they are not recommended.

`Instruments` — Bond instrument objects

array

Bond instrument objects, specified as an array of bond instruments objects.

Data Types: `object`

`CleanPrice` — Observed market prices

vector

Observed market prices, specified as a vector.

Data Types: `double`

Output Arguments

`outCurve` — Fitted Nelson-Siegel model

`parametercurve` object

Fitted Nelson-Siegel model, returned as a `parametercurve` object.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `fitNelsonSiegel` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`discountfactors` | `zerorates` | `forwardrates`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer.analytic`

Compute price for interest-rate, equity, or credit derivative instrument with `Analytic` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

The `Analytic` pricer supports the following pricer objects:

- `BjerksundStensland`
- `IkedaKunitomo`
- `Black`
- `BlackScholes`
- `CDSBlack`
- `ConzeViswanathan`
- `GoldmanSosinGatto`
- `HeynenKat`
- `HullWhite`
- `Heston`
- `KemnaVorst`
- `Kirk`
- `Levy`
- `Normal`
- `RollGeskeWhaley`
- `Rubinstein`
- `SABR`
- `TurnbullWakeman`

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities.

Examples

Use Bjerksund-Stensland Pricer and Black-Scholes Model to Price Spread Instrument

This example shows the workflow to price a European exercise `Spread` instrument when you use a `BlackScholes` model and a `BjerksundStensland` pricing method.

Create Spread Instrument Object

Use `fininstrument` to create a `Spread` instrument object.

```
SpreadOpt = fininstrument("Spread",'Strike',5,'ExerciseDate',datetime(2021,9,15),'OptionType',"pu
```

```
SpreadOpt =
  Spread with properties:

       OptionType: "put"
           Strike: 5
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2021
             Name: "spread_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',[0.2,0.1])
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: [0.2000 0.1000]
    Correlation: [2x2 double]
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                 Type: "zero"
          Compounding: -1
                Basis: 12
                Dates: 15-Sep-2023
                Rates: 0.0350
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BjerksundStensland Pricer Object

Use finpricer to create a BjerksundStensland pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',[100
```

```
outPricer =
  BjerksundStensland with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: [100 105]
    DividendValue: [0.0900 0.1700]
     DividendType: "continuous"
```

Price Spread Instrument

Use price to compute the price and sensitivities for the Spread instrument.

```
[Price, outPR] = price(outPricer,SpreadOpt,["all"])
```

```
Price = 7.0596
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price            Delta                    Gamma                    Lambda                Vega
    _____    ____________________    ____________________    ____________________    ____________
    7.0596    -0.23249     0.27057    0.0069887    0.0055319    -3.2932     3.8327    41.938
```

Use Rubinstein Pricer and Black-Scholes Model to Price the Absolure Return for Cliquet Instruments

This example shows the workflow to price the absolute return for three Cliquet instruments when you use a BlackScholes model and a Rubinstein pricing method.

Create ratecurve Object

Create a flat ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,Basis=12)
```

```
myRC =
  ratecurve with properties:
```

```
                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Cliquet` Instrument Object

Use `fininstrument` to create a `Cliquet` instrument object for three Cliquet instruments.

```
ResetDates = Settle + years(0:0.25:1);
CliquetOpt = fininstrument("Cliquet",ResetDates=ResetDates,InitialStrike=[140;150;160],ExerciseS
```

```
CliquetOpt=3×1 object
  3x1 Cliquet array with properties:

    OptionType
    ExerciseStyle
    ResetDates
    LocalCap
    LocalFloor
    GlobalCap
    GlobalFloor
    ReturnType
    InitialStrike
    Name
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",Volatility=0.28)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2800
    Correlation: 1
```

Create `Rubinstein` Pricer Object

Use `finpricer` to create a `Rubinstein` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",DiscountCurve=myRC,Model=BlackScholesModel,SpotPrice=135,Divide
```

```
outPricer =
  Rubinstein with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
```

```
        SpotPrice: 135
    DividendValue: 0.0250
     DividendType: "continuous"
```

Price Cliquet Instruments

Use `price` to compute the price and sensitivities for the three `Cliquet` instruments.

```
[Price, outPR] = price(outPricer,CliquetOpt,"all")
```

```
Price = 3×1

   28.1905
   25.3226
   23.8168


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Rho      Theta
    _____    ________    _________    ______    ______    _____    _____

    28.191    0.59697    0.020662    2.8588    105.38    60.643    -14.62


ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Rho       Theta
    _____    ________    _________    ______    ______    _____    ________

    25.323    0.41949    0.016816    2.2364    100.47    55.367    -11.708


ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Rho      Theta
    _____    ________    _________    ______    ______    _____    _____

    23.817    0.29729    0.011133    1.6851    93.219    51.616    -7.511
```

Input Arguments

`inpPricer` — Pricer object

`BjerksundStensland` object | `IkedaKunitomo` object | `Black` object | `BlackScholes` object | `CDSBlack` object | `ConzeViswanathan` object | `GoldmanSosinGatto` object | `HeynenKat` object | `HullWhite` object | `Heston` object | `KemnaVorst` object | `Kirk` object | `Levy` object | `Normal` object | `Rubinstein` object | `RollGeskeWhaley` object | `SABR` object | `TurnbullWakeman` object

Pricer object (previously created using `finpricer`), specified as a scalar. The supported pricer objects are:

- `BjerksundStensland`
- `IkedaKunitomo`
- `Black`
- `BlackScholes`
- `CDSBlack`
- `ConzeViswanathan`
- `GoldmanSosinGatto`
- `HeynenKat`
- `HullWhite`
- `Heston`
- `KemnaVorst`
- `Kirk`
- `Levy`
- `Normal`
- `RollGeskeWhaley`
- `SABR`
- `Rubinstein`
- `TurnbullWakeman`

Data Types: `object`

`inpInstrument` — Instrument object
`Cap` object | `Floor` object | `Swaption` object | `Vanilla` object | `Lookback` object | `PartialLookback` object | `Barrier` object | `DoubleBarrier` object | `Asian` object | `Spread` object | `Cliquet` object | `VarianceSwap` object | `CDSOption` object

Instrument object (previously created using `fininstrument`), specified as a scalar or a vector.

The supported instrument objects using a scalar or vector are:

- `Cap`
- `Floor`
- `Swaption`
- `Vanilla`
- `Lookback`
- `PartialLookback`
- `Barrier`
- `DoubleBarrier`
- `Asian`
- `Cliquet`
- `Spread`
- `CDSOption`

The supported instrument object using a scalar is:

- `VarianceSwap`

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute
`[ ]` (default) | string array with values dependent on pricer object | cell array of character vectors with values dependent on pricer object

(Optional) List of sensitivities to compute, specified as a `NOUT`-by-`1` or a `1`-by-`NOUT` cell array of character vectors or string array.

The supported sensitivities depend on the pricing method.

`inpPricer` Object	Supported Sensitivities
`BjerksundStensland`	`{'delta','gamma','vega', 'theta','rho','price','lambda'}`
`IkedaKunitomo`	`{'delta','gamma','vega','theta','rho', 'price','lambda'}`
`Black`	`'price'`
`BlackScholes`	`{'delta','gamma','vega','theta','rho', 'price','lambda'}`
`CDSBlack`	`'price'`
`ConzeViswanathan`	`{'delta','gamma','vega','theta','rho', 'price','lambda}'`
`GoldmanSosinGatto`	`{'delta','gamma','vega','theta','rho', 'price','lambda}'`
`HeynenKat`	`{'delta','gamma','vega','theta','rho', 'price','lambda}'`
`HullWhite`	`'price'`
`Heston`	`'price'`
`KemnaVorst`	`{'delta','gamma','vega','theta','rho', 'price','lambda'}`
`Kirk`	`{'delta','gamma','vega','theta','rho', 'price','lambda'}`
`Levy`	`{'delta','gamma','vega','theta','rho', 'price','lambda'}`
`Normal`	`'price'`
`RollGeskeWhaley`	`{'delta','gamma','vega','theta','rho', 'price','lambda'}`
`Rubinstein`	`{'delta','gamma','vega','theta','rho', 'price','lambda'}`
`SABR`	`'price'`
`TurnbullWakeman`	`{'delta','gamma','vega','theta','rho', 'price',}`

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that all sensitivities for the pricing method are returned. This is the same as specifying `inpSensitivity` to include each sensitivity.

Example: `inpSensitivity = ["delta","gamma","vega","lambda","rho","theta","price"]`

Data Types: `cell` | `string`

Output Arguments

`Price` — Instrument price
numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result
`PriceResult` object

Price result, returned as a `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes sensitivities (if you specify `inpSensitivity`)
- `PriceResult.PricerData` — Structure for pricer data

> **Note** When pricing a `VarianceSwap`, `PriceResult.FairVariance` is returned.

Note The `inpPricer` options that do not support sensitivities do not return a `PriceResult`. For example, there is no `PriceResult` returned for when using a `Black`, `CDSBlack`, `HullWhite`, `Normal`, `Heston`, or `SABR` pricing method.

Version History

Introduced in R2020a

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for equity instrument with `VannaVolga` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities.

Examples

TBD

Input Arguments

`inpPricer` — Pricer object
`VannaVolga` object

Pricer object, specified as a scalar `VannaVolga` pricer object. Use `finpricer` to create the `VannaVolga` pricer object.

Data Types: `object`

`inpInstrument` — Instrument object
`Vanilla` object | `Barrier` object | `DoubleBarrier` object | `Touch` object

Instrument object, specified as a scalar or vector of `Vanilla`, `Barrier`, `DoubleBarrier`, `Touch`, or `DoubleTouch` instrument objects. Use `fininstrument` to create the `Vanilla`, `Barrier`, `DoubleBarrier`, `Touch`, or `DoubleTouch`instrument objects.

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute
`[ ]` (default) | string array with values `"Price"`, `"Delta"`, `"Gamma"`, `"Vega"`, `"Rho"`, `"Theta"`, `'Lambda'`, and `"All"` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Lambda'`, `'Vega'`, `'Rho'`, `'Theta'`, and `'All'`

(Optional) List of sensitivities to compute, specified as a `NOUT`-by-`1` or a `1`-by-`NOUT` cell array of character vectors or string array with supported values.

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that the output is `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'Price'`. This is the same as specifying `inpSensitivity` to include each sensitivity.

Example: `inpSensitivity = {'delta','gamma','vega','rho','lambda','theta','price'}`

The sensitivities supported depend on the `inpInstrument`.

inpInstrument	Supported Sensitivities
`Vanilla`,	`'delta','gamma','vega','rho','lambda', 'theta','price'`
`Barrier`	`'delta','gamma','vega','rho','lambda', 'theta','price'`
`DoubleBarrier`	`'delta','gamma','vega','rho','lambda', 'theta','price'`
`Touch`	`'delta','gamma','vega','rho','lambda', 'theta','price'`
`DoubleTouch`	`'delta','gamma','vega','rho','lambda', 'theta','price'`

Data Types: `string` | `cell`

Output Arguments

`Price` — Instrument price

numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result

`PriceResult` object

Price result, returned as a `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes sensitivities (if you specify `inpSensitivity`)
- `PriceResult.PricerData` — Structure for pricer data
- `PriceResult.PricerData.Overhedge` — TBD

Version History

Introduced in R2020b

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22

“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for equity instrument with `AssetMonteCarlo` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the equity instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities. Use this syntax with the input argument combination in the previous syntax.

Examples

Price Touch Instrument Using Heston Model and Asset Monte Carlo Pricer

This example shows the workflow to price a `Touch` instrument when you use a `Heston` model and an `AssetMonteCarlo` pricing method.

Create Touch Instrument Object

Use `fininstrument` to create a `Touch` instrument object.

```
TouchOpt = fininstrument("Touch",'ExerciseDate',datetime(2022,9,15),'BarrierValue',110,'PayoffVa
```

```
TouchOpt =
  Touch with properties:

    ExerciseDate: 15-Sep-2022
    BarrierValue: 110
     PayoffValue: 140
     BarrierType: "ot"
      PayoffType: "expiry"
            Name: "touch_option"
```

Create Heston Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9)
```

```
HestonModel =
  Heston with properties:
```

```
        V0: 0.0320
    ThetaV: 0.1000
     Kappa: 0.0030
    SigmaV: 0.2000
     RhoSV: 0.9000
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
              Dates: 15-Sep-2023
              Rates: 0.0350
             Settle: 15-Sep-2018
       InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",HestonModel,'SpotPrice',112
```

```
outPricer =
  HestonMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 112
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Heston]
       DividendType: "continuous"
      DividendValue: 0
```

Price Touch Instrument

Use `price` to compute the price and sensitivities for the `Touch` instrument.

```
[Price, outPR] = price(outPricer,TouchOpt,["all"])
```

```
Price = 63.5247
```

```
outPR =
  priceresult with properties:

       Results: [1x8 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×8 table
    Price      Delta     Gamma     Lambda       Rho       Theta      Vega     VegaLT
    _____    ________    _____    ________    ________    _____    _____    _____

    63.525    -7.2363    1.0541    -12.758    -320.21    3.5527    418.94    8.1498
```

Input Arguments

inpPricer — Pricer object

`AssetMonteCarlo` object

Pricer object, specified as a previously created `AssetMonteCarlo` pricer object. Create the pricer object using `finpricer`.

Data Types: `object`

inpInstrument — Instrument object

`Vanilla` object | `Barrier` object | `Lookback` object | `Asian` object | `DoubleBarrier` object | `Spread` object | `Touch` object | `DoubleTouch` object | `Cliquet` object | `Binary` object

Instrument object, specified as a scalar or vector of previously created instrument objects. Create the instrument objects using `fininstrument`. The following instrument objects are supported:

- `Vanilla`
- `Lookback`
- `Barrier`
- `Asian`
- `Spread`
- `DoubleBarrier`
- `Cliquet`
- `Binary`
- `Touch`
- `DoubleTouch`

Data Types: `object`

inpSensitivity — List of sensitivities to compute

`[ ]` (default) | string array with values dependent on pricer object | cell array of character vectors with values dependent on pricer object

(Optional) List of sensitivities to compute, specified as an `NOUT`-by-`1` or `1`-by-`NOUT` cell array of character vectors or string array.

The supported sensitivities depend on the pricing method.

inpInstrument Object	Supported Sensitivities
Vanilla	{'delta','gamma','vega', 'theta','rho','price','lambda'}
Lookback	{'delta','gamma','vega','theta','rho', 'price','lambda'}
Barrier	{'delta','gamma','vega','theta','rho', 'price','lambda'}
Asian	{'delta','gamma','vega','theta','rho', 'price','lambda'}
Spread	{'delta','gamma','vega','theta','rho', 'price','lambda}'
DoubleBarrier	{'delta','gamma','vega','theta','rho', 'price','lambda}'
Cliquet	{'delta','gamma','vega','theta','rho', 'price','lambda}'
Binary	{'delta','gamma','vega','theta','rho', 'price','lambda'}
Touch	{'delta','gamma','vega','theta','rho', 'price','lambda'}
DoubleTouch	{'delta','gamma','vega','theta','rho', 'price','lambda'}

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that all sensitivities for the pricing method are returned. This is the same as specifying `inpSensitivity` to include each sensitivity.

Example: `inpSensitivity = ["delta","gamma","vega","lambda","rho","theta","price"]`

Data Types: `cell` | `string`

Output Arguments

`Price` — Instrument price

numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result

`PriceResult` object

Price result, returned as a `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes sensitivities (if you specify `inpSensitivity`)
- `PriceResult.PricerData` — Structure for pricer data

Note The `inpPricer` options that do not support sensitivities do not return a `PriceResult`. For example, there is no `PriceResult` returned for when you use a `Black`, `CDSBlack`, `HullWhite`, `Normal`, or SABR pricing method.

Version History

Introduced in R2020b

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for interest-rate instrument with `IRMonteCarlo` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the interest-rate instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities. Use this syntax with the input argument combination in the previous syntax.

Examples

Price Fixed Bond Instrument Using Hull-White Model and IRMonteCarlo Pricer

This example shows the workflow to price a `FixedBond` instrument when using a `HullWhite` model and an `IRMonteCarlo` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond","Maturity",datetime(2022,9,15),"CouponRate",0.05,'Name',"fixed_b
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond"
```

Create HullWhite Model Object

Use finmodel to create a HullWhite model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.32,'Sigma',0.49)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.3200
    Sigma: 0.4900
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use finpricer to create an IRMonteCarlo pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",'Model',HullWhiteModel,'DiscountCurve',myRC,'SimulationDates
```

```
outPricer =
  HWMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jan-2021    ...    ]
              Model: [1x1 finmodel.HullWhite]
```

Price FixedBond Instrument

Use price to compute the price and sensitivities for the FixedBond instrument.

```
[Price,outPR] = price(outPricer,FixB,["all"])

Price = 115.0303

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×4 table
    Price      Delta     Gamma     Vega
    _____    ________    _____    ____

    115.03    -397.13    1430.4     0
```

Input Arguments

`inpPricer` — Pricer object

`IRMonteCarlo` object

Pricer object, specified as a previously created `IRMonteCarlo` pricer object. Create the pricer object using `finpricer`.

Data Types: `object`

`inpInstrument` — Instrument object

`Cap` object | `Floor` object | `Swaption` object | `Swap` object | `FixedBond` object | `OptionEmbeddedFixedBond` object | `OptionEmbeddedFloatBond` object | `FixedBondOption` object | `FloatBond` object | `FloatBondOption` object

Instrument object, specified as scalar or a vector of previously created instrument objects. Create the instrument objects using `fininstrument`. The following instrument objects are supported:

- `Cap`
- `Floor`
- `Swap`
- `Swaption`
- `FixedBond`
- `OptionEmbeddedFixedBond`
- `FixedBondOption`
- `FloatBond`
- `FloatBondOption`
- `OptionEmbeddedFloatBond`

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute
`[ ]` (default) | string array with values dependent on pricer object | cell array of character vectors with values dependent on pricer object

(Optional) List of sensitivities to compute, specified as an `NOUT`-by-`1` or `1`-by-`NOUT` cell array of character vectors or string array.

The supported sensitivities depend on the pricing method.

inpInstrument	Supported Sensitivities
`Cap`	`{'delta','gamma','vega','price'}` (`'vega'` not supported when using `SABRBraceGatarekMusiela` model with the `IRMonteCarlo` pricer.)
`Floor`	`{'delta','gamma','vega','price'}` (`'vega'` not supported when using `SABRBraceGatarekMusiela` model with the `IRMonteCarlo` pricer.)
`Swap`	`{'delta','gamma','vega','price'}`
`Swaption`	`{'delta','gamma','vega','price'}`
`FixedBond`	`{'delta','gamma','vega','price'}` (`'vega'` not supported when using `SABRBraceGatarekMusiela` model with the `IRMonteCarlo` pricer.)
`OptionEmbeddedFixedBond`	`{'delta','gamma','vega','price'}` (`'vega'` not supported when using `SABRBraceGatarekMusiela` model with the `IRMonteCarlo` pricer.)
`FixedBondOption`	`{'delta','gamma','vega','price'}` (`'vega'` not supported when using `SABRBraceGatarekMusiela` model with the `IRMonteCarlo` pricer.)
`FloatBond`	`{'delta','gamma','vega','price'}` (`'vega'` not supported when using `SABRBraceGatarekMusiela` model with the `IRMonteCarlo` pricer.)
`FloatBondOption`	`{'delta','gamma','vega','price'}` (`'vega'` not supported when using `SABRBraceGatarekMusiela` model with the `IRMonteCarlo` pricer.)
`OptionEmbeddedFloatBond`	`{'delta','gamma','vega','price'}` (`'vega'` not supported when using `SABRBraceGatarekMusiela` model with the `IRMonteCarlo` pricer.)

`inpSensitivity = 'All'` or `inpSensitivity = "All"` specifies that all sensitivities for the pricing method are returned. This is the same as specifying `inpSensitivity` to include each sensitivity.

Example: `inpSensitivity = ["delta","gamma","vega","price"]`

Data Types: `cell` | `string`

Output Arguments

`Price` — Instrument price

numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result

`PriceResult` object

Price result, returned as a `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes sensitivities (if you specify `inpSensitivity`)
- `PriceResult.PricerData` — Structure for pricer data

Version History

Introduced in R2021b

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for interest-rate instrument with `IRTree` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities.

Examples

Use Hull-White Tree Pricer and Hull-White Model to Price FixedBondOption Instrument

This example shows the workflow to price a `FixedBondOption` instrument when you use a `HullWhite` model and an `IRTree` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",'Maturity',datetime(2029,9,15),'CouponRate',0.025,'Period',
```

```
BondInst =
  FixedBond with properties:

                  CouponRate: 0.0250
                      Period: 1
                       Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2029
                        Name: "fixed_bond_instrument"
```

Create FixedBondOption Instrument Object

Use `fininstrument` to create a `FixedBondOption` instrument object.

```
FixedBOption = fininstrument("FixedBondOption",'ExerciseDate',datetime(2025,9,15),'Strike',98,'Bo
```

```
FixedBOption =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2025
           Strike: 98
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option_instrument"
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calyears([1:10])]';
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.01,'Sigma',0.05)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0100
    Sigma: 0.0500
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("irtree",'Model',HullWhiteModel,'DiscountCurve',myRC,'TreeDates',ZeroDa
```

```
HWTreePricer =
  HWBKTree with properties:

             Tree: [1x1 struct]
        TreeDates: [10x1 datetime]
            Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

```
HWTreePricer.Tree
```

```
ans = struct with fields:
        tObs: [0 1 1.9973 2.9945 3.9918 4.9918 5.9891 6.9863 7.9836 8.9836]
        dObs: [15-Sep-2019    15-Sep-2020    15-Sep-2021    ...    ]
      CFlowT: {1x10 cell}
       Probs: {1x9 cell}
     Connect: {1x9 cell}
     FwdTree: {1x10 cell}
    RateTree: {1x10 cell}
```

Price FixedBondOption Instrument

Use `price` to compute the price and sensitivities for the `FixedBondOption` instrument.

```
[Price, outPR] = price(HWTreePricer,FixedBOption,["all"])
```

```
Price = 11.1739
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma      Vega
    ______    _______    ______    ______

    11.174    -272.19    3667.6    243.09
```

Input Arguments

inpPricer — Pricer object

IRTree object

Pricer object, specified as a scalar `IRTree` pricer object. Use `finpricer` to create the `IRTree` pricer object.

Data Types: `object`

`inpInstrument` — Instrument object
`Cap` object | `Floor` object | `Swaption` object | `FixedBond` object | `OptionEmbeddedFixedBond` object | `OptionEmbeddedFloatBond` object | `FixedBondOption` object | `FloatBond` object | `FloatBondOption` object

Instrument object, specified as scalar or a vector of previously created instrument objects. Create the instrument objects using `fininstrument`. The following instrument objects are supported:

- `Cap`
- `Floor`
- `Swaption`
- `FixedBond`
- `OptionEmbeddedFixedBond`
- `FixedBondOption`
- `FloatBond`
- `FloatBondOption`
- `OptionEmbeddedFloatBond`

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute
`[ ]` (default) | string array with values `"Price"`, `"Delta"`, `"Gamma"`, `"Vega"`, and `"All"` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, and `'All'`

(Optional) List of sensitivities to compute, specified as a `NOUT`-by-`1` or a `1`-by-`NOUT` cell array of character vectors or string array with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, and `'All'`.

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that the output is `'Delta'`, `'Gamma'`, `'Vega'`, and `'Price'`. This is the same as specifying `inpSensitivity` to include each sensitivity.

The sensitivities supported depend on the `inpInstrument`.

inpInstrument	Supported Sensitivities
`Cap`	`{'delta','gamma','vega','price'}`
`Floor`	`{'delta','gamma','vega','price'}`
`Swaption`	`{'delta','gamma','vega','price'}`
`FixedBond`	`{'delta','gamma','vega','price'}`
`OptionEmbeddedFixedBond`	`{'delta','gamma','vega','price'}`
`FixedBondOption`	`{'delta','gamma','vega','price'}`
`FloatBond`	`{'delta','gamma','vega','price'}`
`FloatBondOption`	`{'delta','gamma','vega','price'}`
`OptionEmbeddedFloatBond`	`{'delta','gamma','vega','price'}`

Note Sensitivities are calculated based on yield shifts of 1 basis point, where the ShiftValue = 1/10000. All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide the sensitivities by their respective instrument price.

Example: `inpSensitivity = {'delta','gamma','vega','price'}`

Data Types: `string` | `cell`

Output Arguments

`Price` — Instrument price

numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result

`PriceResult` object

Price result, returned as a `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes sensitivities (if you specify `inpSensitivity`)
- `PriceResult.PricerData` — Structure for pricer data that depends on the instrument that is being priced

  `FixedBond`, `FloatBond`, `FixedBondOption`, and `OptionEmbeddedFixedBond` have the following shared fields for `PriceResult.PricerData.PriceTree`:

  - `tObs` contains the observation times.
  - `Connect` contains the connectivity vectors. Each element in the cell array describes how nodes in that level connect to the next. For a given tree level, there are `NumNodes` elements in the vector, and they contain the index of the node at the next level that the middle branch connects to. Subtracting 1 from that value indicates where the up-branch connects to, and adding 1 indicates where the down branch connects to.

The following additional fields for `PriceResult.PricerData.PriceTree` depend on the instrument type:

- `PTree` contains the clean prices.
- `AITree` contains the accrued interest.
- `Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.
- `dObs` contains the date of each level of the tree.
- `CFlowT` is a cell array with as many elements as levels of the tree. Each cell array element contains the time factors (`tObs`) corresponding to its level of the tree and those levels ahead of it.
- `FwdTree` contains the forward spot rate from one node to the next. The forward spot rate is defined as the inverse of the discount factor.
- `ExTree` contains the exercise indicator arrays. Each element of the cell array is an array containing `1`'s where an option is exercised and `0`'s where it isn't.
- `ProbTree` contains the probability of reaching each node from root node.

- `ExProbTree` contains the exercise probabilities. Each element in the cell array is an array containing `0`'s where there is no exercise, or the probability of reaching that node where exercise happens.
- `ExProbsByTreeLevel` is an array with each row holding the exercise probability for a given option at each tree observation time.

A `FixedBond` instrument has these additional fields within `PriceResult.PricerData.PriceTree`:

- `PTree`
- `AITree`
- `Probs`.

A `FloatBond` instrument has these additional fields within `PriceResult.PricerData.PriceTree`:

- `dObs`
- `CFlowT`
- `Probs`
- `FwdTree`

A `FixedBondOption` instrument has these additional fields within `PriceResult.PricerData.PriceTree`:

- `PTree`
- `Probs`
- `ExTree`

A `OptionEmbeddedFixedBond` instrument has these additional fields within `PriceResult.PricerData.PriceTree`:

- `PTree`
- `ExTree`
- `ProbTree`
- `ExProbTree`
- `ExProbsByTreeLevel`

The following table displays the `PriceResult.PricerData.PriceTree` fields related to each instrument.

PriceResult.PricerData.PriceTree Fields	**FixedBond**	**FloatBond**	**FixedBondOption**	**OptionEmbeddedFixedBond**
`tObs`	✓	✓	✓	✓
`Connect`	✓	✓	✓	✓
`PTree`	✓	No	✓	✓
`AITree`	✓	No	No	No
`Probs`	✓	✓	✓	No

PriceResult.PricerData.PriceTree Fields	FixedBond	FloatBond	FixedBondOption	OptionEmbeddedFixedBond
dObs	No	✓	No	No
CFlowT	No	✓	No	No
FwdTree	No	✓	✓	✓
ExTree	No	No	✓	✓
ProbTree	No	No	No	✓
ExProbTree	No	No	No	✓
ExProbsByTreeLevel	No	No	No	✓

Version History

Introduced in R2020a

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for equity instrument with `AssetTree` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the equity instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities in addition to the required arguments in the previous syntax.

Examples

Use Leisen-Reimer Tree Pricer and Black-Scholes Model to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2019,5,1),'Strike',29,'OptionType','
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 01-May-2019
           Strike: 29
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:
```

```
     Volatility: 0.2500
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2020,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2020
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object for an LR equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
LRPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',30,'
```

```
LRPricer =
  LRTree with properties:

    InversionMethod: PP1
             Strike: 30
               Tree: [1x1 struct]
         NumPeriods: 15
              Model: [1x1 finmodel.BlackScholes]
      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 30
       DividendType: "continuous"
      DividendValue: 0
          TreeDates: [02-Feb-2018 08:00:00    06-Mar-2018 16:00:00    ...    ]
```

Price Vanilla Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(LRPricer,VanillaOpt,"all")
```

```
Price = 2.2542
```

```
outPR =
  priceresult with properties:
```

```
       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Vega      Lambda      Rho       Theta
    ______    ________    ________    ______    ______    _______    ________

    2.2542    -0.33628    0.044039    12.724    -4.469    -16.433    -0.76073
```

Use Standard Trinomial Tree Pricer and Black-Scholes Model to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method for a Standard Trinomial (STT) tree.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2019,5,1),'Strike',29,'OptionType','
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 01-May-2019
           Strike: 29
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2500
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2020,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2020
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object for an Standard Trinomial equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
STTPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',30,
```

```
STTPricer =
  STTree with properties:

            Tree: [1x1 struct]
       NumPeriods: 15
           Model: [1x1 finmodel.BlackScholes]
    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 30
     DividendType: "continuous"
    DividendValue: 0
        TreeDates: [02-Feb-2018 08:00:00    06-Mar-2018 16:00:00    ...    ]
```

Price Vanilla Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(STTPricer,VanillaOpt,"all")
```

```
Price = 2.2826
```

```
outPR =
  priceresult with properties:

      Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Vega     Lambda       Rho       Theta
    ______    _______    ________    _____    _______    _______    ________
```

```
    2.2826    -0.2592    0.030949    12.51    -3.8981    -16.516    -0.73845
```

Input Arguments

`inpPricer` — Pricer object

`AssetTree` object

Pricer object, specified as a scalar `AssetTree` pricer object. Use `finpricer` to create the `AssetTree` pricer object.

Data Types: `object`

`inpInstrument` — Instrument object

`Vanilla` object | `Barrier` object | `Asian` object | `Lookback` object

Instrument object, specified as a scalar or vector of previously created instrument objects. Create the instrument objects using `fininstrument`. The following instrument objects are supported:

- `Vanilla`
- `Lookback`
- `Barrier`
- `Asian`

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute

`[ ]` (default) | string array with values `"Price"`, `"Delta"`, `"Gamma"`, `"Vega"`, `"Theta"`, `"Rho"`, `"Lambda"`, and `"All"` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Theta'`, `'Rho'`, `'Lambda'`, and `'All'`

(Optional) List of sensitivities to compute, specified as an `NOUT`-by-`1` or a `1`-by-`NOUT` cell array of character vectors or string array with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Theta'`, `'Rho'`, `'Lambda'`, and `'All'`.

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that the output is `'Delta'`, `'Gamma'`, `'Vega'`, `'Theta'`, `'Rho'`, `'Lambda'`, and `'Price'`. Using this syntax is the same as specifying `inpSensitivity` to include each sensitivity.

inpInstrument	Supported Sensitivities
`Asian`	`{'delta','gamma','vega','theta','rho','lambda','price'}`
`Barrier`	`{'delta','gamma','vega','theta','rho','lambda','price'}`
`Lookback`	`{'delta','gamma','vega','theta','rho','lambda','price'}`
`Vanilla`	`{'delta','gamma','vega','theta','rho','lambda','price'}`

Note Sensitivities are calculated based on yield shifts of 1 basis point, where the ShiftValue = 1/10000. All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide the sensitivities by their respective instrument price.

Example: `inpSensitivity = {'delta','gamma','vega','price'}`

Data Types: `string` | `cell`

Output Arguments

`Price` — Instrument price

numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result

`PriceResult` object

Price result, returned as a `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes sensitivities (if you specify `inpSensitivity`)
- `PriceResult.PricerData` — Structure for pricer data that depends on the instrument that is being priced

  `Asian` and `Lookback` have an empty (`[]`) `PricerData` field because the pricing functions for these instruments cannot unambiguously assign a price to any node but the root node.

  `Vanilla` and `Barrier` have the following shared fields for `PriceResult.PricerData.PriceTree`:

  - `PTree` contains the clean prices.
  - `ExTree` contains the exercise indicator arrays. Each element of the cell array is an array where `1` indicates that an option is exercised and `0` indicates that an option is not exercised.
  - `dObs` contains the date of each level of the tree.
  - `tObs` contains the observation times.
  - `Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.

Version History

Introduced in R2021a

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Use treeviewer to Examine HWTree and PriceTree When Pricing European Callable Bond” on page 2-194
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22

“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: finpricer

Compute price for credit derivative instrument with Credit pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
```

Description

[Price,PriceResult] = price(inpPricer,inpInstrument) computes the instrument price and related pricing information based on the pricing object inpPricer and the instrument object inpInstrument.

Examples

Use Credit Pricer and Default Probability Curve to Price CDS Instrument

This example shows the workflow to price a CDS instrument when you use a defprobcurve model and a Credit pricing method.

Create CDS Instrument Object

Use fininstrument to create a CDS instrument object.

```
CDS = fininstrument("cds",'Maturity',datetime(2027,9,20),'ContractSpread', 50,'Name',"CDS_instru
```

```
CDS =
  CDS with properties:

           ContractSpread: 50
                 Maturity: 20-Sep-2027
                   Period: 4
                    Basis: 2
              RecoveryRate: 0.4000
    BusinessDayConvention: "actual"
                 Holidays: NaT
        PayAccruedPremium: 1
                 Notional: 10000000
                     Name: "CDS_instrument"
```

Create defprobcurve Object

Create a defprobcurve object using defprobcurve.

```
Settle = datetime(2017, 9, 20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                  Settle: 20-Sep-2017
                   Basis: 2
                   Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                 Type: "zero"
          Compounding: -1
                Basis: 0
                Dates: [10x1 datetime]
                Rates: [10x1 double]
               Settle: 20-Sep-2017
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Credit` Pricer Object

Use `finpricer` to create a `Credit` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
CDSpricer = finpricer("credit",'DiscountCurve',ZeroCurve,'DefaultProbabilityCurve',DefaultProbCu
```

```
CDSpricer =
  Credit with properties:

              DiscountCurve: [1x1 ratecurve]
                   TimeStep: 10
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price CDS Instrument

Use `price` to compute the price for the `CDS` instrument.

```
outPrice = price(CDSpricer, CDS)
```

```
outPrice = 6.9363e+04
```

Input Arguments

`inpPricer` — Pricer object

`Credit` pricer object

Pricer object, specified as a scalar `Credit` pricer object. Use `finpricer` to create the `Credit` pricer object.

Data Types: `object`

`inpInstrument` — Instrument object
`CDS` object

Instrument object, specified as a scalar or vector of previously created `CDS` instrument objects. Use `fininstrument` to create `CDS` instrument objects.

Data Types: `object`

Output Arguments

`Price` — Instrument price
numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result
`PriceResult` object

Price result, returned as a `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes the `CDS` price and accrued premium
- `PriceResult.PricerData` — Structure that includes the premium leg payment dates, times, and cash flows

Version History

Introduced in R2020a

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for interest-rate instrument with `Discount` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities.

Examples

Use Discount Pricer and ratecurve to Price Swap Instrument

This example shows the workflow to price a `Swap` instrument when using a `ratecurve` and a `Discount` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `Swap` instrument.

```
Settle = datetime(2022,1,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: [10x1 datetime]
             Rates: [10x1 double]
            Settle: 15-Jan-2022
      InterpMethod: "linear"
  ShortExtrapMethod: "next"
```

```
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create a `Swap` instrument object.

```
SwapOpt = fininstrument("Swap",'Maturity',datetime(2027,1,15),'LegRate',[0.024 0.015],'LegType',
```

```
SwapOpt =
  Swap with properties:

                    LegRate: [0.0240 0.0150]
                    LegType: ["fixed"    "float"]
                      Reset: [2 2]
                      Basis: [0 0]
                   Notional: 100
          LatestFloatingRate: [NaN NaN]
                 ResetOffset: [0 0]
    DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [1x2 ratecurve]
       BusinessDayConvention: ["actual"    "actual"]
                   Holidays: NaT
                EndMonthRule: [1 1]
                   StartDate: NaT
                    Maturity: 15-Jan-2027
                        Name: "swap_instrument"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price Swap Instrument

Use `price` to compute the price and sensitivities for the `Swap` instrument.

```
[Price, outPR] = price(outPricer, SwapOpt,["all"])
```

```
Price = -1.3834
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
     Price       DV01
    _______    ________

    -1.3834    0.048336
```

Input Arguments

`inpPricer` — Pricer object

`Discount` pricer object

Pricer object, specified as a scalar `Discount` pricer object. Use `finpricer` to create the `Discount`pricer object.

Data Types: `object`

`inpInstrument` — Instrument object

`Deposit` object | `FixedBond` object | `FloatBond` object | `FRA` object | `Swap` object | `OISFuture` object | `STIRFuture` object | `OvernightIndexedSwap` object

Instrument object, specified as a scalar or vector for `Deposit`, `FixedBond`, `FloatBond`, `FRA`, `Swap`, `STIRFuture`, `OISFuture`, or `OvernightIndexedSwap` instrument objects. Use `fininstrument` to create the `Deposit`, `FixedBond`, `FloatBond`, `FRA`, `Swap`, `STIRFuture`, `OISFuture`, or `OvernightIndexedSwap` instrument objects.

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute

`[ ]` (default) | string array of character vector with values `"Price"`, `"DV01"`, and `"All"` | cell array of character vectors with values `'Price'`, `'DV01'`, and `'All'`

(Optional) List of sensitivities to compute, specified as a `NOUT`-by-`1` or a `1`-by-`NOUT` cell array of character vectors or string array with possible values of `'Price'` and `'DV01'`.

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that the output is `Price` and `DV01`. This is the same as specifying `inpSensitivity` to include each sensitivity.

The sensitivities supported depend on the `inpInstrument`.

inpInstrument	Supported Sensitivities
`Deposit`	`{'DV01','price'}`
`FixedBond`	`{'DV01','price'}`
`FloatBond`	`{'DV01','price'}`
`FRA`	`{'DV01','price'}`
`Swap`	`{'DV01','price'}`
`STIRFuture`	`{'DV01','price'}`
`OISFuture`	`{'DV01','price'}`
`OvernightIndexedSwap`	`{'DV01','price'}`

Example: `inpSensitivity = {'DV01','price'}`

Data Types: `cell` | `string`

Output Arguments

`Price` — Instrument price
numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result
`PriceResult` object

Price result, returned as an object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes sensitivities (if you specify `inpSensitivity`)
- `PriceResult.PricerData` — Structure for pricer data

Version History

Introduced in R2020a

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for interest-rate instrument with `Future` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

The `price` function computes the value of the futures contract for the party that is long (not short) for the underlying asset, assuming the contract is held until maturity like a forward contract. If the price is negative for the party that is long, then the value is positive for the other party that is short. If the price is positive for the party that is long, then the value is negative for the other party that is short.

Examples

Use `Future` Pricer and `ratecurve` to Price `BondFuture` Instrument

This example shows the workflow to price a `BondFuture` instrument when you use a `ratecurve` object and a `Future` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2022,3,1);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

Create Underlying `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",Maturity=datetime(2032,9,1),CouponRate=0.05,Name="fixed_bond_ins
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
```

```
                Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Sep-2032
                        Name: "fixed_bond_instrument"
```

Create BondFuture Instrument Object

Use `fininstrument` to create a `BondFuture` instrument object.

```
BondFut = fininstrument("BondFuture",Maturity=datetime(2022,9,1),QuotedPrice=86,Bond=FixB,Convers
```

```
BondFut =
  BondFuture with properties:

            Maturity: 01-Sep-2022
         QuotedPrice: 86
                Bond: [1x1 fininstrument.FixedBond]
    ConversionFactor: 1.4300
            Notional: 100000
                Name: "bondfuture_instrument"
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=ZeroCurve,SpotPrice=125)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 125
```

Price BondFuture Instrument

Use `price` to compute the price and price result for the `BondFuture` instrument.

```
[Price,outPR] = price(outPricer,BondFut)
```

```
Price = -151.9270
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×4 table
     Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    _______    _________________    _______________    _______________

    -151.93       1.2283e+05            85.893                0
```

Input Arguments

`inpPricer` — Pricer object

`Future` pricer object

Pricer object, specified as a scalar `Future` pricer object. Use `finpricer` to create the `Future` pricer object.

Data Types: `object`

`inpInstrument` — Instrument object

`BondFuture` object | `CommodityFuture` object | `EquityIndexFuture` object | `FXFuture` object

Instrument object, specified as a scalar or vector for `BondFuture`, `CommodityFuture`, `FXFuture`, or `EquityIndexFuture` instrument objects. Use `fininstrument` to create these instrument objects.

Data Types: `object`

Output Arguments

`Price` — Instrument price

numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result

`PriceResult` object

Price result, returned as an object. The `PriceResult.Results` is a table of results that includes `Price`, `FairDeliveryPrice`, `FairFuturePrice`, and `AccruedInterest`.

Version History

Introduced in R2022a

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for equity instrument with FFT pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities.

Examples

Use FFT Pricer and Heston Model to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `Heston` model and an FFT pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2022,9,15),'Strike',105,'ExerciseSty
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
           Strike: 105
             Name: "vanilla_option"
```

Create `Heston` Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9)
```

```
HestonModel =
  Heston with properties:
```

```
        V0: 0.0320
    ThetaV: 0.1000
     Kappa: 0.0030
    SigmaV: 0.2000
     RhoSV: 0.9000
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
              Dates: 15-Sep-2023
              Rates: 0.0350
             Settle: 15-Sep-2018
       InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create FFT Pricer Object

Use `finpricer` to create an FFT pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("fft",'DiscountCurve',myRC,'Model',HestonModel,'SpotPrice',100,'Characteris
```

```
outPricer =
  FFT with properties:

                  Model: [1x1 finmodel.Heston]
          DiscountCurve: [1x1 ratecurve]
              SpotPrice: 100
           DividendType: "continuous"
          DividendValue: 0
                 NumFFT: 8192
    CharacteristicFcnStep: 0.2000
          LogStrikeStep: 0.0038
      CharacteristicFcn: @characteristicFcnHeston
          DampingFactor: 1.5000
             Quadrature: "simpson"
        VolRiskPremium: 0
             LittleTrap: 1
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])

Price = 14.7545

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []


outPR.Results

ans=1×7 table
    Price      Delta      Gamma       Theta       Rho       Vega     VegaLT
    _____    _______    _________    ________    ______    ______    ______

    14.754    0.44868    0.021649    -0.20891    120.45    88.192    1.3248
```

Input Arguments

`inpPricer` — Pricer object

FFT object

Pricer object, specified as a scalar `FFT` pricer object. Use `finpricer` to create the `FFT` pricer object.

Data Types: `object`

`inpInstrument` — Instrument object

`Vanilla` object

Instrument object, specified as a scalar or vector of `Vanilla` instrument objects. Use `fininstrument` to create `Vanilla` instrument objects.

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute

`[ ]` (default) | string array with values `"Price"`, `"Delta"`, `"Gamma"`, `"Vega"`, `"Rho"`, `"Theta"`, `"Vegalt"`, and `"All"` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Rho'`, `'Theta'`, `'Vegalt'`, and `'All'`

(Optional) List of sensitivities to compute, specified as a `NOUT`-by-`1` or a `1`-by-`NOUT` cell array of character vectors or string array with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Rho'`, `'Theta'`, `'Vegalt'`, and `'All'`.

Note For a `Vanilla` instrument using a `Heston` model, `'Vegalt'` is not supported.

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that the output is `'Delta'`, `'Gamma'`, `'Vega'`, `'Rho'`, `'Theta'`, `'Vegalt'`, and `'Price'`. This is the same as specifying `inpSensitivity` to include each sensitivity.

Example: `inpSensitivity = {'delta','gamma','vega','rho','theta','vegalt','price'}`

Data Types: `string` | `cell`

Output Arguments

`Price` — Instrument price
numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result
`PriceResult` object

Price result, returned as a `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes sensitivities (if you specify `inpSensitivity`)
- `PriceResult.PricerData` — Structure for pricer data

Version History

Introduced in R2020a

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for equity instrument with `FiniteDifference` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities.

Examples

Use Finite Difference Pricer and Black-Scholes Model to Price Barrier Instrument

This example shows the workflow to price a `Barrier` instrument when you use a `BlackScholes` model and a `FiniteDifference` pricing method.

Create `Barrier` Instrument Object

Use `fininstrument` to create a `Barrier` instrument object.

```
BarrierOpt = fininstrument("Barrier",'Strike',105,'ExerciseDate',datetime(2019,1,1),'OptionType'
```

```
BarrierOpt =
  Barrier with properties:

       OptionType: "call"
           Strike: 105
      BarrierType: "do"
     BarrierValue: 40
           Rebate: 0
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2019
             Name: "barrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.30)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2023,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2023
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create FiniteDifference Pricer Object

Use `finpricer` to create a `FiniteDifference` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("FiniteDifference",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPri
```

```
outPricer =
  FiniteDifference with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    GridProperties: [1x1 struct]
      DividendType: "continuous"
     DividendValue: 0
```

Price Barrier Instrument

Use `price` to compute the price and sensitivities for the `Barrier` instrument.

```
[Price, outPR] = price(outPricer,BarrierOpt,["all"])
```

```
Price = 11.3230
```

```
outPR =
  priceresult with properties:
```

```
       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta     Gamma     Lambda     Theta      Rho       Vega
    _____    ________    _____    _______    ______    ______    ______

    11.323    0.54126    0.0132    4.7802    -7.4408    42.766    39.627
```

Input Arguments

`inpPricer` — Pricer object

`FiniteDifference` object

Pricer object, specified as a scalar `FiniteDifference` pricer object. Use `finpricer` to create the `FiniteDifference` pricer object.

Data Types: `object`

`inpInstrument` — Instrument object

`Vanilla` object | `Barrier` object | `DoubleBarrier` object | `ConvertibleBond` object

Instrument object, specified as a scalar or vector of `Vanilla`, `Barrier`, `DoubleBarrier`, or `ConvertibleBond` instrument objects. Use `fininstrument` to create the `Vanilla`, `Barrier`, `DoubleBarrier`, or `ConvertibleBond` instrument objects.

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute

`[ ]` (default) | string array with values `"Price"`, `"Delta"`, `"Gamma"`, `"Vega"`, `"Rho"`, `"Theta"`, `"Lambda"`, `"Vegalt"`, and `"All"` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Lambda'`, `'Vegalt'`, `'Vega'`, `'Rho'`, `'Theta'`, and `'All'`

(Optional) List of sensitivities to compute, specified as a `NOUT`-by-`1` or a `1`-by-`NOUT` cell array of character vectors or string array with supported values.

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that the output is `'Delta'`, `'Gamma'`, `'Vega'`, `'Vegalt'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'Price'`. This is the same as specifying `inpSensitivity` to include each sensitivity.

Note When you price a `Barrier` or `ConvertibleBond` instruments using a `BlackScholes` model, `'Vegalt'` is not supported.

Example: `inpSensitivity = {'delta','gamma','vega','vegalt','rho','lambda','theta','price'}`

The sensitivities supported depend on the `inpInstrument`.

inpInstrument	Supported Sensitivities
`Vanilla`,	`'delta','gamma','vega','vegalt','rho','lambda','theta','price'`
`Barrier`	`'delta','gamma','vega','rho','lambda','theta','price'`
`DoubleBarrier`	`'delta','gamma','vega','vegalt','rho','lambda','theta','price'`
`ConvertibleBond`	`'delta','gamma','vega','rho','lambda','theta','price'`

Data Types: `string` | `cell`

Output Arguments

`Price` — Instrument price
numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result
`PriceResult` object

Price result, returned as a `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes sensitivities (if you specify `inpSensitivity`)
- `PriceResult.PricerData` — Structure for pricer data

Version History

Introduced in R2020a

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for equity instrument with `NumericalIntegration` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities.

Examples

Use Numerical Integration Pricer and Merton Model to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `Merton` model and a `NumericalIntegration` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2020,3,15),'ExerciseStyle',"europea
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Mar-2020
           Strike: 105
             Name: "vanilla_option"
```

Create `Merton` Model Object

Use `finmodel` to create a `Merton` model object.

```
MertonModel = finmodel("Merton",'Volatility',0.45,'MeanJ',0.02,'JumpVol',0.07,'JumpFreq',0.09)
```

```
MertonModel =
  Merton with properties:
```

```
    Volatility: 0.4500
         MeanJ: 0.0200
       JumpVol: 0.0700
      JumpFreq: 0.0900
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
myRC = ratecurve('zero',datetime(2019,9,15),datetime(2020,3,15),0.02)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: 15-Mar-2020
               Rates: 0.0200
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create NumericalIntegration Pricer Object

Use `finpricer` to create a `NumericalIntegration` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("numericalintegration",'Model',MertonModel,'DiscountCurve',myRC,'SpotPrice
```

```
outPricer =
  NumericalIntegration with properties:

               Model: [1x1 finmodel.Merton]
       DiscountCurve: [1x1 ratecurve]
           SpotPrice: 100
        DividendType: "continuous"
       DividendValue: 0.0100
              AbsTol: 0.5000
              RelTol: 0.4000
     IntegrationRange: [1.0000e-09 Inf]
    CharacteristicFcn: @characteristicFcnMerton76
            Framework: "lewis2001"
       VolRiskPremium: 0.9000
           LittleTrap: 0
```

Price Vanilla Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 10.7325
```

```
outPR =
  priceresult with properties:
```

```
       Results: [1x6 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×6 table
    Price     Delta      Gamma       Theta      Rho       Vega
    ______    ______    _________    _______    ______    ______

    10.732    0.5058    0.012492    -12.969    19.815    27.954
```

Input Arguments

`inpPricer` — Pricer object

`NumericalIntegration` object

Pricer object, specified as a scalar `NumericalIntegration` pricer object. Use `finpricer` to create the `NumericalIntegration` pricer object.

Data Types: `object`

`inpInstrument` — Instrument object

`Vanilla` object

Instrument object, specified as a scalar or vector of `Vanilla` instrument objects. Use `fininstrument` to create `Vanilla` instrument objects.

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute

`[ ]` (default) | string array with values `"Price"`, `"Delta"`, `"Gamma"`, `"Vega"`, `"Rho"`, `"Theta"`, `"Vegalt"`, and `"All"` | cell array of character vectors with values `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Rho'`, `'Theta'`, `'Vegalt'`, and `'All'`

(Optional) List of sensitivities to compute, specified as a `NOUT`-by-`1` or a `1`-by-`NOUT` cell array of character vectors or string array with possible values of `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Rho'`, `'Theta'`, `'Vegalt'`, and `'All'`.

Note `'Vegalt'` is not supported when you price a `Vanilla` option with a `Merton` model.

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that the output is `'Delta'`, `'Gamma'`, `'Vega'`, `'Rho'`, `'Theta'`, `'Vegalt'`, and `'Price'`. This is the same as specifying `inpSensitivity` to include each sensitivity.

Example: `inpSensitivity = {'delta','gamma','vega','rho','theta','vegalt','price'}`

Data Types: `string` | `cell`

Output Arguments

`Price` — Instrument price
numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result
`PriceResult` object

Price result, returned as an object. The `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes sensitivities (if you specify `inpSensitivity`)
- `PriceResult.PricerData` — Structure for pricer data

Version History

Introduced in R2020a

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: `finpricer`

Compute price for equity instrument with `ReplicatingVarianceSwap` pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

`[Price,PriceResult] = price(inpPricer,inpInstrument)` computes the instrument price and related pricing information based on the pricing object `inpPricer` and the instrument object `inpInstrument`.

`[Price,PriceResult] = price( ___ ,inpSensitivity)` adds an optional argument to specify sensitivities.

Examples

Use Replicating Variance Swap Pricer and `ratecurve` to Price Variance Swap Instrument

This example shows the workflow to price a `VarianceSwap` instrument when you use a `ratecurve` and a `ReplicatingVarianceSwap` pricing method.

Create `VarianceSwap` Instrument Object

Use `fininstrument` to create a `VarianceSwap` instrument object.

```
VarianceSwapInst = fininstrument("VarianceSwap",'Maturity',datetime(2021,5,1),'Notional',150,'Sta
```

```
VarianceSwapInst =
  VarianceSwap with properties:

            Notional: 150
    RealizedVariance: 0.0500
              Strike: 0.1000
           StartDate: 01-May-2020
            Maturity: 01-May-2021
                Name: "variance_swap_instrument"
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020, 9, 15);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
```

```
Basis = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Basis',Basis)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create ReplicatingVarianceSwap Pricer Object

Use `finpricer` to create a `ReplicatingVarianceSwap` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
Strike = (50:5:135)';
Volatility = [.49;.45;.42;.38;.34;.31;.28;.25;.23;.21;.2;.21;.21;.22;.23;.24;.25;.26];
VolatilitySmile = table(Strike, Volatility);
SpotPrice = 100;
CallPutBoundary = 100;

outPricer =  finpricer("ReplicatingVarianceSwap",'DiscountCurve', ZeroCurve, 'VolatilitySmile', V
'SpotPrice', SpotPrice, 'CallPutBoundary', CallPutBoundary)
```

```
outPricer =
  ReplicatingVarianceSwap with properties:

     DiscountCurve: [1x1 ratecurve]
      InterpMethod: "linear"
    VolatilitySmile: [18x2 table]
         SpotPrice: 100
    CallPutBoundary: 100
```

Price VarianceSwap Instrument

Use `price` to compute the price and fair variance for the `VarianceSwap` instrument.

```
[Price, outPR] = price(outPricer,VarianceSwapInst,["all"])
```

```
Price = 8.1997
```

```
outPR =
  priceresult with properties:

      Results: [1x2 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×2 table
    Price     FairVariance
```

```
    ______    ____________

    8.1997      0.21701
```

```
outPR.PricerData.ReplicatingPortfolio
```

```
ans=19×6 table
    CallPut    Strike    Volatility      Weight       Value     Contribution
    _______    ______    __________    __________    _______    ____________

    "put"        50         0.49        0.0064038    0.39164      0.002508
    "put"        55         0.45        0.0052877    0.49353     0.0026097
    "put"        60         0.42        0.0044402    0.67329     0.0029895
    "put"        65         0.38        0.0037814    0.80343     0.0030381
    "put"        70         0.34        0.0032592     0.9419     0.0030698
    "put"        75         0.31        0.0028382      1.223     0.0034711
    "put"        80         0.28        0.0024938       1.58     0.0039403
    "put"        85         0.25        0.0022086     2.0456     0.0045177
    "put"        90         0.23        0.0019696     2.9221     0.0057554
    "put"        95         0.21        0.0017675     4.1406     0.0073183
    "put"       100          0.2       0.00082405     6.1408     0.0050603
    "call"      100          0.2       0.00077087     6.4715     0.0049887
    "call"      105         0.21        0.0014465     4.7094     0.0068119
    "call"      110         0.21        0.0013178     3.1644     0.0041701
    "call"      115         0.22        0.0012056      2.307     0.0027814
    "call"      120         0.23        0.0011072     1.7127     0.0018962
      ⋮
```

Input Arguments

`inpPricer` — Pricer object

`ReplicatingVarianceSwap` object

Pricer object, specified as a scalar `ReplicatingVarianceSwap` pricer object. Use `finpricer` to create the `ReplicatingVarianceSwap` pricer object.

Data Types: `object`

`inpInstrument` — Instrument object

`VarianceSwap` object

Instrument object, specified as a scalar `VarianceSwap` instrument object. Use `fininstrument` to create the `VarianceSwap` instrument object.

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute

`[ ]` (default) | string array with values `"Price"` and `"All"` | cell array of character vectors with values `'Price'` and `'All'`

(Optional) List of sensitivities to compute, specified as an `NOUT`-by-`1` or `1`-by-`NOUT` cell array of character vectors or string array with possible values of `'Price'` and `'All'`.

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that the output is `'Price'`. This is the same as specifying `inpSensitivity` to include each sensitivity.

Example: `inpSensitivity = {'price'}`

Data Types: `string` | `cell`

Output Arguments

`Price` — Instrument price
numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result
`PriceResult` object

Price result, returned as a `PriceResult` object. The object has the following fields:

- `PriceResult.Results` — Table of results that includes:
  - `Price` — Numeric scalar swap price value
  - `FairVariance` — Numeric fair variance in decimals
- `PriceResult.PricerData.ReplicatingPortfolio` — Table containing pricer data

Version History

Introduced in R2020b

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

pricePortfolio

Compute price and sensitivities for portfolio of instruments

Syntax

```
[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(inPort)
```

Description

`[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(inPort)` calculates the price and sensitivities for a portfolio of instruments that is previously created using `finportfolio`.

Examples

Price Portfolio of Heterogeneous Instruments

Use `finportfolio` and `pricePortfolio` to create and price a portfolio containing a `FixedBond` instrument and an American `Vanilla` option instrument.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",'Maturity',datetime(2022,9,15),'CouponRate',0.05,'Name',"fixed_
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
```

```
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)

myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object for `FixedBond` Instrument

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
FBPricer = finpricer("Discount",'DiscountCurve',myRC)

FBPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Create `Vanilla` Instrument Object

Use `fininstrument` to create an American `Vanilla` instrument object.

```
Maturity = datetime(2023,9,15);
AmericanOpt = fininstrument("Vanilla",'ExerciseDate',Maturity,'Strike',120,'ExerciseStyle',"amer

AmericanOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2023
           Strike: 120
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object for `Vanilla` Instrument

Use `finmodel` to create a `BlackScholes` model object.

```
BSModel = finmodel("BlackScholes",'Volatility',0.12)

BSModel =
  BlackScholes with properties:

     Volatility: 0.1200
```

```
    Correlation: 1
```

Create BjerksundStensland Pricer Object for Vanilla Instrument

Use `finpricer` to create an analytic pricer object for the `BjerksundStensland` pricing method and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
BJSPricer = finpricer("analytic",'Model',BSModel,'DiscountCurve',myRC,'SpotPrice',100,'DividendVa
```

```
BJSPricer =
  BjerksundStensland with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0200
     DividendType: "continuous"
```

Add the Instruments to a finportfolio Object

Create a `finportfolio` object using `finportfolio` and add the two instruments with their associated pricers to the portfolio.

```
f1 = finportfolio([AmericanOpt,FixB],[BJSPricer,FBPricer])
```

```
f1 =
  finportfolio with properties:

    Instruments: [2x1 fininstrument.FinInstrument]
        Pricers: [2x1 finpricer.FinPricer]
    PricerIndex: [2x1 double]
       Quantity: [2x1 double]
```

Price Portfolio

Use `pricePortfolio` to compute the price and sensitivities for the portfolio and the instruments in the portfolio.

```
[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(f1)
```

```
PortPrice = 119.1665

InstPrice = 2×1

    3.1912
  115.9753


PortSens=1×8 table
    Price      DV01       Delta      Gamma       Lambda     Vega      Theta       Rho
    ______    _______    _______    ________    ______    ______    ________    _____

    119.17    0.04295    0.23188    0.011522    7.2661    65.454    -0.81408    86.71


InstSens=2×8 table
                     Price      DV01       Delta      Gamma       Lambda     Vega      Theta
```

```
                        _______    ________    ________    _________    _______    _______    _________

    vanilla_option      3.1912         NaN     0.23188     0.011522     7.2661     65.454     -0.81408
    fixed_bond          115.98     0.04295         NaN          NaN        NaN        NaN          NaN
```

Price Portfolio of Bond and Bond Option Instruments

This example shows the workflow to create and price a portfolio of bond and bond option instruments. You can use `finportfolio` and `pricePortfolio` to price `FixedBond`, `FixedBondOption`, `OptionEmbeddedFixedBond`, and `FloatBond` instruments using an `IRTree` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018, 1, 1);
 ZeroTimes = calyears(1:4)';
 ZeroRates = [0.035; 0.042147; 0.047345; 0.052707];
 ZeroDates = Settle + ZeroTimes;
 Compounding = 1;
 ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: 1
               Basis: 0
               Dates: [4x1 datetime]
               Rates: [4x1 double]
              Settle: 01-Jan-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create Bond and Option Instruments

Use `fininstrument` to create a `FixedBond`, `FixedBondOption`, `OptionEmbeddedFixedBond`, and `FloatBond` instrument objects.

```
CDates = datetime([2020,1,1 ; 2022,1,1]);
CRates = [.0425; .0750];
CouponRate = timetable(CDates,CRates);
Maturity = datetime(2022,1,1);
Period = 1;

% Vanilla FixedBond
VBond = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',0.0425,'Period',Period,'Name'
```

```
VBond =
  FixedBond with properties:

                 CouponRate: 0.0425
```

```
                    Period: 1
                     Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                        Name: "vanilla_fixed"
```

```
 % Stepped coupon bond
SBond = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',CouponRate,'Period',Period,'Na

SBond =
  FixedBond with properties:

                CouponRate: [2x1 timetable]
                    Period: 1
                     Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                        Name: "stepped_coupon_bond"
```

```
% FloatBond
Spread = 0;
Reset = 1;
Float = fininstrument("FloatBond",'Maturity',Maturity,'Spread',Spread,'Reset', Reset,...
                      'ProjectionCurve',ZeroCurve,'Name',"floatbond")

Float =
  FloatBond with properties:

                    Spread: 0
            ProjectionCurve: [1x1 ratecurve]
                ResetOffset: 0
                      Reset: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
          LatestFloatingRate: NaN
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
```

```
               LastCouponDate: NaT
                    StartDate: NaT
                     Maturity: 01-Jan-2022
                         Name: "floatbond"
```

```
% Call option
Strike = 100;
ExerciseDates = datetime(2020,1,1);
OptionType ='call';
Period = 1;
CallOption = fininstrument("FixedBondOption",'Strike',Strike,'ExerciseDate',ExerciseDates,...
                  'OptionType',OptionType,'ExerciseStyle',"american",'Bond', VBond,'Name',"fixe
```

```
CallOption =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2020
           Strike: 100
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option"
```

```
% Option for embedded bond (callable bond)
CDates = datetime([2020,1,1 ; 2022,1,1]);
CRates = [.0425; .0750];
CouponRate = timetable(CDates,CRates);
StrikeOE = [100; 100];
ExerciseDatesOE = [datetime(2020,1,1); datetime(2021,1,1)];
CallSchedule =  timetable(ExerciseDatesOE,StrikeOE,'VariableNames',{'Strike Schedule'});
CallableBond = fininstrument("OptionEmbeddedFixedBond", 'Maturity',Maturity,...
                             'CouponRate',CouponRate,'Period', Period, ...
                             'CallSchedule',CallSchedule,'Name',"option_embedded_fixedbond")
```

```
CallableBond =
  OptionEmbeddedFixedBond with properties:

                 CouponRate: [2x1 timetable]
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                   CallDates: [2x1 datetime]
                    PutDates: [0x1 datetime]
                CallSchedule: [2x1 timetable]
                 PutSchedule: [0x0 timetable]
           CallExerciseStyle: "american"
            PutExerciseStyle: [0x0 string]
```

```
    Name: "option_embedded_fixedbond"
```

Create HullWhite Model

Use `finmodel` to create a `HullWhite` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;

HWModel = finmodel("hullwhite",'alpha',AlphaCurve,'sigma',VolCurve)
```

```
HWModel =
  HullWhite with properties:

    Alpha: 0.1000
    Sigma: 0.0100
```

Create IRTree Pricer for HullWhite Model

Use `finpricer` to create an `IRTree` pricer object for a `HullWhite` model and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [4x1 datetime]
           Model: [1x1 finmodel.HullWhite]
   DiscountCurve: [1x1 ratecurve]
```

Create finportfolio Object and Add Callable Bond Instrument

Create a `finportfolio` object with the vanilla bond, stepped coupon bond, float bond, and the call option.

```
myportfolio = finportfolio([VBond,SBond,Float,CallOption],HWTreePricer, [1,2,2,1])
```

```
myportfolio =
  finportfolio with properties:

    Instruments: [4x1 fininstrument.FinInstrument]
        Pricers: [1x1 finpricer.irtree.HWBKTree]
    PricerIndex: [4x1 double]
       Quantity: [4x1 double]
```

Use `addInstrument` to add the callable bond instrument to the existing portfolio.

```
myportfolio = addInstrument(myportfolio,CallableBond,HWTreePricer,1)
```

```
myportfolio =
  finportfolio with properties:

    Instruments: [5x1 fininstrument.FinInstrument]
```

```
        Pricers: [1x1 finpricer.irtree.HWBKTree]
    PricerIndex: [5x1 double]
       Quantity: [5x1 double]
```

```
myportfolio.PricerIndex
```

```
ans = 5×1

     1
     1
     1
     1
     1
```

The `PricerIndex` property has a length equal to the length of instrument objects in the `finportfolio` object and stores the index of which pricer is used for each instrument object. In this case, because there is only one pricer, each instrument must use that pricer.

Price Portfolio

Use `pricePortfolio` to compute the price and sensitivities for the portfolio and the bond and option instruments in the portfolio.

```
format bank
[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(myportfolio)
```

```
PortPrice =
        600.55


InstPrice = 5×1

         96.59
        204.14
        200.00
          0.05
         99.77


PortSens=1×4 table
    Price      Delta       Gamma      Vega
    ______    ________    _______    ______

    600.55    -1297.48    5759.65    -63.40


InstSens=5×4 table
                              Price      Delta      Gamma      Vega
                              ______    _______    _______    ______

    vanilla_fixed              96.59    -344.81    1603.49     -0.00
    stepped_coupon_bond       204.14    -725.96    3364.60      0.00
    floatbond                 200.00       0.00      -0.00     -0.00
    fixed_bond_option           0.05      -3.69      24.15     12.48
```

```
    option_embedded_fixedbond     99.77    -223.03     767.41    -75.88
```

Input Arguments

`inPort` — Portfolio object

`finportfolio` object

Portfolio object, previously created using `finportfolio`.

Data Types: `object`

Output Arguments

`PortPrice` — Price of portfolio of instruments

numeric

Price of the portfolio of instruments, returned as a numeric.

`InstPrice` — Instrument prices

numeric

Instrument prices, returned as a numeric.

`PortSens` — Portfolio sensitivities

numeric

Portfolio sensitivities, returned as a numeric.

`InstSens` — Instrument sensitivities

numeric

Instrument sensitivities, returned as a numeric.

Version History

Introduced in R2020a

See Also

`fininstrument` | `addInstrument` | `setPricer` | `removeInstrument`

Topics

“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

addInstrument

Add instrument to portfolio of instruments

Syntax

```
outPort = addInstrument(inPort,inInst)
outPort = addInstrument(inPort,inInst,inPricer)
outPort = addInstrument( ___ ,inQuant)
```

Description

`outPort = addInstrument(inPort,inInst)` adds the instrument `inInst` to a portfolio `inPort` of instruments previously created using `finportfolio`.

`outPort = addInstrument(inPort,inInst,inPricer)` adds the instrument `inInst` and the associated pricer `inPricer` to a portfolio `inPort` of instruments previously created using `finportfolio`.

`outPort = addInstrument( ___ ,inQuant)` optionally specifies the number (`inQuant`) of added instruments. Use this syntax with any of the input argument combinations in previous syntaxes.

Examples

Add Instruments to Portfolio

Use `finportfolio` to create an empty portfolio and then use `addInstrument` to add instruments to the portfolio.

Create `FixedBond` Instrument Objects

Use `fininstrument` to create two `FixedBond` instrument objects.

```
FixB1 = fininstrument("FixedBond", 'Maturity',datetime(2022,9,15),'CouponRate',0.045,'Name',"fixe
```

```
FixB1 =
  FixedBond with properties:

                 CouponRate: 0.0450
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
```

```
                        Name: "fixed_bond1"
```

```
FixB2 = fininstrument("FixedBond", 'Maturity',datetime(2022,9,15),'CouponRate',0.035,'Name',"fixe
```

```
FixB2 =
  FixedBond with properties:

                  CouponRate: 0.0350
                      Period: 2
                       Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond2"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object for `FixedBond` Instruments

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
DiscountPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
DiscountPricer =
  Discount with properties:
```

```
    DiscountCurve: [1x1 ratecurve]
```

Add Instruments to `finportfolio` Object

Create an empty `finportfolio` object using `finportfolio` and then use `addInstrument` to put the `FixedBond` instruments into the portfolio.

```
f1 = finportfolio;
f1 = addInstrument(f1,FixB1)
```

```
f1 =
  finportfolio with properties:

    Instruments: [1x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
    PricerIndex: NaN
       Quantity: 1


f1 = addInstrument(f1,FixB2)
```

```
f1 =
  finportfolio with properties:

    Instruments: [2x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
    PricerIndex: [2x1 double]
       Quantity: [2x1 double]
```

Set Pricer for Portfolio

Use `setPricer` to set the pricer for the portfolio and then use `pricePortfolio` to calculate the price and sensitivities for the instruments in the portfolio.

```
f1 = setPricer(f1,DiscountPricer,[1,2])
```

```
f1 =
  finportfolio with properties:

    Instruments: [2x1 fininstrument.FixedBond]
        Pricers: [1x1 finpricer.Discount]
    PricerIndex: [2x1 double]
       Quantity: [2x1 double]


[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(f1)
```

```
PortPrice = 224.0834

InstPrice = 2×1

  114.0085
  110.0749


PortSens=1×2 table
    Price       DV01
```

```
    ______    ________

    224.08    0.084139


InstSens=2×2 table
                    Price       DV01
                    ______    ________

    fixed_bond1     114.01     0.04251
    fixed_bond2     110.07    0.041629
```

Add Multiple Instruments to Portfolio

Use `finportfolio` to create an empty portfolio and then use `addInstrument` to add multiple instruments to the portfolio.

Create FixedBond Instrument Objects

Use `fininstrument` to create two `FixedBond` instrument objects each with two instruments.

```
FixB1 = fininstrument("FixedBond", 'Maturity',datetime([2022,9,15 ; 2022,10,15]),'CouponRate',0.0
```

```
FixB1=2×1 object
  2x1 FixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    Name
```

```
FixB2 = fininstrument("FixedBond", 'Maturity',datetime([2022,11,15 ; 2022,12,15]),'CouponRate',0
```

```
FixB2=2×1 object
  2x1 FixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
```

```
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    Name
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object for `FixedBond` Instruments

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
DiscountPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
DiscountPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Add Instruments to `finportfolio` Object

Create an empty `finportfolio` object using `finportfolio` and then use `addInstrument` to put the `FixedBond` instruments into the portfolio.

```
f1 = finportfolio;
f1 = addInstrument(f1,FixB1(1))
```

```
f1 =
  finportfolio with properties:

    Instruments: [1x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
```

```
    PricerIndex: NaN
       Quantity: 1


f1 = addInstrument(f1,FixB1(2))

f1 =
  finportfolio with properties:

    Instruments: [2x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
    PricerIndex: [2x1 double]
       Quantity: [2x1 double]


f1 = addInstrument(f1,FixB2(1))

f1 =
  finportfolio with properties:

    Instruments: [3x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
    PricerIndex: [3x1 double]
       Quantity: [3x1 double]


f1 = addInstrument(f1,FixB2(2))

f1 =
  finportfolio with properties:

    Instruments: [4x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
    PricerIndex: [4x1 double]
       Quantity: [4x1 double]
```

Set Pricer for Portfolio

Use `setPricer` to set the pricer for the portfolio and then use `pricePortfolio` to calculate the prices and sensitivities for the instruments in the portfolio.

```
f1 = setPricer(f1,DiscountPricer,[1,2,3,4])

f1 =
  finportfolio with properties:

    Instruments: [4x1 fininstrument.FixedBond]
        Pricers: [1x1 finpricer.Discount]
    PricerIndex: [4x1 double]
       Quantity: [4x1 double]


[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(f1)

PortPrice = 428.2788

InstPrice = 4×1

  107.7226
```

```
  108.0156
  106.1642
  106.3765


PortSens=1×2 table
    Price       DV01
    ______    ________

    428.28    0.088272


InstSens=4×2 table
                      Price       DV01
                      ______    ________

    fixed_bond1       107.72    0.020871
    fixed_bond1_1     108.02    0.021761
    fixed_bond2       106.16    0.022387
    fixed_bond2_1     106.38    0.023253
```

Input Arguments

`inPort` — `finportfolio` object

`finportfolio` object

`finportfolio` object, specified as a scalar `finportfolio` object.

Data Types: `object`

`inInst` — Instrument object to add to portfolio

scalar instrument object

Instrument object to add to portfolio, specified as a scalar instrument object that is previously created using `fininstrument`.

Note If the instrument object for `inInst` is a vector of instruments, you must use `addInstrument` to add each instrument separately.

Data Types: `object`

`inPricer` — Pricer object associated with an added instrument object

scalar pricer object | array of pricer objects

Pricer object associated with an added instrument object, specified as a scalar pricer object or an array of pricer objects that are previously created using `finpricer`.

Data Types: `object`

`inQuant` — Number of added instruments

1 (default) | positive or negative numeric

Number of instruments, specified as a scalar numeric. Use a positive value for a long position and a negative value for a short position.

Data Types: `double`

Output Arguments

`outPort` — Updated `finportfolio` object
`finportfolio` object

Updated `finportfolio`, returned as a `finportfolio` object.

Version History

Introduced in R2020a

See Also

`fininstrument` | `removeInstrument` | `pricePortfolio` | `setPricer`

Topics

“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

removeInstrument

Remove instrument from portfolio of instruments

Syntax

```
outPort = removeInstrument(inPort,inInst)
```

Description

`outPort = removeInstrument(inPort,inInst)` removes an instrument object (`inInst`) from a portfolio of instruments (`inPort`) previously created using `finportfolio`.

Examples

Remove Instrument From a Portfolio

Use `addInstrument` to add instruments to an empty portfolio and then remove an instrument from the portfolio using `removeInstrument`.

Create FixedBond Instrument Objects

Use `fininstrument` to create two `FixedBond` instrument objects.

```
FixB1 = fininstrument("FixedBond", 'Maturity',datetime(2022,9,15),'CouponRate',0.045,'Name',"fixe
```

```
FixB1 =
  FixedBond with properties:

                 CouponRate: 0.0450
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond1"
```

```
FixB2 = fininstrument("FixedBond", 'Maturity',datetime(2022,9,15),'CouponRate',0.035,'Name',"fixe
```

```
FixB2 =
  FixedBond with properties:

                 CouponRate: 0.0350
                     Period: 2
```

```
                    Basis: 0
             EndMonthRule: 1
                Principal: 100
  DaycountAdjustedCashFlow: 0
     BusinessDayConvention: "actual"
                 Holidays: NaT
                IssueDate: NaT
          FirstCouponDate: NaT
           LastCouponDate: NaT
                StartDate: NaT
                 Maturity: 15-Sep-2022
                     Name: "fixed_bond2"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object for `FixedBond` Instruments

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
DiscountPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
DiscountPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Add Instruments to `finportfolio` Object

Create an empty `finportflio` object using `finportfolio` and then use `addInstrument` to add the two `FixedBond` instruments to the portfolio.

```
f1 = finportfolio;
f1 = addInstrument(f1,FixB1)

f1 =
  finportfolio with properties:

    Instruments: [1x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
    PricerIndex: NaN
       Quantity: 1


f1 = addInstrument(f1,FixB2)

f1 =
  finportfolio with properties:

    Instruments: [2x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
    PricerIndex: [2x1 double]
       Quantity: [2x1 double]
```

Remove Instrument from `finportfolio` Object

Use `removeInstrument` to remove the first `FixedBond` instrument from the portfolio.

```
f1 = removeInstrument(f1,1)

f1 =
  finportfolio with properties:

    Instruments: [1x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
    PricerIndex: NaN
       Quantity: 1
```

Set Pricer for Portfolio

Use `setPricer` to set the `Discount` pricer for the portfolio and then use `pricePortfolio` to calculate the price and sensitivities for the single instrument in the portfolio.

```
f1 = setPricer(f1,DiscountPricer)

f1 =
  finportfolio with properties:

    Instruments: [1x1 fininstrument.FixedBond]
        Pricers: [1x1 finpricer.Discount]
    PricerIndex: 1
       Quantity: 1


[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(f1)

PortPrice = 110.0749

InstPrice = 110.0749
```

```
PortSens=1×2 table
    Price       DV01
    ______    ________

    110.07    0.041629


InstSens=1×2 table
                   Price       DV01
                   ______    ________

    fixed_bond2    110.07    0.041629
```

Input Arguments

`inPort` — `finportfolio` object

`finportfolio` object

`finportfolio` object, specified as a scalar `finportfolio` object.

Data Types: `object`

`inInst` — Instrument to remove from `finportfolio` object

instrument object | index | string for the instrument object `'Name'` property

Instrument to remove from `finportfolio` object, specified as a scalar instrument object, string for the instrument object `'Name'` property, or index value for the position of instrument in the `finportfolio` object.

Data Types: `object` | `double` | `string`

Output Arguments

`outPort` — Updated `finportfolio` object

`finportfolio` object

Updated `finportfolio`, returned as a `finportfolio` object.

Version History

Introduced in R2020a

See Also

`fininstrument` | `setPricer` | `pricePortfolio` | `addInstrument`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

setPricer

Set pricer for `finportfolio` object

Syntax

```
outPort = setPricer(inPort,inPricer,Index)
```

Description

`outPort = setPricer(inPort,inPricer,Index)` sets a specified pricer for the previously created `finportfolio` object.

Examples

Set Pricer for Portfolio Containing Instruments

Use `finportfolio` to create a portfolio of instruments and then use `setPricer` to set the pricer for the portfolio.

Create `FixedBond` Instrument Objects

Use `fininstrument` to create two `FixedBond` instrument objects.

```
FixB1 = fininstrument("FixedBond",'Maturity',datetime(2022,9,15),'CouponRate',0.045,'Name',"fixe
```

```
FixB1 =
  FixedBond with properties:

                 CouponRate: 0.0450
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond1"
```

```
FixB2 = fininstrument("FixedBond",'Maturity',datetime(2022,9,15),'CouponRate',0.035,'Name',"fixe
```

```
FixB2 =
  FixedBond with properties:

                 CouponRate: 0.0350
                     Period: 2
```

```
                    Basis: 0
             EndMonthRule: 1
                Principal: 100
DaycountAdjustedCashFlow: 0
   BusinessDayConvention: "actual"
                 Holidays: NaT
                IssueDate: NaT
          FirstCouponDate: NaT
           LastCouponDate: NaT
                StartDate: NaT
                 Maturity: 15-Sep-2022
                     Name: "fixed_bond2"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object for `FixedBond` Instruments

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
DiscountPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
DiscountPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Add Instruments to `finportfolio` Object

Create a `finportflio` object using `finportfolio` and use `addInstrument` to put the `FixedBond` instruments in the portfolio.

```
f1 = finportfolio;
f1 = addInstrument(f1,FixB1)

f1 =
  finportfolio with properties:

    Instruments: [1x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
    PricerIndex: NaN
       Quantity: 1


f1 = addInstrument(f1,FixB2)

f1 =
  finportfolio with properties:

    Instruments: [2x1 fininstrument.FixedBond]
        Pricers: [0x1 finpricer.FinPricer]
    PricerIndex: [2x1 double]
       Quantity: [2x1 double]
```

Set Pricer for Portfolio

Use `setPricer` to set the pricer for the portfolio and then use `pricePortfolio` to calculate the price and sensitivities for the instruments in the portfolio.

```
f1 = setPricer(f1,DiscountPricer,[1,2])

f1 =
  finportfolio with properties:

    Instruments: [2x1 fininstrument.FixedBond]
        Pricers: [1x1 finpricer.Discount]
    PricerIndex: [2x1 double]
       Quantity: [2x1 double]


[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(f1)

PortPrice = 224.0834

InstPrice = 2×1

  114.0085
  110.0749


PortSens=1×2 table
    Price       DV01
    ______    ________

    224.08    0.084139


InstSens=2×2 table
                   Price       DV01
                   ______    ________
```

```
fixed_bond1    114.01     0.04251
fixed_bond2    110.07    0.041629
```

Input Arguments

`inPort` — Portfolio

`finportfolio` object

Portfolio, specified using a previously created `finportfolio` object.

Data Types: `object`

`inPricer` — Pricer object to set for an instrument in a `finportfolio` object

object

Pricer object to set for an instrument in a `finportfolio` object, specified using a previously created pricer object with `finpricer`.

Data Types: `object`

`Index` — Index to instruments in `finportfolio` object

numeric

Index to instruments in the `finportfolio` object, specified as a numeric value.

Data Types: `double`

Output Arguments

`outPort` — Updated portfolio

`finportfolio` object

Updated portfolio, returned as an `finportfolio` object.

Version History

Introduced in R2020a

See Also

`finportfolio` | `pricePortfolio` | `addInstrument` | `removeInstrument`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

cashflows

Package: `fininstrument`

Compute cash flow for `FixedBond`, `FloatBond`, `Swap`, `FRA`, `STIRFuture`, `OISFuture`, `OvernightIndexedSwap`, or `Deposit` instrument

Syntax

```
CF = cashflows(InstrumentObject,Settle)
```

Description

`CF = cashflows(InstrumentObject,Settle)` computes cash flow for a `Deposit`, `FRA`, ,`Swap`, `STIRFuture`, `OISFuture`, `FixedBond`, `OvernightIndexedSwap`, or `FloatBond` instrument object.

Examples

Calculate Cash Flow for FRA Instrument

This example shows the workflow to price a `FRA` (forward rate agreement) instrument and then use `cashflows` to determine the cash flow for the `FRA` instrument.

Create FRA Instrument Object

Use `fininstrument` to create a `FRA` instrument object.

```
FRAObj = fininstrument("FRA",'StartDate',datetime(2020,9,15),'Maturity',datetime(2022,9,15),'Rate
```

```
FRAObj =
  FRA with properties:

                     Rate: 0.1750
                    Basis: 2
                StartDate: 15-Sep-2020
                 Maturity: 15-Sep-2022
                Principal: 100
    BusinessDayConvention: "actual"
                 Holidays: NaT
                     Name: ""
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price FRA Instrument

Use `price` to compute the price and sensitivities for the FRA instrument.

```
[Price, outPR] = price(outPricer, FRAObj,["all"])
```

```
Price = 34.1757
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price      DV01
    ______    _______

    34.176    0.01368
```

Use `cashflows` for the FRA instrument with a `Settle` date of `15-Dec-2021`. The specified `Settle` date must be before the instrument `Maturity` date.

```
CF = cashflows(FRAObj,datetime(2021,12,15))
```

```
CF= 1×1timetable
       Time         CFA
    ___________    ______
```

```
    15-Sep-2022    35.486
```

Calculate Cash Flow for OIS Future Instrument

This example shows the workflow to price an `OISFuture` instrument and then use `cashflows` to calculate the cashflow for the `OISFuture` instrument.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `STIRFuture` instrument.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `OISFuture` Instrument Object

Use `fininstrument` to create an `OISFuture` instrument object.

```
OISFuture = fininstrument("OISFuture",'Maturity',datetime(2022,12,15),'QuotedPrice',99.5,'StartDa
```

```
OISFuture =
  OISFuture with properties:

             QuotedPrice: 99.5000
                  Method: "compound"
                   Basis: 2
               StartDate: 15-Sep-2022
                Maturity: 15-Dec-2022
                Notional: 90
    BusinessDayConvention: "actual"
                Holidays: NaT
          ProjectionCurve: [1x1 ratecurve]
        HistoricalFixing: [0x0 timetable]
                    Name: "ois_future_instrument"
```

Create Discount Pricer Object

Use finpricer to create a Discount pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price OISFuture Instrument

Use price to compute the price and sensitivities for the OISFuture instrument.

```
[Price, outPR] = price(outPricer,OISFuture,["all"])
```

```
Price = 2.6543

outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price        DV01
    ______    __________

    2.6543    -0.0013589
```

Use cashflows to calculate the cash flow for the OISFuture instrument with a Settle date of 15-Sep-2022. The specified Settle date must be before the instrument Maturity date.

```
CF = cashflows(OISFuture,datetime(2022,9,15))
```

```
CF= 1×1timetable
       Time          CFA
    ___________    ______

    15-Dec-2022    2.7225
```

Calculate Cash Flow for Multiple FRA Instruments

This example shows the workflow to price multiple FRA (forward rate agreement) instruments and then use cashflows to determine the cash flow for each of the FRA instruments.

Create FRA Instrument Object

Use fininstrument to create a FRA instrument object for three FRA instruments.

```
FRAObj = fininstrument("FRA",'StartDate',datetime([2020,9,15 ; 2020,10,15 ; 2020,11,15]),'Maturi
```

```
FRAObj=3×1 object
  3x1 FRA array with properties:

    Rate
    Basis
    StartDate
    Maturity
    Principal
    BusinessDayConvention
    Holidays
    Name
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price FRA Instruments

Use `price` to compute the prices and sensitivities for the three FRA instrument.

```
[Price, outPR] = price(outPricer, FRAObj,["all"])
```

```
Price = 3×1

   34.1757
   34.1207
   34.0627


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData


outPR.Results

ans=1×2 table
    Price      DV01
    ______    _______

    34.176    0.01368


ans=1×2 table
    Price       DV01
    ______    ________

    34.121    0.013938


ans=1×2 table
    Price       DV01
    ______    ________

    34.063    0.014204
```

Use `cashflows` for the three `FRA` instruments with a `Settle` date of April 15, 2022. The specified `Settle` date must be before the instrument `Maturity` date.

```
CF = cashflows(FRAObj(1),datetime(2022,4,15))

CF= 1×1timetable
       Time          CFA
    ___________    ______

    15-Sep-2022    35.486


CF = cashflows(FRAObj(2),datetime(2022,4,15))

CF= 1×1timetable
       Time          CFA
    ___________    ______

    15-Oct-2022    35.486


CF = cashflows(FRAObj(3),datetime(2022,4,15))
```

```
CF= 1×1timetable
       Time          CFA
    ____________    ______

    15-Nov-2022    35.486
```

Calculate Cash Flow for FixedBond Instrument

This example shows the workflow to price a `FixedBond` instrument and then use `cashflows` to calculate the cash flow for the `FixedBond` instrument.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",'Maturity',datetime(2022,9,15),'CouponRate',0.05,'Period',4,'Bas
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 4
                      Basis: 7
                EndMonthRule: 1
                   Principal: 1000
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "follow"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond_instrument"
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 0
              Dates: [10x1 datetime]
              Rates: [10x1 double]
```

```
              Settle: 15-Sep-2018
          InterpMethod: "linear"
     ShortExtrapMethod: "next"
      LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `FixedBond` Instrument

Use `price` to compute the price and sensitivities for the `FixedBond` instrument.

```
[Price, outPR] = price(outPricer, FixB,["all"])
```

```
Price = 1.1600e+03
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price     DV01
    _____    _______

    1160     0.42712
```

Use `cashflows` to calculate the cash flow for the `FixedBond` instrument for any specified `Settle` date before the instrument `Maturity` date.

```
CF = cashflows(FixB,datetime(2021,9,15))
```

```
CF=5×1 timetable
       Time         Var1
    ____________    ______

    15-Sep-2021        0
    15-Dec-2021      12.5
    15-Mar-2022      12.5
    15-Jun-2022      12.5
```

```
15-Sep-2022    1012.5
```

Input Arguments

`InstrumentObject` — Instrument object

`Deposit` object | `FixedBond` object | `FloatBond` object | `Swap` object | `STIRFuture` object | `OISFuture` object | `OvernightIndexedSwap` object | `FRA` object

Instrument object, specified using a previously created instrument object for one of the following: `Deposit`, `FixedBond`, `FloatBond`, `Swap`, `STIRFuture`, `OISFuture`, `OvernightIndexedSwap`, or `FRA`.

Note If the `InstrumentObject` is a vector of instruments, you must use `cashflows` separately with each instrument.

Data Types: `object`

`Settle` — Settlement date for instrument cash flow

datetime scalar | string scalar | date character vector

Settlement date for instrument cash flow, specified as a scalar datetime, string, or date character vector.

Note The `Settle` date you specify must be before the `Maturity` date for the `Deposit`, `FixedBond`, `FloatBond`, `Swap`, `STIRFuture`, `OISFuture`,`OvernightIndexedSwap`, or `FRA` instrument.

To support existing code, `cashflows` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`CF` — Cash flow

timetable

Cash flow, returned as a timetable.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `cashflows` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`fininstrument` | `finmodel` | `finpricer` | `timetable`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

cashsettle

Package: `fininstrument`

Compute cash settlement for `BondFuture`, `CommodityFuture`, `EquityIndexFuture`, or `FXFuture` instrument

Syntax

```
outCS = cashsettle(InstrumentObject,SpotPrice,DiscountCurve)
```

Description

`outCS = cashsettle(InstrumentObject,SpotPrice,DiscountCurve)` computes the cash settlement for a `BondFuture`, `CommodityFuture`, `FXFuture`, or `EquityIndexFuture` instrument object.

Examples

Compute Cash Settlement for `BondFuture` Instrument

This example shows the workflow to price a `BondFuture` instrument and then use `cashsettle` to compute the cash settlement amount for the `BondFuture` instrument.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2022,3,1);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

Create Underlying `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",Maturity=datetime(2032,9,1),CouponRate=0.05,Name="fixed_bond_ins
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
```

```
              FirstCouponDate: NaT
               LastCouponDate: NaT
                    StartDate: NaT
                     Maturity: 01-Sep-2032
                         Name: "fixed_bond_instrument"
```

Create BondFuture Instrument Object

Use `fininstrument` to create a `BondFuture` instrument object.

```
BondFut = fininstrument("BondFuture",Maturity=datetime(2022,9,1),QuotedPrice=86,Bond=FixB,Convers
```

```
BondFut =
  BondFuture with properties:

           Maturity: 01-Sep-2022
        QuotedPrice: 86
               Bond: [1x1 fininstrument.FixedBond]
    ConversionFactor: 1.4300
           Notional: 100000
               Name: "bondfuture_instrument"
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=ZeroCurve,SpotPrice=125)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 125
```

Price BondFuture Instrument

Use `price` to compute the price and price result for the `BondFuture` instrument.

```
[Price,outPR] = price(outPricer,BondFut)
```

```
Price = -151.9270

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×4 table
     Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    _______    _________________    _______________    _______________
```

```
    -151.93       1.2283e+05            85.893                0
```

Compute Cash Settlement Amount

Use `cashsettle` with the `BondFuture` instrument to compute the cash settlement.

```
SpotPrice = 125; % Clean spot price for $100 face value of underlying bond.
outCS = cashsettle(BondFut,SpotPrice,ZeroCurve)
```

```
outCS= 1×1timetable
       Time        CashSettleAmount
    ___________    ________________

    01-Sep-2022        -152.33
```

Compute Cash Settlement for Multiple FXFuture Instruments

This example shows the workflow to price multiple `FXFuture` instruments and then use `cashsettle` to compute the cash settlement amount for the `FXFuture` instruments.

Create `ratecurve` Objects

Create `ratecurve` objects using `ratecurve` for the foreign and domestic zero curves.

```
% Define Foreign Zero Curve
Settle = datetime(2022, 3, 1);
ForeignZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ForeignZeroRates = [0.0031 0.0035 0.0047 0.0058 0.0062 0.0093 0.0128 0.0182 0.0223 0.0285]';
ForeignZeroDates = Settle + ForeignZeroTimes;
ForeignRC = ratecurve('zero', Settle, ForeignZeroDates, ForeignZeroRates);

% Define Domestic Zero Curve
DomesticZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
DomesticZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
DomesticZeroDates = Settle + DomesticZeroTimes;
DomesticRC = ratecurve('zero', Settle, DomesticZeroDates, DomesticZeroRates);
```

Create FXFuture Instrument Object

Use `fininstrument` to create a `FXFuture` instrument object for three FX Future instruments.

```
FXFut = fininstrument("FXFuture",Maturity=datetime([2022,9,1 ; 2022,10,1 ; 2022,11,1]),QuotedPric
```

```
FXFut=3×1 object
  3x1 FXFuture array with properties:

    Maturity
    QuotedPrice
    ForeignRateCurve
    Notional
    Name
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=DomesticRC,SpotPrice=0.79)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 0.7900
```

Price FXFuture Instruments

Use `price` to compute the prices and price results for the `FXFuture` instrument.

```
[Price,outPR] = price(outPricer,FXFut)
```

```
Price = 3×1
10^4 ×

    0.2162
   -0.5789
   -1.3732


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    2161.7       1.5817e+05            0.79084              0


ans=1×4 table
    Price    FairDeliveryPrice    FairFuturePrice    AccruedInterest
    _____    _________________    _______________    _______________

    -5789       1.5819e+05            0.79097              0


ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    -13732       1.5822e+05            0.7911               0
```

Compute Cash Settlement Amounts

Use `cashsettle` with the `FXFuture` instruments to compute the cash settlement.

```
SpotPrice = 0.79; % Quoted in domestic currency for one unit of foreign currency
outCS = cashsettle(FXFut(1),SpotPrice,DomesticRC)
```

```
outCS= 1×1timetable
       Time        CashSettleAmount
    ___________    ________________

    01-Sep-2022         2167.4
```

```
outCS = cashsettle(FXFut(2),SpotPrice,DomesticRC)
```

```
outCS= 1×1timetable
       Time        CashSettleAmount
    ___________    ________________

    01-Oct-2022        -5806.9
```

```
outCS = cashsettle(FXFut(3),SpotPrice,DomesticRC)
```

```
outCS= 1×1timetable
       Time        CashSettleAmount
    ___________    ________________

    01-Nov-2022         -13781
```

Input Arguments

`InstrumentObject` — Instrument object
`BondFuture` object | `CommodityFuture` object | `EquityIndexFuture` object | `FXFuture` object

Instrument object, specified using a previously created instrument object for one of the following: `BondFuture`, `CommodityFuture`, `FXFuture`, or `EquityIndexFuture`.

Note If `InstrumentObject` is a vector of instruments, you must use `cashsettle` separately with each instrument.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified using the name of a previously created `ratecurve` object.

Data Types: `object`

`SpotPrice` — Quoted spot price for underlying asset to be delivered
numeric

Quoted spot price for underlying asset to be delivered, specified using a numeric value that depends on the type of future instrument being priced:

- `BondFuture` instrument — Clean spot price quoted for $100 face value of underlying bond
- `CommodityFuture` instrument — Spot price for underlying commodity quantity specified in contract
- `EquityIndexFuture` instrument — Spot equity index value
- `FXFuture` instrument — Spot price quoted in domestic currency for one unit of foreign currency

Data Types: `double`

Output Arguments

`outCS` — Cash settlement
timetable

Cash settlement, returned as a timetable.

Version History

Introduced in R2022a

See Also

`fininstrument` | `finmodel` | `finpricer` | `fairdelivery`

Topics

“Select Cheapest-to-Deliver Bond Using BondFuture Instrument” on page 2-212
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

fairdelivery

Package: `fininstrument`

Compute fair delivery price of underlying asset for `BondFuture`, `CommodityFuture`, `EquityIndexFuture`, or `FXFuture` instrument

Syntax

```
[FairDeliveryPrice,FairFuturePrice,AccruedInterest] = fairdelivery(
InstrumentObject,SpotPrice,DiscountCurve)
```

Description

`[FairDeliveryPrice,FairFuturePrice,AccruedInterest] = fairdelivery( InstrumentObject,SpotPrice,DiscountCurve)` computes the fair delivery price of the underlying asset for a `BondFuture`, `CommodityFuture`, `FXFuture`, or `EquityIndexFuture` instrument object.

Examples

Compute Fair Delivery Price of Underlying Bond for `BondFuture` Instrument

This example shows the workflow to price a `BondFuture` instrument and then use `fairdelivery` to compute the fair delivery price for the underlying `FixedBond`.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2022,3,1);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

Create Underlying `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",Maturity=datetime(2032,9,1),CouponRate=0.05,Name="fixed_bond_ins
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
```

```
                  Holidays: NaT
                 IssueDate: NaT
           FirstCouponDate: NaT
            LastCouponDate: NaT
                 StartDate: NaT
                  Maturity: 01-Sep-2032
                      Name: "fixed_bond_instrument"
```

Create BondFuture Instrument Object

Use `fininstrument` to create a `BondFuture` instrument object.

```
BondFut = fininstrument("BondFuture",Maturity=datetime(2022,9,1),QuotedPrice=86,Bond=FixB,Convers
```

```
BondFut =
  BondFuture with properties:

            Maturity: 01-Sep-2022
         QuotedPrice: 86
                Bond: [1x1 fininstrument.FixedBond]
    ConversionFactor: 1.4300
            Notional: 100000
                Name: "bondfuture_instrument"
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=ZeroCurve,SpotPrice=125)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 125
```

Price BondFuture Instrument

Use `price` to compute the price and price result for the `BondFuture` instrument.

```
[Price,outPR] = price(outPricer,BondFut)
```

```
Price = -151.9270

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×4 table
     Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
```

```
    _______    _________________    _______________    _______________

    -151.93       1.2283e+05            85.893                0
```

Compute Fair Delivery Price

Use `fairdelivery` with the `BondFuture` instrument to compute the fair delivery price for the underlying `FixedBond`.

```
SpotPrice = 125; % Clean spot price for $100 face value of underlying bond.
[FairDeliveryPrice,FairFuturePrice,AccruedInterest] = fairdelivery(BondFut,SpotPrice,ZeroCurve)
```

```
FairDeliveryPrice = 1.2283e+05

FairFuturePrice = 85.8935

AccruedInterest = 0
```

Compute Fair Delivery Price for Multiple FXFuture Instruments

This example shows the workflow to price multiple `FXFuture` instruments and then use `fairdelivery` to compute the fair delivery price for the `FXFuture` instruments.

Create ratecurve Objects

Create `ratecurve` objects using `ratecurve` for the foreign and domestic zero curves.

```
% Define Foreign Zero Curve
Settle = datetime(2022, 3, 1);
ForeignZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ForeignZeroRates = [0.0031 0.0035 0.0047 0.0058 0.0062 0.0093 0.0128 0.0182 0.0223 0.0285]';
ForeignZeroDates = Settle + ForeignZeroTimes;
ForeignRC = ratecurve('zero', Settle, ForeignZeroDates, ForeignZeroRates);

% Define Domestic Zero Curve
DomesticZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
DomesticZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
DomesticZeroDates = Settle + DomesticZeroTimes;
DomesticRC = ratecurve('zero', Settle, DomesticZeroDates, DomesticZeroRates);
```

Create FXFuture Instrument Object

Use `fininstrument` to create a `FXFuture` instrument object for three FX Future instruments.

```
FXFut = fininstrument("FXFuture",Maturity=datetime([2022,9,1 ; 2022,10,1 ; 2022,11,1]),QuotedPric
```

```
FXFut=3×1 object
  3x1 FXFuture array with properties:

    Maturity
    QuotedPrice
    ForeignRateCurve
    Notional
    Name
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=DomesticRC,SpotPrice=0.79)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 0.7900
```

Price FXFuture Instruments

Use `price` to compute the prices and price results for the `FXFuture` instrument.

```
[Price,outPR] = price(outPricer,FXFut)
```

```
Price = 3×1
10^4 ×

    0.2162
   -0.5789
   -1.3732


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    2161.7       1.5817e+05            0.79084              0


ans=1×4 table
    Price    FairDeliveryPrice    FairFuturePrice    AccruedInterest
    _____    _________________    _______________    _______________

    -5789       1.5819e+05            0.79097              0


ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    -13732       1.5822e+05            0.7911               0
```

Compute Fair Delivery for FXFuture Instruments

Use `fairdelivery` with the `FXFuture` instruments to compute the fair delivery price for the `FXFuture` instruments.

```
SpotPrice = 0.79; % Quoted in domestic currency for one unit of foreign currency
[FairDeliveryPrice,FairFuturePrice,AccruedInterest] = fairdelivery(FXFut(1),SpotPrice,DomesticRC)

FairDeliveryPrice = 1.5817e+05

FairFuturePrice = 0.7908

AccruedInterest = 0

[FairDeliveryPrice,FairFuturePrice,AccruedInterest] = fairdelivery(FXFut(2),SpotPrice,DomesticRC)

FairDeliveryPrice = 1.5819e+05

FairFuturePrice = 0.7910

AccruedInterest = 0

[FairDeliveryPrice,FairFuturePrice,AccruedInterest] = fairdelivery(FXFut(3),SpotPrice,DomesticRC)

FairDeliveryPrice = 1.5822e+05

FairFuturePrice = 0.7911

AccruedInterest = 0
```

Input Arguments

`InstrumentObject` — Instrument object

`BondFuture` object | `CommodityFuture` object | `EquityIndexFuture` object | `FXFuture` object

Instrument object, specified using a previously created instrument object for one of the following: `BondFuture`, `CommodityFuture`, `FXFuture`, or `EquityIndexFuture`.

Note If the `InstrumentObject` is a vector of instruments, you must use `fairdelivery` separately with each instrument.

Data Types: `object`

`SpotPrice` — Quoted spot price for underlying asset to be delivered

numeric

Quoted spot price for underlying asset to be delivered, specified as `SpotPrice` and a numeric value that depends on the type of future instrument being priced:

- `BondFuture` instrument — Clean spot price quoted for $100 face value of underlying bond
- `CommodityFuture` instrument — Spot price for underlying commodity quantity specified in contract
- `EquityIndexFuture` instrument — Spot equity index value
- `FXFuture` instrument — Spot price quoted in domestic currency for one unit of foreign currency

Data Types: `double`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as `DiscountCurve` and the name of a previously created `ratecurve` object.

Data Types: `object`

Output Arguments

`FairDeliveryPrice` — Fair delivery price for underlying asset
numeric

Fair delivery price for underlying asset, returned as a numeric. Depending on `InstrumentObject`, the `FairDeliveryPrice` output is defined:

- `BondFuture` instrument — Fair delivery price (full cash price) for underlying bond
- `CommodityFuture` instrument — Fair delivery price for underlying commodity
- `EquityIndexFuture` instrument — Fair delivery price for equity index future
- `FXFuture` instrument — Fair delivery price for FX future in domestic currency

`FairFuturePrice` — Fair future price (clean price) for $100 face value
numeric

Fair future price (clean price) for 100 face value, returned as a numeric. Depending on `InstrumentObject`, the `FairFuturePrice` output is defined:

- `BondFuture` instrument — Fair future price (clean price) for $100 face value
- `CommodityFuture` instrument — Fair future price
- `EquityIndexFuture` instrument — Fair future price
- `FXFuture` instrument — Fair future price in domestic currency for one unit of foreign currency

`AccruedInterest` — Accrued interest at delivery
numeric

Accrued interest at delivery, returned as a numeric. Depending on `InstrumentObject`, the `AccruedInterest` output is defined:

- `BondFuture` instrument — Accrued interest at delivery for $100 face value
- `CommodityFuture` instrument — Accrued interest at delivery
- `EquityIndexFuture` instrument — Accrued interest at delivery
- `FXFuture` instrument — Accrued interest at delivery

Version History

Introduced in R2022a

See Also

`fininstrument` | `finmodel` | `finpricer` | `cashsettle`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

setCallExercisePolicy

Package: `fininstrument`

Set call exercise policy for `OptionEmbeddedFixedBond`, `OptionEmbeddedFloatBond`, or `ConvertibleBond` instrument

Syntax

```
UpdatedInstrumentObject = setCallExercisePolicy(InstrumentObject,
exerciseSchedule,exerciseStyle)
```

Description

`UpdatedInstrumentObject = setCallExercisePolicy(InstrumentObject, exerciseSchedule,exerciseStyle)` sets the call exercise policy for a `OptionEmbeddedFixedBond`, `OptionEmbeddedFloatBond`, or `ConvertibleBond` instrument object.

Examples

Set Call Exercise Policy for Option Embedded Fixed Bond Instrument

This example shows how to use `setCallExercisePolicy` to maintain consistency between the exercise schedule and exercise style when using a `OptionEmbeddedFixedBond` instrument object.

Create `OptionEmbeddedFixedBond` Instrument Object

Use `fininstrument` to create an `OptionEmbeddedFixedBond` instrument object with different exercise styles.

```
Maturity = datetime(2024,1,1);
Strike = [100;100];
ExerciseDates = [datetime(2020,1,1); datetime(2024,1,1)];
Period = 1;
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});

CallableBond = fininstrument("OptionEmbeddedFixedBond",'Maturity',Maturity,...
                             'CouponRate',0.025,'Period',Period, ...
                             'CallSchedule',CallSchedule)
```

```
CallableBond =
  OptionEmbeddedFixedBond with properties:

                 CouponRate: 0.0250
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
```

```
                 Holidays: NaT
                IssueDate: NaT
          FirstCouponDate: NaT
           LastCouponDate: NaT
                StartDate: NaT
                 Maturity: 01-Jan-2024
                CallDates: [2x1 datetime]
                 PutDates: [0x1 datetime]
             CallSchedule: [2x1 timetable]
              PutSchedule: [0x0 timetable]
        CallExerciseStyle: "american"
         PutExerciseStyle: [0x0 string]
                     Name: ""
```

Set the Exercise Style to Bermudan

Use `setCallExercisePolicy` to define the `CallExerciseStyle` as `Bermudan`.

```
CallableBond = setCallExercisePolicy(CallableBond, CallSchedule, "Bermudan")
```

```
CallableBond =
  OptionEmbeddedFixedBond with properties:

                CouponRate: 0.0250
                    Period: 1
                     Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
                   Maturity: 01-Jan-2024
                  CallDates: [2x1 datetime]
                   PutDates: [0x1 datetime]
               CallSchedule: [2x1 timetable]
                PutSchedule: [0x0 timetable]
          CallExerciseStyle: "bermudan"
           PutExerciseStyle: [0x0 string]
                       Name: ""
```

Use `setCallExercisePolicy` to modify Call`Schedule` and continue using a Bermudan exercise style.

```
Strike = [100; 101;102;103];
ExerciseDates = [datetime(2018,1,1);datetime(2020,1,1);datetime(2022,1,1);datetime(2024,1,1)];

CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});

CallableBond = setCallExercisePolicy(CallableBond, CallSchedule)
```

```
CallableBond =
  OptionEmbeddedFixedBond with properties:
```

```
                CouponRate: 0.0250
                    Period: 1
                     Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
                   Maturity: 01-Jan-2024
                  CallDates: [4x1 datetime]
                   PutDates: [0x1 datetime]
               CallSchedule: [4x1 timetable]
                PutSchedule: [0x0 timetable]
          CallExerciseStyle: "bermudan"
           PutExerciseStyle: [0x0 string]
                       Name: ""
```

Set Call Exercise Policy for Specific Option Embedded Fixed Bond Instrument

This example shows how to use `setCallExercisePolicy` to maintain consistency between the exercise schedule and exercise style when using a `OptionEmbeddedFixedBond` instrument object with three `OptionEmbeddedFixedBond` instruments.

Create `OptionEmbeddedFixedBond` Instrument Object

Use `fininstrument` to create an `OptionEmbeddedFixedBond` instrument object for three Option Embedded Fixed Bond instruments with `American` exercise styles.

```
Maturity = datetime([2024,1,1 ; 2024,4,1 ; 2024,8,1]);
Strike = [100;100];
ExerciseDates = [datetime(2020,1,1); datetime(2024,1,1)];
Period = 1;
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});

CallableBond = fininstrument("OptionEmbeddedFixedBond",'Maturity',Maturity,...
                             'CouponRate',0.025,'Period',Period, ...
                             'CallSchedule',CallSchedule)
```

```
CallableBond=3×1 object
  3x1 OptionEmbeddedFixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
```

```
    LastCouponDate
    StartDate
    Maturity
    CallDates
    PutDates
    CallSchedule
    PutSchedule
    CallExerciseStyle
    PutExerciseStyle
    Name


CallableBond.CallExerciseStyle

ans =
"american"

ans =
"american"

ans =
"american"
```

The `CallExerciseStyle` is `"American"` because the `fininstrument` syntax does not contain a `CallExercideStyle` specification and there are two exercise dates defined in the `CallSchedule`.

Set the Exercise Style to Bermudan

Use `setCallExercisePolicy` to define the `CallExerciseStyle` as `Bermudan` for the second (`CallableBond(2)`) instrument.

```
CallableBond(2) = setCallExercisePolicy(CallableBond(2), CallSchedule, "Bermudan")

CallableBond=3×1 object
  3x1 OptionEmbeddedFixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    CallDates
    PutDates
    CallSchedule
    PutSchedule
    CallExerciseStyle
    PutExerciseStyle
    Name


CallableBond.CallExerciseStyle
```

```
ans =
"american"

ans =
"bermudan"

ans =
"american"
```

Use `setCallExercisePolicy` to modify Call`Schedule` and continue using a `Bermudan` exercise style.

```
Strike = [100; 101;102;103];
ExerciseDates = [datetime(2018,1,1);datetime(2020,1,1);datetime(2022,1,1);datetime(2024,1,1)];

CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});

CallableBond(2) = setCallExercisePolicy(CallableBond(2), CallSchedule)

CallableBond=3×1 object
  3x1 OptionEmbeddedFixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    CallDates
    PutDates
    CallSchedule
    PutSchedule
    CallExerciseStyle
    PutExerciseStyle
    Name


CallableBond.CallExerciseStyle

ans =
"american"

ans =
"bermudan"
```

```
ans =
"american"
```

Input Arguments

`InstrumentObject` — Instrument object
`OptionEmbeddedFixedBond` object | `OptionEmbeddedFloatBond` object | `ConvertibleBond` object

Instrument object, specified as a previously created `OptionEmbeddedFixedBond`, `OptionEmbeddedFloatBond`, or `ConvertibleBond` object instrument object.

Note If the `OptionEmbeddedFixedBond`, `OptionEmbeddedFloatBond`, or `ConvertibleBond` instrument object is a vector of instruments, you must use `setCallExercisePolicy` separately with each instrument.

Data Types: `object`

`exerciseSchedule` — Call exercise schedule
timetable

Call exercise schedule, specified as a timetable. The timetable must contain both the `exerciseDate` value and `Strike` information.

Data Types: `timetable`

`exerciseStyle` — Call option exercise style
`"American"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'`

Call option exercise style, specified as a scalar string or character vector.

Data Types: `string` | `char`

Output Arguments

`UpdatedInstrumentObject` — Updated instrument
object

Updated instrument, returned as an object.

Version History

Introduced in R2020b

See Also

`fininstrument` | `finmodel` | `finpricer` | `timetable`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22

“Choose Instruments, Models, and Pricers” on page 1-53

setPutExercisePolicy

Package: `fininstrument`

Set put exercise policy for `OptionEmbeddedFixedBond`, `OptionEmbeddedFloatBond`, or `ConvertibleBond` instrument

Syntax

```
UpdatedInstrumentObject = setPutExercisePolicy(InstrumentObject,
exerciseSchedule,exerciseStyle)
```

Description

`UpdatedInstrumentObject = setPutExercisePolicy(InstrumentObject, exerciseSchedule,exerciseStyle)` sets the put exercise policy for an `OptionEmbeddedFixedBond`, `OptionEmbeddedFloatBond`, or `ConvertibleBond` instrument object.

Examples

Set Put Exercise Policy for Option Embedded Fixed Bond Instrument

This example shows how to use `setPutExercisePolicy` to maintain consistency between the exercise schedule and exercise style when using a `OptionEmbeddedFixedBond` instrument object.

Create `OptionEmbeddedFixedBond` Instrument Object

Use `fininstrument` to create an `OptionEmbeddedFixedBond` instrument object with different exercise styles.

```
Maturity = datetime(2024,1,1);
Strike = [100;100];
ExerciseDates = [datetime(2020,1,1); datetime(2024,1,1)];
Period = 1;
PutSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});

PuttableBond = fininstrument("OptionEmbeddedFixedBond",'Maturity',Maturity,...
                             'CouponRate',0.025,'Period',Period, ...
                             'PutSchedule',PutSchedule)
```

```
PuttableBond =
  OptionEmbeddedFixedBond with properties:

                 CouponRate: 0.0250
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
```

```
                  Holidays: NaT
                 IssueDate: NaT
           FirstCouponDate: NaT
            LastCouponDate: NaT
                 StartDate: NaT
                  Maturity: 01-Jan-2024
                 CallDates: [0x1 datetime]
                  PutDates: [2x1 datetime]
              CallSchedule: [0x0 timetable]
               PutSchedule: [2x1 timetable]
        CallExerciseStyle: [0x0 string]
         PutExerciseStyle: "american"
                     Name: ""
```

Set Exercise Style to Bermudan

Use `setPutExercisePolicy` to define `PutExerciseStyle` as `Bermudan`.

```
PuttableBond = setPutExercisePolicy(PuttableBond, PutSchedule, "Bermudan")
```

```
PuttableBond =
  OptionEmbeddedFixedBond with properties:

                CouponRate: 0.0250
                    Period: 1
                     Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
                   Maturity: 01-Jan-2024
                  CallDates: [0x1 datetime]
                   PutDates: [2x1 datetime]
               CallSchedule: [0x0 timetable]
                PutSchedule: [2x1 timetable]
          CallExerciseStyle: [0x0 string]
           PutExerciseStyle: "bermudan"
                       Name: ""
```

Use `setPutExercisePolicy` to modify `PutSchedule` and continue using a Bermudan exercise style.

```
Strike = [100; 101;102;103];
ExerciseDates = [datetime(2018,1,1);datetime(2020,1,1);datetime(2022,1,1);datetime(2024,1,1)];

PutSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});

PuttableBond = setPutExercisePolicy(PuttableBond, PutSchedule)
```

```
PuttableBond =
  OptionEmbeddedFixedBond with properties:
```

```
                CouponRate: 0.0250
                    Period: 1
                     Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
                   Maturity: 01-Jan-2024
                  CallDates: [0x1 datetime]
                   PutDates: [4x1 datetime]
               CallSchedule: [0x0 timetable]
                PutSchedule: [4x1 timetable]
          CallExerciseStyle: [0x0 string]
           PutExerciseStyle: "bermudan"
                       Name: ""
```

Set Put Exercise Policy for Specific Option Embedded Fixed Bond Instrument

This example shows how to use `setPutExercisePolicy` to maintain consistency between the exercise schedule and exercise style when using a `OptionEmbeddedFixedBond` instrument object with three `OptionEmbeddedFixedBond` instruments.

Create `OptionEmbeddedFixedBond` Instrument Object

Use `fininstrument` to create an `OptionEmbeddedFixedBond` instrument object for three Option Embedded Fixed Bond instruments with `American` exercise styles.

```
Maturity = datetime([2024,1,1 ; 2024,4,1 ; 2024,8,1]);
Strike = [100;100];
ExerciseDates = [datetime(2020,1,1); datetime(2024,1,1)];
Period = 1;
PutSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});

PuttableBond = fininstrument("OptionEmbeddedFixedBond",'Maturity',Maturity,...
                             'CouponRate',0.025,'Period',Period, ...
                             'PutSchedule',PutSchedule)
```

```
PuttableBond=3×1 object
  3x1 OptionEmbeddedFixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
```

```
    LastCouponDate
    StartDate
    Maturity
    CallDates
    PutDates
    CallSchedule
    PutSchedule
    CallExerciseStyle
    PutExerciseStyle
    Name


PuttableBond.PutExerciseStyle

ans =
"american"

ans =
"american"

ans =
"american"
```

The `PutExerciseStyle` is `"American"` because the `fininstrument` syntax does not contain a `PutExercideStyle` specification and there are two exercise dates defined in the `PutSchedule`.

Set Exercise Style to Bermudan

Use `setPutExercisePolicy` to define `PutExerciseStyle` as `Bermudan` for the second (`PuttableBond(2)`) instrument.

```
PuttableBond(2) = setPutExercisePolicy(PuttableBond(2), PutSchedule, "Bermudan")

PuttableBond=3×1 object
  3x1 OptionEmbeddedFixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    CallDates
    PutDates
    CallSchedule
    PutSchedule
    CallExerciseStyle
    PutExerciseStyle
    Name


PuttableBond.PutExerciseStyle
```

```
ans =
"american"

ans =
"bermudan"

ans =
"american"
```

Use `setPutExercisePolicy` to modify `PutSchedule` and continue using a `Bermudan` exercise style.

```
Strike = [100; 101;102;103];
ExerciseDates = [datetime(2018,1,1);datetime(2020,1,1);datetime(2022,1,1);datetime(2024,1,1)];

PutSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});

PuttableBond(2) = setPutExercisePolicy(PuttableBond(2), PutSchedule)

PuttableBond=3×1 object
  3x1 OptionEmbeddedFixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    CallDates
    PutDates
    CallSchedule
    PutSchedule
    CallExerciseStyle
    PutExerciseStyle
    Name


PuttableBond.PutExerciseStyle

ans =
"american"

ans =
"bermudan"
```

```
ans =
"american"
```

Input Arguments

`InstrumentObject` — Instrument object
`OptionEmbeddedFixedBond` object | `OptionEmbeddedFloatBond` object | `ConvertibleBond` object

Instrument object, specified using a previously created `OptionEmbeddedFixedBond`, `OptionEmbeddedFloatBond`, or `ConvertibleBond` instrument object.

Note If the `OptionEmbeddedFixedBond`, `OptionEmbeddedFloatBond`, or `ConvertibleBond` instrument object is a vector of instruments, you must use `setCallExercisePolicy` separately with each instrument.

Data Types: `object`

`exerciseSchedule` — Put exercise schedule
timetable

Put exercise schedule, specified as a timetable. The timetable must contain both the `exerciseDate` value and `Strike` information.

Data Types: `timetable`

`exerciseStyle` — Put option exercise style
`"American"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'`

Put option exercise style, specified as a scalar string or character vector.

Data Types: `string` | `char`

Output Arguments

`UpdatedInstrumentObject` — Updated instrument
object

Updated instrument, returned as an object.

Version History

Introduced in R2020b

See Also

`fininstrument` | `finmodel` | `finpricer` | `timetable`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22

“Choose Instruments, Models, and Pricers” on page 1-53

setExercisePolicy

Package: `fininstrument`

Set exercise policy for `FixedBondOption`, `FloatBondOption`, or `Vanilla` instrument

Syntax

```
UpdatedInstrumentObject = setExercisePolicy(InstrumentObject,exerciseDate,
Strike,exerciseStyle)
```

Description

`UpdatedInstrumentObject = setExercisePolicy(InstrumentObject,exerciseDate, Strike,exerciseStyle)` sets the exercise policy for a `FixedBondOption`, `FloatBondOption`, or `Vanilla` instrument object.

Examples

Set Exercise Policy for Fixed Bond Option Instrument

This example shows how to use `setExercisePolicy` to maintain consistency between the exercise schedule and exercise style when using a `FixedBondOption` instrument.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",'Maturity',datetime(2029,9,15),'CouponRate',.021,'Period',1
```

Create FixedBondOption Instrument Object

Use `fininstrument` to create a callable `FixedBondOption` instrument object with a European exercise.

```
FixedBOption = fininstrument("FixedBondOption",'ExerciseDate',datetime(2025,9,15),'Strike',98,'Bo
```

```
FixedBOption =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2025
           Strike: 98
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option"
```

Set the Exercise Style to American

Use `setExercisePolicy` to define `ExerciseStyle` as `American`.

```
FixedBOption = setExercisePolicy(FixedBOption,datetime(2025,9,15),98,"American")
```

```
FixedBOption =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2025
           Strike: 98
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option"
```

Set Exercise Policy for a Specific Fixed Bond Option Instrument

This example shows how to use `setExercisePolicy` to maintain consistency between the exercise schedule and exercise style when using a `FixedBondOption` instrument object with three Fixed Bond instruments.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",'Maturity',datetime(2029,9,15),'CouponRate',.021,'Period',1
```

Create `FixedBondOption` Instrument Object

Use `fininstrument` to create a callable `FixedBondOption` instrument object for three Fixed Bond Option instruments with European exercises.

```
FixedBOption = fininstrument("FixedBondOption",'ExerciseDate',datetime([2025,9,15 ; 2025,10,15 ;
```

```
FixedBOption=3×1 object
  3x1 FixedBondOption array with properties:

    OptionType
    ExerciseStyle
    ExerciseDate
    Strike
    Bond
    Name
```

Set the Exercise Style to American

Use `setExercisePolicy` to define `ExerciseStyle` as `American` for the second (`FixedBOption(2)`) instrument.

```
FixedBOption(2) = setExercisePolicy(FixedBOption(2),datetime(2025,9,15),98,"American")
```

```
FixedBOption=3×1 object
  3x1 FixedBondOption array with properties:

    OptionType
    ExerciseStyle
    ExerciseDate
    Strike
```

```
    Bond
    Name


FixedBOption(2).ExerciseStyle

ans =
"american"

FixedBOption.ExerciseStyle

ans =
"european"

ans =
"american"

ans =
"european"
```

Input Arguments

`InstrumentObject` — Instrument object

`FixedBondOption` object | `FloatBondOption` object | `Vanilla` object

Instrument object, specified using a previously created `FixedBondOption`, `FloatBondOption`, or `Vanilla` instrument object.

Note If the `FixedBondOption`, `FloatBondOption`, or `Vanilla` instrument object is a vector of instruments, you must use `setExercisePolicy` separately with each instrument.

Data Types: `object`

`exerciseDate` — Exercise date

datetime

Exercise date, specified as a scalar datetime.

Data Types: `datetime`

`Strike` — Strike

numeric

Strike, specified as a scalar numeric.

Data Types: `double`

`exerciseStyle` — Option exercise style

`"American"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'`

Option exercise style, specified as a scalar string or character vector.

Data Types: `string` | `char`

Output Arguments

`UpdatedInstrumentObject` — Updated instrument object
object

Updated instrument object, returned as an object.

Version History

Introduced in R2020b

See Also

`fininstrument` | `finmodel` | `finpricer` | `timetable`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

parswaprate

Package: `fininstrument`

Compute par swap rate for `Swap` instrument

Syntax

```
outRate = parswaprate(SwapObject,inCurve)
```

Description

`outRate = parswaprate(SwapObject,inCurve)` computes a par swap rate for a `Swap` instrument object.

Examples

Compute Par Swap Rate for Vanilla Swap Instrument Using ratecurve and Discount Pricer

This example shows the workflow to compute the par swap rate for a vanilla `Swap` instrument when you use a `ratecurve` and a `Discount` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `Swap` instrument.

```
Settle = datetime(2018,3,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Mar-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create a vanilla `Swap` instrument object.

```
Swap = fininstrument("Swap",'Maturity',datetime(2020,9,15),'LegRate',[0.022 0.019 ],'LegType',["
```

```
Swap =
  Swap with properties:

                     LegRate: [0.0220 0.0190]
                     LegType: ["float"    "fixed"]
                       Reset: [2 2]
                       Basis: [0 0]
                    Notional: 100
           LatestFloatingRate: [NaN NaN]
                  ResetOffset: [0 0]
      DaycountAdjustedCashFlow: [0 0]
               ProjectionCurve: [1x2 ratecurve]
         BusinessDayConvention: ["actual"    "actual"]
                     Holidays: NaT
                 EndMonthRule: [1 1]
                    StartDate: NaT
                     Maturity: 15-Sep-2020
                         Name: "swap_instrument"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `Swap` Instrument

Use `price` to compute the price and sensitivities for the vanilla `Swap` instrument.

```
[Price, outPR] = price(outPricer, Swap,["all"])
```

```
Price = 2.4066
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    _________

    2.4066    -0.024499
```

Compute the par swap rate using `parswaprate`.

```
outRate = parswaprate(Swap,myRC)
```

```
outRate = 0.0287
```

Compute Par Swap Rate for Multiple Vanilla Swap Instruments Using ratecurve and Discount Pricer

This example shows the workflow to compute the par swap rate for multiple vanilla `Swap` instruments when you use a `ratecurve` and a `Discount` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `Swap` instrument.

```
Settle = datetime(2019,4,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Apr-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create a vanilla `Swap` instrument object for three Swap instruments.

```
Swap = fininstrument("Swap",'Maturity',datetime([2020,4,15 ; 2021,4,15 ; 2024,4,15]),'LegRate',[0
```

```
Swap=3×1 object
  3x1 Swap array with properties:

    LegRate
    LegType
    Reset
    Basis
    Notional
    LatestFloatingRate
    ResetOffset
    DaycountAdjustedCashFlow
    ProjectionCurve
    BusinessDayConvention
    Holidays
```

```
    EndMonthRule
    StartDate
    Maturity
    Name
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price Swap Instruments

Use `price` to compute the prices and sensitivities for the vanilla `Swap` instruments.

```
[Price, outPR] = price(outPricer, Swap,["all"])
```

```
Price = 3×1

    0.8473
    1.8067
    7.2322


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×2 table
     Price        DV01
    _______    __________

    0.84728    -0.0099228


ans=1×2 table
    Price       DV01
    ______    _________

    1.8067    -0.019656


ans=1×2 table
    Price       DV01
    ______    ________
```

```
    7.2322    -0.04664
```

Compute the par swap rate for each of the three Swap instruments using `parswaprate`.

```
outRate = parswaprate(Swap(1),myRC)
```

```
outRate = 0.0275
```

```
outRate = parswaprate(Swap(2),myRC)
```

```
outRate = 0.0281
```

```
outRate = parswaprate(Swap(3),myRC)
```

```
outRate = 0.0338
```

Input Arguments

`SwapObject` — Swap object
`Swap` object

Swap object, specified using a previously created `Swap` instrument object.

Note If the `SwapObject` is a vector of instruments, you must use `parswaprate` separately with each instrument.

Data Types: `object`

`inCurve` — Rate curve
`ratecurve` object

Rate curve, specified as a previously created `ratecurve` object.

Data Types: `object`

Output Arguments

`outRate` — Par swap rate
decimal

Par swap rate, returned as a decimal.

More About

Par Swap Rate

The par swap rate is the rate that renders a swap value equal to zero.

In other words, the par swap rate is the value of the fixed rate that gives the swap a zero present value, or the fixed rate that makes the value of both legs equal (that is, the value of the fixed leg and the value of the floating leg).

Version History

Introduced in R2020b

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

volatilities

Package: `finpricer.analytic`

Compute implied volatilities when using SABR pricer

Syntax

```
outVolatilities = volatilities(inpPricer,ExerciseDate,ForwardValue,Strike)
```

Description

`outVolatilities = volatilities(inpPricer,ExerciseDate,ForwardValue,Strike)` computes implied volatilities for a `Swaption` instrument when you use a SABR pricer.

Examples

Compute Implied Volatilities for Swaption Instrument

This example shows the workflow to compute implied volatilities for a `Swaption` Instrument.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
ValuationDate = datetime(2016,3,5);
ZeroDates = datemnth(ValuationDate,[1 2 3 6 9 12*[1 2 3 4 5 6 7 8 9 10 12]])';
ZeroRates = [-0.33 -0.28 -0.24 -0.12 -0.08 -0.03 0.015 0.028 ...
             0.033 0.042 0.056 0.095 0.194 0.299 0.415 0.525]'/100;
Compounding = 1;
ZeroCurve = ratecurve("zero",ValuationDate,ZeroDates,ZeroRates,'Compounding',Compounding)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: 1
               Basis: 0
               Dates: [16x1 datetime]
               Rates: [16x1 double]
              Settle: 05-Mar-2016
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create SABR Model Object

Use `finmodel` to create a SABR model object.

```
Alpha = 0.0135;
Beta = 0.5;
```

```
Rho = 0.4654;
Nu = 0.4957;
Shift = 0.008;

SABRModel = finmodel("SABR",'Alpha',Alpha,'Beta',Beta,'Rho',Rho,'Nu',Nu,'Shift',Shift)
```

```
SABRModel =
  SABR with properties:

            Alpha: 0.0135
             Beta: 0.5000
              Rho: 0.4654
               Nu: 0.4957
            Shift: 0.0080
    VolatilityType: "black"
```

Create SABR Pricer Object

Use `finpricer` to create a SABR pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
SABRPricer = finpricer("Analytic", 'Model', SABRModel,'DiscountCurve', ZeroCurve)
```

```
SABRPricer =
  SABR with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.SABR]
```

Compute Implied Volatilities

Use `volatilities` to compute the implied volatilities for a `Swaption` instrument.

```
SwaptionExerciseDate = datetime(2017,3,5);
ForwardValue = 0.0007;
StrikeGrid = [-0.5; -0.25; -0.125; 0; 0.125; 0.25; 0.5; 1.0; 1.5]/100;
MarketStrikes = ForwardValue + StrikeGrid;

outVolatilities = volatilities(SABRPricer, SwaptionExerciseDate, ForwardValue, MarketStrikes)
```

```
outVolatilities = 9×1

    0.2132
    0.1500
    0.1409
    0.1474
    0.1609
    0.1752
    0.2004
    0.2372
```

```
    0.2627
```

Input Arguments

`inpPricer` — SABR pricer
SABR pricer object

SABR pricer, specified as a previously created SABR pricer object.

Data Types: `object`

`ExerciseDate` — Option exercise date
datetime scalar | string scalar | date character vector

Option exercise date, specified as a scalar datetime, string, or date character vector.

To support existing code, `volatilities` also accepts serial date numbers as inputs, but they are not recommended.

`ForwardValue` — Forward swap rate
decimal

Forward swap rate, specified as a scalar decimal.

Data Types: `object`

`Strike` — Strike rate
decimal

Strike rate, specified as a scalar decimal.

Data Types: `double`

Output Arguments

`outVolatilities` — Output volatilities
numeric

Output volatilities, returned as a numeric.

Version History

Introduced in R2020b

Serial date numbers not recommended
Not recommended starting in R2022b

Although `volatilities` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`fininstrument` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

fitSvensson

Fit Svensson model to bond market data

Syntax

```
outCurve = fitSvensson(Settle,Instruments,CleanPrice)
```

Description

`outCurve = fitSvensson(Settle,Instruments,CleanPrice)` fits a Svensson model to bond data.

Examples

Fit Svensson Model to Bond Market Data

Define the bond data and use `fininstrument` to create `FixedBond` instrument objects.

```
 Settle = datetime(2017,9,15);
  Maturity = [datetime(2019,9,15);datetime(2021,9,15);...
      datetime(2023,9,15);datetime(2026,9,7);...
      datetime(2035,9,15);datetime(2047,9,15)];

  CleanPrice = [100.1;100.1;100.8;96.6;103.3;96.3];
  CouponRate = [0.0400;0.0425;0.0450;0.0400;0.0500;0.0425];

nInst = numel(CouponRate);
Bonds(nInst,1) = fininstrument.FinInstrument;
for ii=1:nInst
    Bonds(ii) = fininstrument("FixedBond",'Maturity',Maturity(ii),...
        'CouponRate',CouponRate(ii));
end
```

Use `fitSvensson` to create a `parametercurve` object.

```
SvenModel = fitSvensson(Settle,Bonds,CleanPrice)
```

```
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

SvenModel =
  parametercurve with properties:

             Type: "zero"
           Settle: 15-Sep-2017
      Compounding: -1
            Basis: 0
   FunctionHandle: @(t)fitF(Params,t)
       Parameters: [3.3043e-08 0.0197 0.0624 0.1391 1.3563 11.7741]
```

Input Arguments

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `fitSvensson` also accepts serial date numbers as inputs, but they are not recommended.

`Instruments` — Bond instrument objects

array

Bond instrument objects, specified as an array of bond instrument objects.

Data Types: `object`

`CleanPrice` — Observed market prices

vector

Observed market prices, specified as a vector.

Data Types: `double`

Output Arguments

`outCurve` — Fitted Svensson model

object

Fitted Svensson model, returned as a `parametercurve` object.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `fitSvensson` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`discountfactors` | `zerorates` | `forwardrates`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

discountfactors

Calculate discount factors for a `ratecurve` object

Syntax

```
outDF = discountfactors(obj,inpDates)
```

Description

`outDF = discountfactors(obj,inpDates)` calculates the discount factors for a `ratecurve` object.

Examples

Calculate Discount Factors for a ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates,'Compounding',2,'Basis',5,'InterpMethod',"pch
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: 2
               Basis: 5
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "pchip"
   ShortExtrapMethod: "linear"
    LongExtrapMethod: "pchip"
```

Compute the discount factors using `discountfactors`.

```
CurveSettle = datetime(2019,9,15);
outRates = discountfactors(myRC,CurveSettle+30:30:CurveSettle+720)
```

```
outRates = 1×24

    0.9996    0.9992    0.9988    0.9983    0.9979    0.9974    0.9970    0.9965    0.9961    0.9
```

Input Arguments

`obj` — `ratecurve` object

`ratecurve` object

`ratecurve` object, specified using a previously created `ratecurve` object.

Data Types: `object`

`inpDates` — Input dates

datetime array | string array | date character vector

Input dates, specified as a scalar or an `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `discountfactors` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`outDF` — Discount factors

numeric

Discount factors, returned as a numeric.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `discountfactors` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`ratecurve` | `forwardrates` | `zerorates` | `irbootstrap`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22

“Choose Instruments, Models, and Pricers” on page 1-53

forwardrates

Calculate forward rates for `ratecurve` object

Syntax

```
outRates = forwardrates(obj,startDates,endDates)
outRates = forwardrates( ___ ,inpComp,inpBasis)
```

Description

`outRates = forwardrates(obj,startDates,endDates)` calculates forward rates for the `ratecurve` object (`obj`) based on the `startDates` and `endDates`.

`outRates = forwardrates( ___ ,inpComp,inpBasis)` optionally specifies the input compounding frequency (`inpComp`) and the input day-count basis (`inpBasis`) in addition to any of the input argument combinations in the previous syntax.

Examples

Calculate Forward Rates for `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates,'Compounding',2,'Basis',5,'InterpMethod',"pchi
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: 2
               Basis: 5
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "pchip"
   ShortExtrapMethod: "linear"
    LongExtrapMethod: "pchip"
```

Compute the forward rates using `forwardrates`.

```
outRates = forwardrates(myRC,datetime(2019,12,15),datetime(2021,9,15),6,7)
```

```
outRates = 0.0062
```

Input Arguments

`obj` — `ratecurve` object

`ratecurve` object

`ratecurve` object, specified using a previously created `ratecurve` object.

Data Types: `object`

`startDates` — Start dates of interval to discount over

datetime array | string array | date character vector

Start dates of the interval to discount over, specified as a scalar or an `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors. `startDates` must be earlier than `endDates`.

To support existing code, `forwardrates` also accepts serial date numbers as inputs, but they are not recommended.

`endDates` — Maturity dates ending interval to discount over

datetime array | string array | date character vector

Maturity dates ending the interval to discount over, specified as a scalar or an `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `forwardrates` also accepts serial date numbers as inputs, but they are not recommended.

`inpComp` — Input compounding frequency

`Compounding` for the `ratecurve` object (default) | possible values include: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

(Optional) Input compounding frequency, specified as a scalar numeric using one of the supported values: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`inpBasis` — Input day-count basis

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Input day-count basis, specified as a scalar integer.

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

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

`outRates` — Forward rates

numeric

Forward rates, returned as a numeric.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `forwardrates` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`ratecurve` | `discountfactors` | `zerorates` | `irbootstrap`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

zerorates

Calculate zero rates for `ratecurve` object

Syntax

```
outRates = zerorates(obj,inpDates)
outRates = zerorates( ___ ,inpComp,inpBasis)
```

Description

`outRates = zerorates(obj,inpDates)` computes zero rates for the `ratecurve` object (`obj`) based on `inpDates`.

`outRates = zerorates( ___ ,inpComp,inpBasis)` optionally specifies the input compounding frequency (`inpComp`) and the input day-count basis (`inpBasis`) in addition to any of the input argument combinations in the previous syntax.

Examples

Calculate Zero Rates for `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates,'Compounding',2,'Basis',5,'InterpMethod',"pchi
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: 2
               Basis: 5
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "pchip"
    ShortExtrapMethod: "linear"
     LongExtrapMethod: "pchip"
```

Compute the zero rates using `zerorates`.

```
CurveSettle = datetime(2019,9,15);
outRates = zerorates(myRC,CurveSettle+30:30:CurveSettle+720)
```

```
outRates = 1×24
```

```
    0.0049    0.0050    0.0050    0.0051    0.0051    0.0052    0.0052    0.0053    0.0053    0.0
```

Input Arguments

`obj` — `ratecurve` object

`ratecurve` object

`ratecurve` object, specified using a previously created `ratecurve` object.

Data Types: `object`

`inpDates` — Input dates

datetime array | string array | date character vector

Input dates, specified as a scalar or an `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `zerorates` also accepts serial date numbers as inputs, but they are not recommended.

`inpComp` — Input compounding frequency

`Compounding` for the `ratecurve` object (default) | possible values include: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

(Optional) Input compounding frequency, specified as a scalar numeric using one of the supported values: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`inpBasis` — Input day-count basis

`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Input day-count basis, specified as a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

`outRates` — Zero rates

numeric

Zero rates, returned as a numeric.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `zerorates` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`ratecurve` | `forwardrates` | `discountfactors` | `irbootstrap`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

discountfactors

Calculate discount factors for `parametercurve` object

Syntax

```
outDF = discountfactors(obj,inpDates)
```

Description

`outDF = discountfactors(obj,inpDates)` calculates the discount factors for a `parametercurve` object.

Examples

Calculate Discount Factors for parametercurve Object

Create a `parametercurve` object using `parametercurve`.

```
PCobj = parametercurve('zero',datetime(2019,9,15),@(t)polyval([-0.0001 0.003 0.02],t),'Compoundi
```

```
PCobj =
  parametercurve with properties:

             Type: "zero"
           Settle: 15-Sep-2019
      Compounding: 4
            Basis: 5
    FunctionHandle: @(t)polyval([-0.0001,0.003,0.02],t)
        Parameters: [-1.0000e-04 0.0030 0.0200]
```

Compute the discount factors using `discountfactors`.

```
CurveSettle = datetime(2019,9,15);
outDF = discountfactors(PCobj,CurveSettle+30:30:CurveSettle+720)
```

```
outDF = 1×24

    0.9983    0.9967    0.9949    0.9932    0.9914    0.9895    0.9876    0.9857    0.9838    0.9
```

Input Arguments

obj — parametercurve object

`parametercurve` object

`parametercurve` object, specified as a previously created `parametercurve` object.

Data Types: `object`

`inpDates` — Input dates
datetime array | string array | date character vector

Input dates, specified as a scalar or an `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `discountfactors` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`outDF` — Discount factors
numeric

Discount factors, returned as a numeric.

Version History

Introduced in R2020a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `discountfactors` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`ratecurve` | `zerorates` | `forwardrates` | `fitNelsonSiegel` | `fitSvensson`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

forwardrates

Calculate forward rates for `parametercurve` object

Syntax

```
outRates = forwardrates(obj,startDates,endDates)
outRates = forwardrates( ___ ,inpComp,inpBasis)
```

Description

`outRates = forwardrates(obj,startDates,endDates)` computes forward rates for the `parametercurve` object (`obj`) based on `startDates` and `endDates`.

`outRates = forwardrates( ___ ,inpComp,inpBasis)` specifies options in addition to any of the input argument combinations in the previous syntax.

Examples

Calculate Forward Rates for `parametercurve` Object

Create a `parametercurve` object using `parametercurve`.

```
PCobj = parametercurve('zero',datetime(2019,9,15),@(t)polyval([-0.0001 0.003 0.02],t),'Compoundi
```

```
PCobj =
  parametercurve with properties:

              Type: "zero"
            Settle: 15-Sep-2019
       Compounding: 4
             Basis: 5
    FunctionHandle: @(t)polyval([-0.0001,0.003,0.02],t)
        Parameters: [-1.0000e-04 0.0030 0.0200]
```

Compute the forward rates using `forwardrates`.

```
CurveSettle = datetime(2019,9,15);
outRates = forwardrates(PCobj,datetime(2019,12,15),datetime(2020,9,15),6,7)
```

```
outRates = 0.0236
```

Input Arguments

`obj` — `parametercurve` object

`parametercurve` object

`parametercurve` object, specified as a previously created `parametercurve` object.

Data Types: `object`

`startDates` — Start dates of interval to discount over
datetime array | string array | date character vector

Start dates of the interval to discount over, specified as a scalar or an `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors. `startDates` must be earlier than `endDates`.

To support existing code, `forwardrates` also accepts serial date numbers as inputs, but they are not recommended.

`endDates` — Maturity dates ending the interval to discount over
datetime array | string array | date character vector

Maturity dates ending the interval to discount over, specified as a scalar or an `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `forwardrates` also accepts serial date numbers as inputs, but they are not recommended.

`inpComp` — Input compounding frequency
`Compounding` for the `parametercurve` object (default) | possible values include: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

(Optional) Input compounding frequency, specified as a scalar numeric using one of the supported values: `–1`, `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`inpBasis` — Input day-count basis
`0` (actual/actual) (default) | integer from `0` to `13`

(Optional) Input day-count basis, specified as a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

Output Arguments

`outRates` — Forward rates
numeric

Forward rates, returned as a numeric.

Version History

Introduced in R2020a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `forwardrates` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`ratecurve` | `discountfactors` | `zerorates` | `fitNelsonSiegel` | `fitSvensson`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

zerorates

Calculate zero rates for `parametercurve` object

Syntax

```
outRates = zerorates(obj,inpDates)
outRates = zerorates( ___ ,inpComp,inpBasis)
```

Description

`outRates = zerorates(obj,inpDates)` computes zero rates for the `parametercurve` object (`obj`) based on `inpDates`.

`outRates = zerorates( ___ ,inpComp,inpBasis)` specifies options in addition to any of the input argument combinations in the previous syntax.

Examples

Calculate Zero Rates for `parametercurve` Object

Create a `parametercurve` object using `parametercurve`.

```
PCobj = parametercurve('zero',datetime(2019,9,15),@(t)polyval([-0.0001 0.003 0.02],t),'Compoundi
```

```
PCobj =
  parametercurve with properties:

             Type: "zero"
           Settle: 15-Sep-2019
      Compounding: 4
            Basis: 5
   FunctionHandle: @(t)polyval([-0.0001,0.003,0.02],t)
       Parameters: [-1.0000e-04 0.0030 0.0200]
```

Compute the zero rates using `zerorates`.

```
CurveSettle = datetime(2019,9,15);
outRates = zerorates(PCobj,CurveSettle+30:30:CurveSettle+720)
```

```
outRates = 1×24

    0.0202    0.0205    0.0207    0.0210    0.0212    0.0215    0.0217    0.0219    0.0222    0.0
```

Input Arguments

`obj` — `parametercurve` object

`parametercurve` object

parametercurve object, specified as a previously created parametercurve object.

Data Types: object

inpDates — Input dates
datetime array | string array | date character vector

Input dates, specified as a scalar or an NPOINTS-by-1 vector using a datetime array, string array, or date character vectors. StartDates must be earlier than EndDates.

To support existing code, zerorates also accepts serial date numbers as inputs, but they are not recommended.

inpComp — Input compounding frequency
Compounding for the parametercurve object (default) | possible values include: –1, 0, 1, 2, 3, 4, 6, 12.

(Optional) Input compounding frequency, specified as a scalar numeric using one of the supported values: –1, 0, 1, 2, 3, 4, 6, or 12.

Data Types: double

inpBasis — Input day-count basis
0 (actual/actual) (default) | integer from 0 to 13

(Optional) Input day-count basis, specified as a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: double

Output Arguments

outRates — Zero rates
numeric

Zero rates, returned as a numeric.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `zerorates` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`ratecurve` | `discountfactors` | `forwardrates` | `fitNelsonSiegel` | `fitSvensson`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

finportfolio

Create a `finportfolio` object

Description

Create a `finportfolio` object for a collection of instrument objects.

After creating instruments, models, and pricer objects, use `finportfolio` to create a `finportfolio` object for a collection of instruments. For more detailed information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
finportfolio_obj = finportfolio
finportfolio_obj = finportfolio(inInstruments)
finportfolio_obj = finportfolio(inInstruments,inPricers)
finportfolio_obj = finportfolio( ___ ,inQuant)
```

Description

`finportfolio_obj = finportfolio` creates an empty `finportfolio` object.

`finportfolio_obj = finportfolio(inInstruments)` creates a `finportfolio` object containing the instrument objects `inInstruments`.

`finportfolio_obj = finportfolio(inInstruments,inPricers)` creates a `finportfolio` object containing the instrument objects `inInstruments` and the pricer objects `inPricers`.

`finportfolio_obj = finportfolio( ___ ,inQuant)` optionally sets the `inQuant` property which specifies the number of instruments. Use this syntax with any of the input argument combinations in previous syntaxes to set the properties on page 11-2437 for the `finportfolio` object. For example, `finportfolio_obj = finportfolio([CapObj,FloorObj,SwaptionObj], [BlackPricerObj,NormalPricerObj,SabrPricerObj])` creates a `finportfolio` object that contains instrument and pricer objects.

Input Arguments

`inInstruments` — Instrument objects in the portfolio
scalar Instrument object | array of Instrument objects

Instrument objects in the portfolio, specified as a scalar Instrument object or an array of Instrument objects.

Data Types: `object`

`inPricers` — Pricer objects in the portfolio
scalar Pricer object | array of Pricer objects

Pricer objects in the portfolio, specified as a scalar Pricer object or an array of Pricer objects.

Data Types: `object`

`inQuant` — Number of instruments
positive or negative numeric | array of positive or negative numerics

Number of instruments, specified as a scalar numeric or an `NINST`-by-`1` array of numeric values. Use a positive value for long positions and a negative value for short positions.

Data Types: `double`

Properties

`Instruments` — Instrument objects in the portfolio
scalar instrument object | array of instrument objects

Instrument objects in the portfolio, returned as a scalar instrument object or an array of instrument objects.

Data Types: `struct`

`Pricers` — Pricer objects in the portfolio
scalar pricer object | array of pricer objects

Pricer objects in the portfolio, returned as a scalar pricer object or an array of pricer objects.

Data Types: `struct`

`PricerIndex` — Mapping of instrument objects to pricer objects in the portfolio
numeric

This property is read-only.

Mapping of instrument objects to pricer objects in the portfolio, returned as numeric.

`PricerIndex` has a length equal to the number of instrument objects in the `finportfolio` object and stores an index of which pricer is used for each instrument object.

Data Types: `struct`

`Quantity` — Number of instruments
numeric

Number of instruments, returned as a scalar numeric or numeric array.

Data Types: `double`

Object Functions

pricePortfolio	Compute price and sensitivities for portfolio of instruments

addInstrument	Add instrument to portfolio of instruments
removeInstrument	Remove instrument from portfolio of instruments
setPricer	Set pricer for finportfolio object

Examples

Price Portfolio of Heterogeneous Instruments

Use `finportfolio` and `pricePortfolio` to create and price a portfolio containing a `FixedBond` instrument and an American `Vanilla` option instrument.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",'Maturity',datetime(2022,9,15),'CouponRate',0.05,'Name',"fixed_b
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 0
              Dates: [10x1 datetime]
              Rates: [10x1 double]
```

```
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object for `FixedBond` Instrument

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
FBPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
FBPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Create `Vanilla` Instrument Object

Use `fininstrument` to create an American `Vanilla` instrument object.

```
Maturity = datetime(2023,9,15);
AmericanOpt = fininstrument("Vanilla",'ExerciseDate',Maturity,'Strike',120,'ExerciseStyle',"amer
```

```
AmericanOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2023
           Strike: 120
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object for `Vanilla` Instrument

Use `finmodel` to create a `BlackScholes` model object.

```
BSModel = finmodel("BlackScholes",'Volatility',0.12)
```

```
BSModel =
  BlackScholes with properties:

     Volatility: 0.1200
    Correlation: 1
```

Create `BjerksundStensland` Pricer Object for `Vanilla` Instrument

Use `finpricer` to create an analytic pricer object for the `BjerksundStensland` pricing method and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
BJSPricer = finpricer("analytic",'Model',BSModel,'DiscountCurve',myRC,'SpotPrice',100,'DividendVa
```

```
BJSPricer =
  BjerksundStensland with properties:
```

```
    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0200
     DividendType: "continuous"
```

Add the Instruments to a `finportfolio` Object

Create a `finportfolio` object using `finportfolio` and add the two instruments with their associated pricers to the portfolio.

```
f1 = finportfolio([AmericanOpt,FixB],[BJSPricer,FBPricer])
```

```
f1 =
  finportfolio with properties:

    Instruments: [2x1 fininstrument.FinInstrument]
        Pricers: [2x1 finpricer.FinPricer]
    PricerIndex: [2x1 double]
       Quantity: [2x1 double]
```

Price Portfolio

Use `pricePortfolio` to compute the price and sensitivities for the portfolio and the instruments in the portfolio.

```
[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(f1)
```

```
PortPrice = 119.1665

InstPrice = 2×1

    3.1912
  115.9753


PortSens=1×8 table
    Price      DV01       Delta      Gamma      Lambda     Vega      Theta       Rho
    ______    _______    _______    ________    ______    ______    ________    _____

    119.17    0.04295    0.23188    0.011522    7.2661    65.454    -0.81408    86.71


InstSens=2×8 table
                       Price      DV01       Delta      Gamma      Lambda     Vega      Theta
                       ______    _______    _______    ________    ______    ______    ________

    vanilla_option     3.1912       NaN     0.23188    0.011522    7.2661    65.454    -0.81408
    fixed_bond         115.98    0.04295        NaN         NaN       NaN       NaN         NaN
```

Price Portfolio of Bond and Bond Option Instruments

This example shows the workflow to create and price a portfolio of bond and bond option instruments. You can use `finportfolio` and `pricePortfolio` to price `FixedBond`,

`FixedBondOption`, `OptionEmbeddedFixedBond`, and `FloatBond` instruments using an `IRTree` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018, 1, 1);
 ZeroTimes = calyears(1:4)';
 ZeroRates = [0.035; 0.042147; 0.047345; 0.052707];
 ZeroDates = Settle + ZeroTimes;
 Compounding = 1;
 ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: 1
               Basis: 0
               Dates: [4x1 datetime]
               Rates: [4x1 double]
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Bond and Option Instruments

Use `fininstrument` to create a `FixedBond`, `FixedBondOption`, `OptionEmbeddedFixedBond`, and `FloatBond` instrument objects.

```
CDates = datetime([2020,1,1 ; 2022,1,1]);
CRates = [.0425; .0750];
CouponRate = timetable(CDates,CRates);
Maturity = datetime(2022,1,1);
Period = 1;

% Vanilla FixedBond
VBond = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',0.0425,'Period',Period,'Name'
```

```
VBond =
  FixedBond with properties:

                CouponRate: 0.0425
                    Period: 1
                     Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
                   Maturity: 01-Jan-2022
```

```
                        Name: "vanilla_fixed"


 % Stepped coupon bond
SBond = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',CouponRate,'Period',Period,'Na

SBond =
  FixedBond with properties:

                 CouponRate: [2x1 timetable]
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                        Name: "stepped_coupon_bond"


% FloatBond
Spread = 0;
Reset = 1;
Float = fininstrument("FloatBond",'Maturity',Maturity,'Spread',Spread,'Reset', Reset,...
                      'ProjectionCurve',ZeroCurve,'Name',"floatbond")

Float =
  FloatBond with properties:

                     Spread: 0
             ProjectionCurve: [1x1 ratecurve]
                 ResetOffset: 0
                       Reset: 1
                       Basis: 0
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
          LatestFloatingRate: NaN
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                        Name: "floatbond"


% Call option
Strike = 100;
ExerciseDates = datetime(2020,1,1);
OptionType ='call';
Period = 1;
```

```
CallOption = fininstrument("FixedBondOption",'Strike',Strike,'ExerciseDate',ExerciseDates,...
                  'OptionType',OptionType,'ExerciseStyle',"american",'Bond', VBond,'Name',"fixed
```

```
CallOption =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2020
           Strike: 100
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option"
```

```
% Option for embedded bond (callable bond)
CDates = datetime([2020,1,1 ; 2022,1,1]);
CRates = [.0425; .0750];
CouponRate = timetable(CDates,CRates);
StrikeOE = [100; 100];
ExerciseDatesOE = [datetime(2020,1,1); datetime(2021,1,1)];
CallSchedule =  timetable(ExerciseDatesOE,StrikeOE,'VariableNames',{'Strike Schedule'});
CallableBond = fininstrument("OptionEmbeddedFixedBond", 'Maturity',Maturity,...
                             'CouponRate',CouponRate,'Period', Period, ...
                             'CallSchedule',CallSchedule,'Name',"option_embedded_fixedbond")
```

```
CallableBond =
  OptionEmbeddedFixedBond with properties:

                 CouponRate: [2x1 timetable]
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                  CallDates: [2x1 datetime]
                   PutDates: [0x1 datetime]
                CallSchedule: [2x1 timetable]
                 PutSchedule: [0x0 timetable]
          CallExerciseStyle: "american"
           PutExerciseStyle: [0x0 string]
                       Name: "option_embedded_fixedbond"
```

Create `HullWhite` Model

Use `finmodel` to create a `HullWhite` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;

HWModel = finmodel("hullwhite",'alpha',AlphaCurve,'sigma',VolCurve)
```

```
HWModel =
  HullWhite with properties:

    Alpha: 0.1000
    Sigma: 0.0100
```

Create IRTree Pricer for HullWhite Model

Use `finpricer` to create an `IRTree` pricer object for a `HullWhite` model and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [4x1 datetime]
           Model: [1x1 finmodel.HullWhite]
   DiscountCurve: [1x1 ratecurve]
```

Create finportfolio Object and Add Callable Bond Instrument

Create a `finportfolio` object with the vanilla bond, stepped coupon bond, float bond, and the call option.

```
myportfolio = finportfolio([VBond,SBond,Float,CallOption],HWTreePricer, [1,2,2,1])
```

```
myportfolio =
  finportfolio with properties:

    Instruments: [4x1 fininstrument.FinInstrument]
        Pricers: [1x1 finpricer.irtree.HWBKTree]
    PricerIndex: [4x1 double]
       Quantity: [4x1 double]
```

Use `addInstrument` to add the callable bond instrument to the existing portfolio.

```
myportfolio = addInstrument(myportfolio,CallableBond,HWTreePricer,1)
```

```
myportfolio =
  finportfolio with properties:

    Instruments: [5x1 fininstrument.FinInstrument]
        Pricers: [1x1 finpricer.irtree.HWBKTree]
    PricerIndex: [5x1 double]
       Quantity: [5x1 double]
```

```
myportfolio.PricerIndex
```

```
ans = 5×1

     1
     1
     1
```

```
     1
     1
```

The `PricerIndex` property has a length equal to the length of instrument objects in the `finportfolio` object and stores the index of which pricer is used for each instrument object. In this case, because there is only one pricer, each instrument must use that pricer.

Price Portfolio

Use `pricePortfolio` to compute the price and sensitivities for the portfolio and the bond and option instruments in the portfolio.

```
format bank
[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(myportfolio)
```

```
PortPrice =
        600.55


InstPrice = 5×1

         96.59
        204.14
        200.00
          0.05
         99.77


PortSens=1×4 table
    Price      Delta       Gamma      Vega
    _______    ________    _______    _______

    600.55    -1297.48    5759.65    -63.40


InstSens=5×4 table
                                Price      Delta      Gamma      Vega
                                _____    ________    _______    _____

    vanilla_fixed                96.59    -344.81    1603.49     -0.00
    stepped_coupon_bond         204.14    -725.96    3364.60      0.00
    floatbond                   200.00       0.00      -0.00     -0.00
    fixed_bond_option             0.05      -3.69      24.15     12.48
    option_embedded_fixedbond    99.77    -223.03     767.41    -75.88
```

Price Portfolio of Multiple Instances of Heterogeneous Instruments

Use `finportfolio` and `pricePortfolio` to create and price a portfolio containing three `FixedBond` instruments and three American `Vanilla` option instruments.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object for three Fixed Bond instruments.

```
FixB = fininstrument("FixedBond",'Maturity',datetime([2022,9,15 ; 2022,10,15 ; 2022,11,15]),'Coup
```

```
FixB=3×1 object
  3x1 FixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    Name
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object for `FixedBond` Instruments

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
FBPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
FBPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Create Vanilla Instrument Object

Use `fininstrument` to create an American `Vanilla` instrument object for three Vanilla instruments.

```
Maturity = datetime([2023,9,15 ; 2023,10,15 ; 2023,11,15]);
AmericanOpt = fininstrument("Vanilla",'ExerciseDate',Maturity,'Strike',120,'ExerciseStyle',"amer
```

```
AmericanOpt=3×1 object
  3x1 Vanilla array with properties:

    OptionType
    ExerciseStyle
    ExerciseDate
    Strike
    Name
```

Create BlackScholes Model Object for Vanilla Instruments

Use `finmodel` to create a `BlackScholes` model object.

```
BSModel = finmodel("BlackScholes",'Volatility',0.12)
```

```
BSModel =
  BlackScholes with properties:

     Volatility: 0.1200
    Correlation: 1
```

Create BjerksundStensland Pricer Object for Vanilla Instruments

Use `finpricer` to create an analytic pricer object for the `BjerksundStensland` pricing method and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
BJSPricer = finpricer("analytic",'Model',BSModel,'DiscountCurve',myRC,'SpotPrice',100,'DividendVa
```

```
BJSPricer =
  BjerksundStensland with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0200
     DividendType: "continuous"
```

Add the Instruments to a finportfolio Object

Create a `finportfolio` object using `finportfolio` and add the six instruments with their associated pricers to the portfolio.

```
f1 = finportfolio([AmericanOpt;FixB],[BJSPricer, BJSPricer, BJSPricer, FBPricer, FBPricer, FBPric
```

```
f1 =
  finportfolio with properties:

    Instruments: [6x1 fininstrument.FinInstrument]
```

```
        Pricers: [6x1 finpricer.FinPricer]
    PricerIndex: [6x1 double]
       Quantity: [6x1 double]
```

Price Portfolio

Use `pricePortfolio` to compute the price and sensitivities for the portfolio and the instruments in the portfolio.

```
[PortPrice,InstPrice,PortSens,InstSens] = pricePortfolio(f1)
```

```
PortPrice = 358.4108

InstPrice = 6×1

    3.1912
    3.2579
    3.3272
  115.9753
  116.2114
  116.4478


PortSens=1×8 table
    Price      DV01       Delta      Gamma      Lambda     Vega      Theta      Rho
    _____    ________    _______    ________    ______    ______    _______    _____

    358.41    0.13159    0.70286    0.034471    21.572    198.96    -2.4455    266.62


InstSens=6×8 table
                        Price       DV01       Delta      Gamma      Lambda     Vega      Theta
                        _____    _________    _______    ________    ______    ______    _______

    vanilla_option      3.1912         NaN    0.23188    0.011522    7.2661    65.454    -0.81408
    vanilla_option_1    3.2579         NaN    0.23427    0.011494    7.1907    66.314    -0.81353
    vanilla_option_2    3.3272         NaN    0.23672    0.011455    7.1147    67.196    -0.81784
    fixed_bond          115.98     0.04295        NaN         NaN       NaN       NaN         NaN
    fixed_bond_1        116.21    0.043858        NaN         NaN       NaN       NaN         NaN
    fixed_bond_2        116.45    0.044786        NaN         NaN       NaN       NaN         NaN
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `finpricer`

Topics

“Create and Price Portfolio of Instruments” on page 3-131
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22

“Choose Instruments, Models, and Pricers” on page 1-53

ratecurve

Create `ratecurve` object for interest-rate curve from dates and data

Description

Build a `ratecurve` object using `ratecurve`.

After creating a `ratecurve` object, you can use the associated object functions `forwardrates`, `discountfactors`, and `zerorates`.

Note If you have the `RateSpec` obtained previously from `intenvset` or `toRateSpec` for an `IRDataCurve` or `toRateSpec` for an `IRFunctionCurve`, refer to "Convert RateSpec to a ratecurve Object" on page 1-49.

To price a `Swap`, `FixedBond`, `FloatBond`, `FRA`, or `Deposit` instrument, you must create a `ratecurve` object and then create `Discount` pricer object.

For more detailed information on this workflow, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" on page 1-22.

For more information on the available instruments, models, and pricing methods, see "Choose Instruments, Models, and Pricers" on page 1-53.

Creation

Syntax

```
ratecurve_obj = ratecurve(Type,Settle,Dates,Rates)
ratecurve_obj = ratecurve( ___ ,Name,Value)
```

Description

`ratecurve_obj = ratecurve(Type,Settle,Dates,Rates)` creates a `ratecurve` object.

`ratecurve_obj = ratecurve( ___ ,Name,Value)` creates a `ratecurve` object using name-value pairs and any of the arguments in the previous syntax. For example, `myRC = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Compounding',2,'Basis',5,'InterpMethod',"pchip",'ShortExtrapMethod',"linear",'LongExtrapMethod',"cubic")` creates a `ratecurve` object for a zero curve. You can specify multiple name-value pair arguments.

Input Arguments

Type — Type of interest-rate curve
string with value `"zero"`, `"forward"`, or `"discount"` | character vector with value `'zero'`, `'forward'`, or `'discount'`

Type of interest-rate curve, specified as a string or character vector for one of the supported types.

Data Types: `char` | `string`

`Settle` — Settlement date
datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `ratecurve` also accepts serial date numbers as inputs, but they are not recommended.

If you use a date character vector or string, the format must be recognizable by `datetime` because the `Settle` property is stored as a datetime.

`Dates` — Dates corresponding to rate data
datetime array | string array | date character vector

Dates corresponding to the rate data, specified as vector using a datetime array, string array, or date character vectors.

To support existing code, `ratecurve` also accepts serial date numbers as inputs, but they are not recommended.

If you use a date character vectors or strings, the format must be recognizable by `datetime` because the `Dates` property is stored as a datetime.

`Rates` — Interest-rate data for the curve
numeric

Interest-rate data for the curve, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `myRC = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Compounding',2,'Basis',5,'InterpMethod',"pchip",'ShortExtrapMethod',"linear",'LongExtrapMethod',"cubic")`

`Compounding` — Compounding frequency
`-1` (default) | possible values include: –1, `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Compounding frequency, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using the supported values: –1, `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`InterpMethod` — Interpolation method
`"linear"` (default) | string with value `"linear"`, `"cubic"`, `"next"`, `"previous"`, `"pchip"`, `"v5cubic"`, `"makima"`, or `"spline"` | character vector with value `'linear'`, `'cubic'`, `'next'`, `'previous'`, `'pchip'`, `'v5cubic'`, `'makima'`, or `'spline'`

Interpolation method, specified as the comma-separated pair consisting of `'InterpMethod'` and a scalar string or character vector using a supported value. For more information on interpolation methods, see `interp1`.

Data Types: `char` | `string`

`ShortExtrapMethod` — Extrapolation method for data before first data
`"next"` (default) | string with value `"linear"`, `"next"`, `"previous"`, `"pchip"`, `"cubic"`,`"v5cubic"`, `"makima"`, or `"spline"` | character vector with value `'linear'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`,`'v5cubic'`, `'makima'`, or `'spline'`

Extrapolation method for data before first data, specified as the comma-separated pair consisting of `'ShortExtrapMethod'` and a scalar string or character vector using a supported value. For more information on interpolation methods, see `interp1`.

Data Types: `char` | `string`

`LongExtrapMethod` — Extrapolation method for data after last data
`"previous"` (default) | string with value `"linear"`, `"next"`, `"previous"`, `"pchip"`, `"cubic"`,`"v5cubic"`, `"makima"`, or `"spline"` | character vector with value `'linear'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`,`'v5cubic'`, `'makima'`, or `'spline'`

Extrapolation method for data after last data, specified as the comma-separated pair consisting of `'LongExtrapMethod'` and a scalar string or character vector using a supported value. For more information on interpolation methods, see `interp1`.

Data Types: `char` | `string`

Properties

`Type` — Type of interest-rate curve
string with value `"zero"`, `"forward"`, or `"discount"`

Type of interest-rate curve, returned as a string.

Data Types: `string`

`Compounding` — Compounding frequency
`-1` (default) | possible values include: –1, `0`, 1, 2, 3, 4, 6, or 12.

Compounding frequency, returned as a scalar numeric.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of the instrument, returned as a scalar integer.

Data Types: `double`

`Dates` — Dates corresponding to rate data
datetime

Dates corresponding to the rate data, returned as a datetime.

Data Types: `datetime`

`Rates` — Rates corresponding to dates data
vector

Rates corresponding to dates data, returned as vector.

Data Types: `datetime`

`Settle` — Settlement date
datetime

Settlement date, returned as a datetime.

Data Types: `datetime`

`InterpMethod` — Interpolation method
`"linear"` (default) | string with value `"linear"`, `"cubic"`, `"next"`, `"previous"`, `"pchip"`, `"v5cubic"`, `"makima"`, or `"spline"`

Interpolation method, returned as a scalar string.

Data Types: `string`

`ShortExtrapMethod` — Short extrapolation method
`"next"` (default) | string with value `"linear"`, `"next"`, `"previous"`, `"pchip"`, `"cubic"`,`"v5cubic"`, `"makima"`, or `"spline"`

Short extrapolation method, returned as a scalar string.

Data Types: `string`

LongExtrapMethod — Long extrapolation method
`"previous"` (default) | string with value `"linear"`, `"next"`, `"previous"`, `"pchip"`, `"cubic"`,`"v5cubic"`, `"makima"`, or `"spline"`

Log extrapolation method, returned as a scalar string.

Data Types: `string`

Object Functions

forwardrates	Calculate forward rates for ratecurve object
discountfactors	Calculate discount factors for a ratecurve object
zerorates	Calculate zero rates for ratecurve object
irbootstrap	Bootstrap interest-rate curve from market data

Examples

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Compounding',2,'Basis',5,'InterpMethod',"pchi
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: 2
               Basis: 5
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "pchip"
   ShortExtrapMethod: "linear"
    LongExtrapMethod: "cubic"
```

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `ratecurve` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`parametercurve`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Convert RateSpec to a ratecurve Object” on page 1-49

inflationcurve

Create `inflationcurve` object for interest-rate curve from dates and data

Description

Build an `inflationcurve` object using `inflationcurve`.

After creating a `inflationcurve` object, you can use the associated object function `indexvalues`.

To price an `InflationBond`, `YearYearInflationSwap`, or `ZeroCouponInflationSwap` instrument, you must create an `inflationcurve` object and then create an `Inflation` pricer object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
inflationcurve_obj = inflationcurve(Dates,InflationIndexValues)
inflationcurve_obj = inflationcurve( ___ ,Name,Value)
```

Description

`inflationcurve_obj = inflationcurve(Dates,InflationIndexValues)` creates an `inflationcurve` object.

`inflationcurve_obj = inflationcurve( ___ ,Name,Value)` creates an `inflationcurve` object using name-value pairs and any of the arguments in the previous syntax. For example, `myInflationCurve = inflationcurve(InflationDates,InflationIndexValues,'Basis',4)` creates an `inflationcurve` object. You can specify multiple name-value pair arguments.

Input Arguments

`Dates` — Dates corresponding to `InflationIndexValues`
datetime array | serial date number | cell array of date character vectors | date string array

Dates corresponding to `InflationIndexValues`, specified as a datetime array, serial date numbers, cell array of date character vectors, or date string array. The first date is the base date.

If you use a date character vector or date string, the format must be recognizable by `datetime` because the `Dates` property is stored as a datetime.

Data Types: `double` | `char` | `cell` | `string` | `datetime`

`InflationIndexValues` — Inflation index values for the curve

vector of positive values

Inflation index values for the curve, specified as a vector of positive values. The first value is the base index value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `myInflationCurve = inflationcurve(InflationDates,InflationIndexValues,'Basis',4)`

`Basis` — Day count basis

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Seasonality` — Seasonal adjustment rates

12-by-1 vector of `0`s (no seasonality) (default) | decimal

Seasonal adjustment rates, specified as the comma-separated pair consisting of `'Seasonality'` and a 12-by-1 vector in decimals for each month ordered from January to December. The rates are annualized and continuously compounded seasonal rates that are internally corrected to add to `0`.

Data Types: `double`

Properties

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis of the instrument, returned as a scalar integer.

Data Types: `double`

`Dates` — Dates corresponding to `InflationIndexValues`
datetime

Dates corresponding to `InflationIndexValues`, returned as a datetime array.

Data Types: `datetime`

`InflationIndexValues` — Inflation index values for the curve
vector

Inflation index values for the curve, returned as vector.

Data Types: `double`

`ForwardInflationRates` — Forward inflation rates
vector

Forward inflation rates, returned as vector.

Data Types: `double`

`Seasonality` — Seasonal adjustment rates
vector

Seasonal adjustment rates, returned as a 12-by-1 vector.

Data Types: `double`

Object Functions

indexvalues Calculate index values for inflationcurve object

Examples

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2020, 9, 20);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;

myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:
```

```
                Basis: 0
                Dates: [10x1 datetime]
 InflationIndexValues: [10x1 double]
ForwardInflationRates: [9x1 double]
          Seasonality: [12x1 double]
```

Algorithms

Build an inflation curve from a series of breakeven zero-coupon inflation swap (ZCIS) rates:

$I(0, T_{1Y}) = I(T_0)($1

$I(0, T_{2Y}) = I(T_0)($1

$I(0, T_{3Y}) = I(T_0)($1

...

$I(0, T_i) = I(T_0)($1

where

- $I(0, T_i)$ is the breakeven inflation index reference number for maturity date T_i.
- $I(T_0)$ is the base inflation index value for the starting date T_0.
- $b(0; T_0, T_i)$ is the breakeven inflation rate for the ZCIS maturing on T_i.

The ZCIS rates typically have maturities that increase in whole number of years. So the inflation curve is built on an annual basis. From the annual basis inflation curve, the annual unadjusted (that is, not seasonally adjusted) forward inflation rates are computed as follows:

$$f_i = \frac{1}{(T_i - T_{i-1})} \log\left(\frac{I(0, T_i)}{I(0, T_{i-1})}\right)$$

The unadjusted forward inflation rates are used for interpolating and also for incorporating seasonality to the inflation curve.

For monthly periods that are not a whole number of years, seasonal adjustments can be made to reflect seasonal patterns of inflation within the year. These 12 monthly seasonal adjustments are annualized and they add up to zero to ensure that the cumulative seasonal adjustments are reset to zero every year.

$$I(0, T_i) = I(T_0)\exp\left(\int_{T_0}^{T_i} f(u)du\right)\exp\left(\int_{T_0}^{T_i} s(u)du\right)$$

$$I(0, T_i) = I(0, T_{i-1})\exp((T_i - T_{i-1})(f_i + s_i))$$

where

- $I(0, T_i)$ is the breakeven inflation index reference number.
- $I(0, T_{i-1})$ is the previous inflation reference number.

- f_i is the annual unadjusted forward inflation rate.
- s_i is the annualized seasonal component for the period $[T_{i-1}, T_i]$.

The first year seasonal adjustment may need special treatment, because typically, the breakeven inflation reference number of the first month is already known. If that is the case, the unadjusted forward inflation rate for the first year needs to be recomputed for the remaining 11 months.

Version History

Introduced in R2021a

References

[1] Brody, D. C., Crosby, J., and Li, H. "Convexity Adjustments in Inflation-Linked Derivatives." *Risk Magazine*. November 2008, pp. 124–129.

[2] Kerkhof, J. "Inflation Derivatives Explained: Markets, Products, and Pricing." *Fixed Income Quantitative Research*, Lehman Brothers, July 2005.

[3] Zhang, J. X. "Limited Price Indexation (LPI) Swap Valuation Ideas." *Wilmott Magazine*. no. 57, January 2012, pp. 58–69.

See Also

Functions

`inflationbuild`

Topics

“Analyze Inflation-Indexed Instruments” on page 2-132
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Convert RateSpec to a ratecurve Object” on page 1-49

indexvalues

Calculate index values for `inflationcurve` object

Syntax

```
outIndexValues = indexvalues(inpInflationCurve,inpDates)
```

Description

`outIndexValues = indexvalues(inpInflationCurve,inpDates)` calculates the inflation index values for the `inflationcurve` object (`inpInflationCurve`) based on the Inflation index end dates (`inpDates`).

Examples

Calculate Index Values for `inflationcurve` Object

Create an inflation`curve` object using `inflationcurve`.

```
BaseDate = datetime(2020,9,20);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues);
```

Compute the index values using `indexvalues`.

```
outIndexValues = indexvalues(myInflationCurve,datetime(2023,9,20))
```

```
outIndexValues = 105
```

Input Arguments

`inpInflationCurve` — Inflation curve

`inflationcurve` object

Inflation curve, specified using a previously created `inflationcurve` object.

Data Types: `object`

`inpDates` — Inflation index end dates

datetime array | string array | date character vector

Inflation index end dates, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `indexvalues` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`outIndexValues` — Output inflation index values
numeric

Output inflation index values, returned as a numeric.

Version History

Introduced in R2021a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `indexvalues` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`inflationbuild`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

inflationbuild

Build inflation curve from market zero-coupon inflation swap rates

Syntax

```
InflationCurve = inflationbuild(BaseDate,BaseIndexValue,ZCISDates,ZCISRates)
myInflationCurve = inflationbuild( ___ ,Name,Value)
```

Description

`InflationCurve = inflationbuild(BaseDate,BaseIndexValue,ZCISDates,ZCISRates)` builds an inflation curve from market zero-coupon inflation swap (ZCIS) rates. The `InflationCurve` output is an `inflationcurve` object.

`myInflationCurve = inflationbuild( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any of the input argument combinations in the previous syntax. For example, `myInflationCurve = inflationbuild(BaseDate,BaseIndexValue,ZCISDates,ZCISRates,'Seasonality',SeasonalRates)` builds an `inflationcurve` object from market zero ZCIS dates and rates.

Examples

Build Inflation Curve from Zero-Coupon Inflation Swap Rates

This example shows the workflow to build an `inflationcurve` object from zero-coupon inflation swap (ZCIS) rates using `inflationbuild`.

Define the inflation curve parameters.

```
BaseDate = datetime(2020,9,20);
BaseIndexValue = 100;
ZCISTimes = [calyears([1 2 3 4 5 7 10 20 30])]';
ZCISRates = [0.51 0.65 0.87 0.92 0.95 1.42 1.75 2.03 2.54]'./100;
ZCISDates = BaseDate + ZCISTimes;
SeasonalRates = [-0.19 -0.09 -0.04 0.1 0.16 0.11 0.26 0.17 -0.07 -0.08 -0.14 -0.19]'./100;
```

Use `inflationbuild` to create an `inflationcurve` object.

```
myInflationCurve = inflationbuild(BaseDate,BaseIndexValue,ZCISDates,ZCISRates,'Seasonality',Seaso
```

```
myInflationCurve =
  inflationcurve with properties:

                    Basis: 0
                    Dates: [10x1 datetime]
     InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Input Arguments

`BaseDate` — Base date of inflation curve

datetime scalar | string scalar | date character vector

Base date of inflation curve, specified as a scalar datetime, string, or data character vector.

To support existing code, `inflationbuild` also accepts serial date numbers as inputs, but they are not recommended.

`BaseIndexValue` — Base index value of inflation curve

numeric

Base index value of inflation curve, specified as a scalar numeric.

Data Types: `double`

`ZCISDates` — Market ZCIS maturity dates minus lag

datetime array | string array | date character vector

Market ZCIS maturity dates minus lag, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `inflationbuild` also accepts serial date numbers as inputs, but they are not recommended.

`ZCISRates` — Market ZCIS rates

decimal

Market ZCIS rates, specified as an `NINST`-by-`1` vector of decimals.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `myInflationCurve = inflationbuild(BaseDate,BaseIndexValue,ZCISDates,ZCISRates,'Seasonality',SeasonalRates)`

`Basis` — Day count basis

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365

- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)
- 11 — 30/360E (ICMA)
- 12 — actual/365 (ISDA)
- 13 — BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Seasonality` — Seasonal adjustment rates
`12`-by-`1` vector of `0`s (no seasonality) (default) | decimal

Seasonal adjustment rates, specified as the comma-separated pair consisting of `'Seasonality'` and a `12`-by-`1` vector in decimals for each month ordered from January to December. The rates are annualized and continuously compounded seasonal rates that are internally corrected to add to `0`.

Data Types: `double`

`FirstMonthIndex` — First month inflation index
`[]` (not known) (default) | positive numeric

First month inflation index, specified as the comma-separated pair consisting of `'FirstMonthIndex'` and a positive numeric.

Data Types: `double`

Output Arguments

`InflationCurve` — Inflation curve
`inflationcurve` object

Inflation curve, returned as an `inflationcurve` object. The object has the following properties:

- `Basis`
- `Dates`
- `InflationIndexValues`
- `ForwardInflationRates`
- `Seasonality`

Algorithms

Build an inflation curve from a series of breakeven zero-coupon inflation swap (ZCIS) rates:

$$I(0,T_{1Y}) = I(T_0)(^1$$

$$I(0,T_{2Y}) = I(T_0)(^1$$

$$I(0,T_{3Y}) = I(T_0)(^1$$

...

$$I(0,T_i) = I(T_0)(^1$$

where

- $I(0,T_i)$ is the breakeven inflation index reference number for maturity date T_i.
- $I(T_0)$ is the base inflation index value for the starting date T_0.
- $b(0;T_0,T_i)$ is the breakeven inflation rate for the ZCIS maturing on T_i.

The ZCIS rates typically have maturities that increase in whole number of years, so the inflation curve is built on an annual basis. From the annual basis inflation curve, the annual unadjusted (that is, not seasonally adjusted) forward inflation rates are computed as follows:

$$f_i = \frac{1}{(T_i - T_{i-1})}\log\left(\frac{I(0,T_i)}{I(0,T_{i-1})}\right)$$

The unadjusted forward inflation rates are used for interpolating and also for incorporating seasonality to the inflation curve.

For monthly periods that are not a whole number of years, seasonal adjustments can be made to reflect seasonal patterns of inflation within the year. These 12 monthly seasonal adjustments are annualized and they add up to zero to ensure that the cumulative seasonal adjustments are reset to zero every year.

$$I(0,T_i) = I(T_0)\exp\left(\int_{T_0}^{T_i} f(u)du\right)\exp\left(\int_{T_0}^{T_i} s(u)du\right)$$

$$I(0,T_i) = I(0,T_{i-1})\exp((T_i - T_{i-1})(f_i + s_i))$$

where

- $I(0,T_i)$ is the breakeven inflation index reference number.
- $I(0,T_{i-1})$ is the previous inflation reference number.
- f_i is the annual unadjusted forward inflation rate.
- s_i is the annualized seasonal component for the period $[T_{i-1},T_i]$.

The first year seasonal adjustment may need special treatment because, typically, the breakeven inflation reference number of the first month is already known. If that is the case, the unadjusted forward inflation rate for the first year needs to be recomputed for the remaining 11 months.

Version History

Introduced in R2021a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `inflationbuild` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`indexvalues`

Topics

“Analyze Inflation-Indexed Instruments” on page 2-132
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

price

Package: finpricer

Compute price for inflation instrument with Inflation pricer

Syntax

```
[Price,PriceResult] = price(inpPricer,inpInstrument)
[Price,PriceResult] = price( ___ ,inpSensitivity)
```

Description

[Price,PriceResult] = price(inpPricer,inpInstrument) computes the instrument price and related pricing information based on the pricing object inpPricer and the instrument object inpInstrument.

[Price,PriceResult] = price(___ ,inpSensitivity) adds an optional argument to specify sensitivities.

Examples

Use Inflation Pricer and inflationcurve to Price Inflation Bond Instrument

This example shows the workflow to price an InflationBond instrument when you use an inflationcurve object and an Inflation pricing method.

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2021
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2020,10,1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
     InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create `InflationBond` Instrument Object

Use `fininstrument` to create an `InflationBond` instrument object.

```
IssueDate = datetime(2021,1,1);
Maturity = datetime(2026,1,1);
CouponRate = 0.02;

InflationBond = fininstrument("InflationBond",'IssueDate',IssueDate,'Maturity',Maturity,'CouponRa
```

```
InflationBond =
  InflationBond with properties:

                  CouponRate: 0.0200
                      Period: 2
                       Basis: 0
                   Principal: 100
    DaycountAdjustedCashFlow: 0
                         Lag: 3
       BusinessDayConvention: "actual"
                    Holidays: NaT
                EndMonthRule: 1
                   IssueDate: 01-Jan-2021
             FirstCouponDate: NaT
              LastCouponDate: NaT
                    Maturity: 01-Jan-2026
                        Name: "inflation_bond_instrument"
```

Create `Inflation` Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:
```

```
     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price `InflationBond` Instrument

Use `price` to compute the price and sensitivities for the `InflationBond` instrument.

```
[Price,outPR] = price(outPricer,InflationBond)
```

```
Price = 112.1856

outPR =
  priceresult with properties:

       Results: [1x1 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=table
    Price
    ______

    112.19
```

Input Arguments

`inpPricer` — Pricer object
`Inflation` pricer object

Pricer object, specified as a scalar `Inflation` pricer object. Use `finpricer` to create the `Inflation` pricer object.

Data Types: `object`

`inpInstrument` — Instrument object
`ZeroCouponInflationSwap` object | `YearYearInflationSwap` object | `InflationBond` object

Instrument object, specified as a scalar or vector of `InflationBond`, `YearYearInflationSwap`, or `ZeroCouponInflationSwap` instrument objects. Use `fininstrument` to create the `InflationBond`, `YearYearInflationSwap`, or `ZeroCouponInflationSwap` instrument objects.

Data Types: `object`

`inpSensitivity` — List of sensitivities to compute
`[ ]` (default) | string array with values `"Price"` and `"All"` | cell array of character vectors with values `'Price'` and `'All'`

(Optional) List of sensitivities to compute, specified as an `NOUT`-by-`1` or `1`-by-`NOUT` cell array of character vectors or string array with possible values of `'Price'` and `'All'`.

`inpSensitivity = {'All'}` or `inpSensitivity = ["All"]` specifies that the output is `'Price'`. This option is the same as specifying `inpSensitivity` to include each sensitivity.

Example: `inpSensitivity = {'price'}`

Data Types: `string` | `cell`

Output Arguments

`Price` — Instrument price

numeric

Instrument price, returned as a numeric.

`PriceResult` — Price result

`PriceResult` object

Price result, returned as an object. The object has the following fields:

- `PriceResult.Results` — Table of results
- `PriceResult.PricerData` — Structure for pricer data

Version History

Introduced in R2021a

See Also

`fininstrument` | `finpricer`

Topics

“Analyze Inflation-Indexed Instruments” on page 2-132
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

inflationCashflows

Package: fininstrument

Compute cash flows for InflationBond instrument

Syntax

```
outCF = inflationCashflows(inpInstrumentObject,Settle,inpInflationCurve)
```

Description

outCF = inflationCashflows(inpInstrumentObject,Settle,inpInflationCurve) computes cash flows for an InflationBond instrument object.

Examples

Price Inflation Bond Instrument Using inflationcurve and Inflation Pricer and Compute Cash Flows

This example shows the workflow to price an InflationBond instrument when you use an inflationcurve object and an Inflation pricing method. The cash flows for the InflationBond instrument are computed using inflationCashflows.

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2021
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create inflationcurve Object

Create an inflationcurve object using inflationcurve.

```
BaseDate = datetime(2020,10,1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
     InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create `InflationBond` Instrument Object

Use `fininstrument` to create an `InflationBond` instrument object.

```
IssueDate = datetime(2021,1,1);
Maturity = datetime(2026,1,1);
CouponRate = 0.02;

InflationBond = fininstrument("InflationBond",'IssueDate',IssueDate,'Maturity',Maturity,'CouponRa
```

```
InflationBond =
  InflationBond with properties:

                 CouponRate: 0.0200
                     Period: 2
                      Basis: 0
                  Principal: 100
    DaycountAdjustedCashFlow: 0
                        Lag: 3
       BusinessDayConvention: "actual"
                    Holidays: NaT
                EndMonthRule: 1
                   IssueDate: 01-Jan-2021
             FirstCouponDate: NaT
              LastCouponDate: NaT
                    Maturity: 01-Jan-2026
                        Name: "inflation_bond_instrument"
```

Create `Inflation` Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price InflationBond Instrument

Use price to compute the price and sensitivities for the InflationBond instrument.

```
[Price,outPR] = price(outPricer,InflationBond)
```

```
Price = 112.1856

outPR =
  priceresult with properties:

       Results: [1x1 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=table
    Price
    ______

    112.19
```

Compute Cash Flows for InflationBond Instrument

Use inflationCashflows to compute the cash flows for the InflationBond instrument.

```
outCF = inflationCashflows(InflationBond,datetime(2021,1,15),myInflationCurve)
```

```
outCF=11×1 timetable
       Time        InflationCFAmounts
    ___________    __________________

    15-Jan-2021        -0.077407
    01-Jul-2021           1.0099
    01-Jan-2022             1.02
    01-Jul-2022           1.0275
    01-Jan-2023            1.035
    01-Jul-2023           1.0425
    01-Jan-2024             1.05
    01-Jul-2024            1.059
    01-Jan-2025            1.068
    01-Jul-2025            1.075
    01-Jan-2026           109.28
```

Price Multiple Inflation Bond Instruments Using inflationcurve and Inflation Pricer and Compute Cash Flows

This example shows the workflow to price multiple InflationBond instruments when you use an inflationcurve object and an Inflation pricing method. The cash flows for the InflationBond instruments are computed using inflationCashflows.

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2021
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2019,8,1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
    InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create `InflationBond` Instrument Object

Use `fininstrument` to create an `InflationBond` instrument object for three Inflation Bond instruments.

```
IssueDate = datetime([2020,1,1 ; 2019,12,1 ; 2019,11,1]);
Maturity = datetime([2026,1,1 ; 2026,2,1 ; 2026,3,1]);
CouponRate = 0.02;

InflationBond = fininstrument("InflationBond",'IssueDate',IssueDate,'Maturity',Maturity,'CouponRa
```

```
InflationBond=3×1 object
  3x1 InflationBond array with properties:

    CouponRate
    Period
    Basis
    Principal
```

```
    DaycountAdjustedCashFlow
    Lag
    BusinessDayConvention
    Holidays
    EndMonthRule
    IssueDate
    FirstCouponDate
    LastCouponDate
    Maturity
    Name
```

Create `Inflation` Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price `InflationBond` Instruments

Use `price` to compute the prices and sensitivities for the `InflationBond` instruments.

```
[Price,outPR] = price(outPricer,InflationBond)
```

```
Price = 3×1

  113.6829
  113.9533
  114.2316


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=table
    Price
    ______

    113.68


ans=table
    Price
    ______
```

```
    113.95


ans=table
    Price
    _____

    114.23
```

Compute Cash Flows for `InflationBond` Instruments

Use `inflationCashflows` to compute the cash flows for the three `InflationBond` instruments.

```
outCF = inflationCashflows(InflationBond(1),datetime(2021,1,15),myInflationCurve)
```

```
outCF=11×1 timetable
       Time        InflationCFAmounts
    ___________    __________________

    15-Jan-2021        -0.078871
    01-Jul-2021           1.0266
    01-Jan-2022           1.0341
    01-Jul-2022           1.0415
    01-Jan-2023           1.0495
    01-Jul-2023           1.0585
    01-Jan-2024           1.0668
    01-Jul-2024           1.0738
    01-Jan-2025            1.081
    01-Jul-2025           1.0886
    01-Jan-2026           110.73


outCF = inflationCashflows(InflationBond(2),datetime(2021,1,15),myInflationCurve)

outCF=12×1 timetable
       Time        InflationCFAmounts
    ___________    __________________

    15-Jan-2021         -0.92699
    01-Feb-2021            1.022
    01-Aug-2021           1.0295
    01-Feb-2022            1.037
    01-Aug-2022           1.0444
    01-Feb-2023           1.0527
    01-Aug-2023           1.0617
    01-Feb-2024           1.0697
    01-Aug-2024           1.0767
    01-Feb-2025            1.084
    01-Aug-2025           1.0917
    01-Feb-2026           111.05


outCF = inflationCashflows(InflationBond(3),datetime(2021,1,15),myInflationCurve)

outCF=12×1 timetable
       Time        InflationCFAmounts
    ___________    __________________
```

```
    15-Jan-2021         -0.76871
    01-Mar-2021            1.025
    01-Sep-2021           1.0325
    01-Mar-2022             1.04
    01-Sep-2022           1.0475
    01-Mar-2023            1.056
    01-Sep-2023            1.065
    01-Mar-2024           1.0726
    01-Sep-2024           1.0797
    01-Mar-2025           1.0871
    01-Sep-2025           1.0948
    01-Mar-2026           111.36
```

Input Arguments

`inpInstrumentObject` — Instrument object

`InflationBond` object

Instrument object, specified using a previously created instrument object for an `InflationBond`.

Note If the `inpInstrumentObject` is a vector of instruments, you must use `inflationCashflows` separately with each instrument.

Data Types: `object`

`Settle` — Settlement date for instrument cash flow

datetime scalar | string scalar | date character vector

Settlement date for instrument cash flow, specified as a scalar datetime, string, or date character vector.

Note The `Settle` date you specify must be before the `Maturity` date for the `InflationBond` instrument.

To support existing code, `inflationCashflows` also accepts serial date numbers as inputs, but they are not recommended.

`inpInflationCurve` — Inflation curve

`inflationcurve` object

Inflation curve, specified using a previously created inflation curve object using `inflationcurve`.

Data Types: `object`

Output Arguments

`outCF` — Output cash flow

timetable

Output cash flow, returned as a timetable.

Version History

Introduced in R2021a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `inflationCashflows` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`fininstrument` | `finpricer` | `timetable`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

inflationCashflows

Package: fininstrument

Compute cash flows for YearYearInflationSwap instrument

Syntax

```
outCF = inflationCashflows(inpInstrumentObject,Settle,inpInflationCurve)
```

Description

outCF = inflationCashflows(inpInstrumentObject,Settle,inpInflationCurve) computes cash flows for an YearYearInflationSwap instrument object.

Examples

Price Year-on-Year Inflation-Indexed Swap Instrument Using inflationcurve and Inflation Pricer and Compute Cash Flows

This example shows the workflow to price a YearYearInflationSwap instrument when you use an inflationcurve object and an Inflation pricing method. Then use inflationCashflows to compute the cash flows for the YearYearInflationSwap instrument.

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2021
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create inflationcurve Object

Create an inflationcurve object using inflationcurve.

```
BaseDate = datetime(2020,10,1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
     InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create YearYearInflationSwap Instrument Object

Use `fininstrument` to create a `YearYearInflationSwap` instrument object.

```
Maturity = datetime(2025,1,1);
FixedInflationRate = 0.015;
Notional = 2000;

YYInflationSwap = fininstrument("YearYearInflationSwap",'Maturity',Maturity,'FixedInflationRate'
```

```
YYInflationSwap =
  YearYearInflationSwap with properties:

              Notional: 2000
    FixedInflationRate: 0.0150
                 Basis: 0
                   Lag: 3
              Maturity: 01-Jan-2025
                  Name: "YYInflationSwap_instrument"
```

Create Inflation Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price YearYearInflationSwap Instrument

Use `price` to compute the price and sensitivities for the `YearYearInflationSwap` instrument.

```
[Price,outPR] = price(outPricer,YYInflationSwap,"all")
```

```
Price = 12.5035
```

```
outPR =
  priceresult with properties:

       Results: [1x1 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=table
    Price
    ______

    12.504
```

Compute Cash Flows for YearYearInflationSwap Instrument

Use `inflationCashflows` to compute the cash flows for the `YearYearInflationSwap` instrument.

```
outCF = inflationCashflows(YYInflationSwap,datetime(2021,1,15),myInflationCurve)
```

```
outCF=4×2 timetable
       Time        Var1     Var2
    ___________    ____    ______

    01-Jan-2022    -30         40
    01-Jan-2023    -30     29.412
    01-Jan-2024    -30     28.986
    01-Jan-2025    -30     34.286
```

Price Multiple Year-on-Year Inflation-Indexed Swap Instruments Using `inflationcurve` and Inflation Pricer and Compute Cash Flows

This example shows the workflow to price multiple `YearYearInflationSwap` instrument when you use an `inflationcurve` object and an `Inflation` pricing method. Then use `inflationCashflows` to compute the cash flows for the `YearYearInflationSwap` instruments.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
```

```
                Dates: [10x1 datetime]
                Rates: [10x1 double]
               Settle: 15-Jan-2021
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2019,10,1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                    Basis: 0
                    Dates: [10x1 datetime]
     InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create `YearYearInflationSwap` Instrument Object

Use `fininstrument` to create a `YearYearInflationSwap` instrument object for three Year-on-Year Inflation-Indexed Swap instruments.

```
Maturity = datetime([2024,1,1 ; 2024,11,1 ; 2024,12,1]);
FixedInflationRate = 0.015;
Notional = [20000 ; 30000 ; 40000];

YYInflationSwap = fininstrument("YearYearInflationSwap",'Maturity',Maturity,'FixedInflationRate'
```

```
YYInflationSwap=3×1 object
  3x1 YearYearInflationSwap array with properties:

    Notional
    FixedInflationRate
    Basis
    Lag
    Maturity
    Name
```

Create `Inflation` Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:
```

```
     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price YearYearInflationSwap Instruments

Use `price` to compute the prices and sensitivities for the `YearYearInflationSwap` instruments.

```
[Price,outPR] = price(outPricer,YYInflationSwap,"all")
```

```
Price = 3×1

   26.0701
   18.1540
    1.3201


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=table
    Price
    _____

    26.07


ans=table
    Price
    ______

    18.154


ans=table
    Price
    ______

    1.3201
```

Compute Cash Flows for YearYearInflationSwap Instruments

Use `inflationCashflows` to compute the cash flows for the `YearYearInflationSwap` instruments.

```
outCF = inflationCashflows(YYInflationSwap(1),datetime(2021,1,15),myInflationCurve)
```

```
outCF=3×2 timetable
       Time        Var1     Var2
    ___________    ____    ______
```

```
    01-Jan-2022    -300    294.12
    01-Jan-2023    -300    289.86
    01-Jan-2024    -300    342.86
```

```
outCF = inflationCashflows(YYInflationSwap(2),datetime(2021,1,15),myInflationCurve)
```

```
outCF=4×2 timetable
       Time        Var1     Var2
    ___________    ____    ______

    01-Nov-2021    -450    467.39
    01-Nov-2022    -450    435.85
    01-Nov-2023    -450    500.98
    01-Nov-2024    -450    413.63
```

```
outCF = inflationCashflows(YYInflationSwap(3),datetime(2021,1,15),myInflationCurve)
```

```
outCF=4×2 timetable
       Time        Var1     Var2
    ___________    ____    ______

    01-Dec-2021    -600    605.42
    01-Dec-2022    -600    580.41
    01-Dec-2023    -600    676.99
    01-Dec-2024    -600     537.7
```

Input Arguments

`inpInstrumentObject` — Instrument object

`YearYearInflationSwap` object

Instrument object, specified using a previously created instrument object for a `YearYearInflationSwap`.

Note If the `inpInstrumentObject` is a vector of instruments, you must use `inflationCashflows` separately with each instrument.

Data Types: `object`

`Settle` — Settlement date for instrument cash flow

datetime scalar | string scalar | date character vector

Settlement date for instrument cash flow, specified as a scalar datetime, string, or date character vector.

Note The `Settle` date you specify must be before the `Maturity` date for the `YearYearInflationSwap` instrument.

To support existing code, `inflationCashflows` also accepts serial date numbers as inputs, but they are not recommended.

`inpInflationCurve` — Inflation curve
`inflationcurve` object

Inflation curve, specified using a previously created inflation curve object using `inflationcurve`.

Data Types: `object`

Output Arguments

`outCF` — Output cash flow
timetable

Output cash flow, returned as a timetable.

Version History

Introduced in R2021a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `inflationCashflows` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`fininstrument` | `finpricer` | `timetable`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

inflationCashflows

Package: `fininstrument`

Compute cash flows for `ZeroCouponInflationSwap` instrument

Syntax

```
outCF = inflationCashflows(inpInstrumentObject,Settle,inpInflationCurve)
```

Description

`outCF = inflationCashflows(inpInstrumentObject,Settle,inpInflationCurve)` computes cash flows for a `ZeroCouponInflationSwap` instrument object.

Examples

Price Zero-Coupon Inflation Swap Instrument Using `inflationcurve` and Inflation Pricer and Compute Cash Flow

This example shows the workflow to price a `ZeroCouponInflationSwap` instrument when you use an `inflationcurve` object and an `Inflation` pricing method. Then use `inflationCashflows` to compute the cash flow for the `ZeroCouponInflationSwap` instrument.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2021
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2020, 10, 1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
     InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create ZeroCouponInflationSwap Instrument Object

Use `fininstrument` to create a `ZeroCouponInflationSwap` instrument object.

```
StartDate = datetime(2021,1,1);
Maturity = datetime(2022,10,1);
FixedInflationRate = 0.015;
Notional = 2000;

ZCInflationSwap = fininstrument("ZeroCouponInflationSwap",'StartDate',StartDate,'Maturity',Matur
```

```
ZCInflationSwap =
  ZeroCouponInflationSwap with properties:

             Notional: 2000
    FixedInflationRate: 0.0150
                Basis: 0
                  Lag: 3
            StartDate: 01-Jan-2021
             Maturity: 01-Oct-2022
                 Name: "zero_coupon_inflation_swap_instrument"
```

Create Inflation Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price ZeroCouponInflationSwap Instrument

Use `price` to compute the price and sensitivities for the `ZeroCouponInflationSwap` instrument.

```
[Price,outPR] = price(outPricer,ZCInflationSwap,"all")
```

```
Price = 9.5675

outPR =
  priceresult with properties:

       Results: [1x1 table]
    PricerData: []
```

```
outPR.Results

ans=table
    Price
    ______

    9.5675
```

Compute Cash Flow for ZeroCouponInflationSwap Instrument

Use `inflationCashflows` to compute the cash flow for the `ZeroCouponInflationSwap` instrument.

```
outCF = inflationCashflows(ZCInflationSwap,datetime(2021,1,1),myInflationCurve)

outCF=1×2 timetable
       Time         Var1       Var2
    ____________    _______    ______

    01-Oct-2022    -52.732    62.397
```

Price Multiple Zero-Coupon Inflation Swap Instruments Using `inflationcurve` and Inflation Pricer and Compute Cash Flow

This example shows the workflow to price multiple `ZeroCouponInflationSwap` instruments when you use an `inflationcurve` object and an `Inflation` pricing method. Then use `inflationCashflows` to compute the cash flow for the `ZeroCouponInflationSwap` instruments.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2021,12,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)

ZeroCurve =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 0
              Dates: [10x1 datetime]
```

```
                Rates: [10x1 double]
               Settle: 15-Dec-2021
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2020, 10, 1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
    InflationIndexValues: [10x1 double]
   ForwardInflationRates: [9x1 double]
             Seasonality: [12x1 double]
```

Create `ZeroCouponInflationSwap` Instrument Object

Use `fininstrument` to create a `ZeroCouponInflationSwap` instrument object for three Zero-Coupon Inflation Swap instruments.

```
StartDate = datetime([2021,5,1 ; 2021,6,1 ; 2021,7,1]);
Maturity = datetime([2022,10,1 ; 2022,11,1 ;2022,12,1]);
FixedInflationRate = 0.015;
Notional = [20000 ; 30000 ; 40000];

ZCInflationSwap = fininstrument("ZeroCouponInflationSwap",'StartDate',StartDate,'Maturity',Matur
```

```
ZCInflationSwap=3×1 object
  3x1 ZeroCouponInflationSwap array with properties:

    Notional
    FixedInflationRate
    Basis
    Lag
    StartDate
    Maturity
    Name
```

Create `Inflation` Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price ZeroCouponInflationSwap Instruments

Use `price` to compute the prices and sensitivities for the `ZeroCouponInflationSwap` instruments.

```
[Price,outPR] = price(outPricer,ZCInflationSwap,"all")
```

```
Price = 3×1

   59.4576
   80.6037
   89.4137


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=table
    Price
    ______

    59.458


ans=table
    Price
    ______

    80.604


ans=table
    Price
    ______

    89.414
```

Compute Cash Flow for ZeroCouponInflationSwap Instruments

Use `inflationCashflows` to compute the cash flow for the `ZeroCouponInflationSwap` instruments.

```
outCF = inflationCashflows(ZCInflationSwap(1),datetime(2022,1,1),myInflationCurve)
```

```
outCF=1×2 timetable
       Time         Var1      Var2
```

```
    ____________    ________    _____

    01-Oct-2022    -427.09    486.8
```

```
outCF = inflationCashflows(ZCInflationSwap(2),datetime(2022,1,1),myInflationCurve)
```

```
outCF=1×2 timetable
       Time          Var1       Var2
    ____________    ________    _______

    01-Nov-2022    -640.63    721.62
```

```
outCF = inflationCashflows(ZCInflationSwap(3),datetime(2022,1,1),myInflationCurve)
```

```
outCF=1×2 timetable
       Time          Var1       Var2
    ____________    ________    _______

    01-Dec-2022    -854.18    944.06
```

Input Arguments

`inpInstrumentObject` — Instrument object

`ZeroCouponInflationSwap` object

Instrument object, specified using a previously created instrument object for a `ZeroCouponInflationSwap`.

Note If the `inpInstrumentObject` is a vector of instruments, you must use `inflationCashflows` separately with each instrument.

Data Types: `object`

`Settle` — Settlement date for instrument cash flow

datetime scalar | string scalar | date character vector

Settlement date for instrument cash flow, specified as a scalar datetime, string, or date character vector.

Note The `Settle` date you specify must be before the `Maturity` date for the `ZeroCouponInflationSwap` instrument.

To support existing code, `inflationCashflows` also accepts serial date numbers as inputs, but they are not recommended.

`inpInflationCurve` — Inflation curve

`inflationcurve` object

Inflation curve, specified using a previously created inflation curve object using `inflationcurve`.

Data Types: `object`

Output Arguments

`outCF` — Output cash flow

timetable

Output cash flow, returned as a timetable.

Version History

Introduced in R2021a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `inflationCashflows` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`fininstrument` | `finpricer` | `timetable`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

OISFuture

`OISFuture` instrument object

Description

Create and price an `OISFuture` instrument object for one or more one-month or three-month future instruments using this workflow:

1. Use `fininstrument` to create an `OISFuture` instrument object for one or more OIS Future instruments.
2. Use `ratecurve` to specify an interest-rate model for the `OISFuture` instrument object.
3. Use `finpricer` to specify a `Discount` pricing method for one or more `OISFuture` instruments.

Create an `OISFuture` instrument object for one or more OIS futures instruments to use in curve construction using this workflow:

1. Use `fininstrument` to create an `OISFuture` instrument object for one or more OIS future instruments.
2. Use `irbootstrap` to create an interest-rate curve (`ratecurve`) for one or more `OISFuture` instruments.

For more information on these workflows, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for an `OISFuture` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
OISFutureInst = fininstrument(InstrumentType,QuotedPrice=OIS_quoted_price,
Maturity=maturity_date,StartDate=start_date)
OISFutureInst = fininstrument( ___ ,Name=Value)
```

Description

`OISFutureInst = fininstrument(InstrumentType,QuotedPrice=OIS_quoted_price, Maturity=maturity_date,StartDate=start_date)` creates an `OISFuture` instrument object for one or more OIS future instruments by specifying `InstrumentType`, `QuotedPrice`, `Maturity`, and `StartDate`.

The `OISFuture` instrument supports many alternative reference rate (ARR) securities that are compliant with standards from the International Organization of Securities Commissions (IOSCO). For example, ARRs like SOFR, EONIA, SONIA, SARON, and TONAR focus on risk-free rate or near risk-free rates based on transactions of overnight funding.

`OISFutureInst = fininstrument( ___ ,Name=Value)` sets optional properties on page 11-2498 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `OISFutureInst = fininstrument("OISFuture",QuotedPrice=99.5,Maturity=datetime(2022,12,15),StartDate=datetime(2022,9,15))` creates an OIS future instrument. You can specify multiple name-value arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"OISFuture"` | string array with values of `"OISFuture"` | character vector with value `'OISFuture'` | cell array of character vectors with values of `'OISFuture'`

Instrument type, specified as a string with the value of `"OISFuture"`, a character vector with the value of `'OISFuture'`, an `NINST`-by-`1` string array with values of `"OISFuture"`, or an `NINST`-by-`1` cell array of character vectors with values of `'OISFuture'`.

Data Types: `char` | `cell` | `string`

`OISFuture` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `OISFutureInst = fininstrument("OISFuture",QuotedPrice=99.5,Maturity=datetime(2022,12,15),StartDate=datetime(2022,9,15))`

Required `OISFuture` Name-Value Arguments

`QuotedPrice` — OIS future quoted price

scalar numeric | numeric decimal

OIS future quoted price, specified as `QuotedPrice` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Maturity` — OIS future maturity date

datetime array | string array | date character vector

OIS future maturity date, specified as `Maturity` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OISFuture` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`StartDate` — OIS future underlying rate end date

datetime array | string array | date character vector

OIS future underlying rate end date, specified as `StartDate` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OISFuture` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `StartDate` property is stored as a datetime.

Optional OISFuture Name-Value Arguments

`Method` — Computation method
`"Compound"` (default) | string with value `"Compound"` or `"Average"` | string array with values of `"Compound"` or `"Average"` | character vector with value `'Compound'` or `'Average'` | cell array of character vectors with values of `'Compound'` or `'Average'`

Computation method, specified as `Method` and a scalar character vector or string or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `cell` | `char` | `string`

`Basis` — Day count basis
`2` (actual/360) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, specified as `Basis` and a scalar integer or an `NINST`-by-`1` vector of integers for the following:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Notional` — Notional principal amount
`100` (default) | scalar numeric | numeric vector

Notional principal amount, specified as `Notional` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`BusinessDayConvention` — Business day convention for cash flow dates
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day convention for cash flow dates, specified as `BusinessDayConvention` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as `Holidays` and dates using an `NINST`-by-`1` vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
OISFutureInst = fininstrument("OISFuture",Maturity=datetime(2022,12,15),QuotedPrice=99.5,Exercise
```

To support existing code, `OISFuture` also accepts serial date numbers as inputs, but they are not recommended.

`ProjectionCurve` — Projection curve used to price OIS future
`ratecurve.empty` (default) | `ratecurve` object | vector of `ratecurve` objects

Projection curve used to price OIS future, specified as `ProjectionCurve` and a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects. These objects must be created using `ratecurve`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `object`

`HistoricalFixing` — Historical fixing for `OISFuture`
`timetable.empty` (default) | timetable

Historical fixing for `OISFuture`, specified as `HistoricalFixing` and a timetable.

Data Types: `timetable`

`Name` — User-defined name for instrument
`""` (default) | string | character vector

User-defined name for the instrument, specified as `Name` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`QuotedPrice` — OIS future quoted price
scalar numeric | numeric vector

OIS future quoted price, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Maturity` — OIS future maturity date
datetime | vector of datetimes

OIS future maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`StartDate` — OIS future underlying end date
datetime | vector of datetimes

OIS future underlying end date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Method` — Computation method
`"Compound"` (default) | string with value `"Compound"` or `"Average"` | string array with values of `"Compound"` or `"Average"`

Computation method, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Notional` — Notional principal amount
`100` (default) | scalar numeric | numeric vector

Notional principal amount, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`BusinessDayConvention` — Business day convention for cash flows
`"actual"` (default) | scalar string | string array

Business day convention for cash flows, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`ProjectionCurve` — Projection curve used to price OIS future
`ratecurve.empty` (default) | `ratecurve` object | vector of `ratecurve` objects

Projection curve used to price OIS future, returned as a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects.

Data Types: `object`

`HistoricalFixing` — Historical fixing for `OISFuture`
`timetable.empty` (default) | timetable

Historical fixing for `OISFuture`, returned as a timetable.

Data Types: `timetable`

`Name` — User-defined name for instrument
`""` (default) | string | string array

User-defined name for the instrument, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

cashflows	Compute cash flow for FixedBond, FloatBond, Swap, FRA, STIRFuture, OISFuture, OvernightIndexedSwap, or Deposit instrument

Examples

Price SOFR Future Using `ratecurve` and Discount Pricer

This example shows the workflow to price an `OISFuture` instrument for a one-month SOFR future when you use a `ratecurve` object and a `Discount` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `OISFuture` instrument.

```
Settle = datetime(2021,1,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
```

```
                Dates: [10x1 datetime]
                Rates: [10x1 double]
               Settle: 15-Jan-2021
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `OISFuture` Instrument Object

Use `fininstrument` to create an `OISFuture` instrument object for a one-month SOFR future.

```
HFDates = datetime(2021,3,1) + caldays(0:3)';
HistFixing = timetable(HFDates,[0.02;0.04;0.04;0.02]);

% Data from the following: https://www.cmegroup.com/trading/interest-rates/stir/one-month-sofr_qu
Prices_1M = 99.97;
Maturity_1M = lbusdate(2021,3,[],[],'datetime');
StartDate_1M = fbusdate(2021,3,[],[],'datetime');
FutInstrument_1M = fininstrument("OISFuture",Maturity=Maturity_1M ,QuotedPrice=Prices_1M,StartDat
    HistoricalFixing=HistFixing,Name="1MonthSOFRFuture")
```

```
FutInstrument_1M =
  OISFuture with properties:

             QuotedPrice: 99.9700
                  Method: "average"
                   Basis: 2
               StartDate: 01-Mar-2021
                Maturity: 31-Mar-2021
                Notional: 100
    BusinessDayConvention: "actual"
                Holidays: NaT
          ProjectionCurve: [0x0 ratecurve]
         HistoricalFixing: [4x1 timetable]
                     Name: "1MonthSOFRFuture"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",DiscountCurve=myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `OISFuture` Instrument for SOFR Future

Use `price` to compute the price and sensitivities for the `OISFuture` instrument for a one-month SOFR future.

```
[Price,outPR] = price(outPricer,FutInstrument_1M,["all"])
```

```
Price = 0.0408
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
     Price        DV01
    _______    ___________

    0.04079    -0.00083163
```

Price Multiple SOFR Futures Using `ratecurve` and Discount Pricer

This example shows the workflow to price multiple `OISFuture` instruments for one-month SOFR futures and three-month SOFR futures when you use a `ratecurve` object and a `Discount` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `OISFuture` instruments.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `OISFuture` Instrument Objects for SOFR Futures

Use `fininstrument` to create an `OISFuture` instrument object for one-month SOFR futures.

```
HFDates = datetime(2021,3,1) + caldays(0:3)';
HistFixing = timetable(HFDates,[0.02;0.04;0.04;0.02]);

% Data from the following: https://www.cmegroup.com/trading/interest-rates/stir/one-month-sofr_qu
```

```
Prices_1M = [99.97 99.96 99.95]';
Maturity_1M = lbusdate(2021,[3 4 5]',[],[],'datetime');
StartDate_1M = fbusdate(2021,[3 4 5]',[],[],'datetime');
FutInstruments_1M = fininstrument("OISFuture",Maturity=Maturity_1M ,QuotedPrice=Prices_1M,StartDa
    HistoricalFixing=HistFixing,Name="1MonthSOFRFuture")
```

```
FutInstruments_1M=3×1 object
  3x1 OISFuture array with properties:

    QuotedPrice
    Method
    Basis
    StartDate
    Maturity
    Notional
    BusinessDayConvention
    Holidays
    ProjectionCurve
    HistoricalFixing
    Name
```

Use `fininstrument` to create an `OISFuture` instrument object for three-month SOFR futures.

```
% Data from the following: https://www.cmegroup.com/trading/interest-rates/stir/three-month-sofr_
Prices_3M = [99.92 99.895 99.84 99.74]';
Dates_3M_Maturity = thirdwednesday([6 9 12 3]',[2021 2021 2021 2022]','datetime');
Dates_3M_Start = thirdwednesday([3 6 9 12]',2021,'datetime');
FutInstruments_3M = fininstrument("OISFuture",Maturity=Dates_3M_Maturity, ...
    QuotedPrice=Prices_3M,StartDate=Dates_3M_Start,HistoricalFixing=HistFixing,Name="3MonthSOFRFu
```

```
FutInstruments_3M=4×1 object
  4x1 OISFuture array with properties:

    QuotedPrice
    Method
    Basis
    StartDate
    Maturity
    Notional
    BusinessDayConvention
    Holidays
    ProjectionCurve
    HistoricalFixing
    Name
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",DiscountCurve=myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price OISFuture Instruments for SOFR Futures

Use `price` to compute the prices for the `OISFuture` instruments for one-month and three-month SOFR futures.

```
Price = price(outPricer,[FutInstruments_1M ; FutInstruments_3M])
```

```
Price = 7×1

    0.0527
    0.0509
    0.0439
    0.1511
    0.1520
    0.1791
    0.1687
```

More About

OIS Future

An OIS future is a futures contract that has an overnight index swap as the underlying asset.

To support the LIBOR transition, the `OISFuture` instrument supports the adoption of alternative reference rates (ARR) like SOFR, EONIA, SONIA, SARON, and TONAR. The ARRs replace the LIBOR benchmark, which underpins many loans, mortgages, bonds, and interest-rate derivatives.

The secured overnight financing rate (SOFR) ARR tracks the overnight effective federal funds rate (which is a benchmark of the US short-term interest rate market). SOFR is becoming the benchmark rate for dollar-denominated derivatives and loans. Other countries have sought their own alternative rates, such as SONIA and EONIA. In the US, SOFR futures mature in three months and the start dates for an SOFR future coincide with international money market (IMM) expiration dates. The SOFR futures trade on maturity dates of money market futures and money market futures options, which are set by futures and options exchanges. These dates are always the third Wednesday of the last month of the quarter (March, June, September, December). You can determine these third Wednesday dates using `thirdwednesday`.

Version History

Introduced in R2021b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `OISFuture` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`OvernightIndexedSwap` | `STIRFuture` | `finmodel` | `finpricer` | `timetable`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Bootstrap ratecurve Object from STIRFuture, Deposit, and Swap BootInstruments” on page 11-2269
“Choose Instruments, Models, and Pricers” on page 1-53

STIRFuture

`STIRFuture` instrument object

Description

Create and price a `STIRFuture` instrument object for one or more STIR future instruments using this workflow:

1. Use `fininstrument` to create a `STIRFuture` instrument object for one or more STIR future instruments.
2. Use `ratecurve` to specify an interest-rate model for the `STIRFuture` instrument object.
3. Use `finpricer` to specify a `Discount` pricing method for one or more `STIRFuture` instruments.

Create a `STIRFuture` instrument object for one or more STIR future instruments to use in curve construction using this workflow:

1. Use `fininstrument` to create a `STIRFuture` instrument object for one or more STIR future instruments.
2. Use `irbootstrap` to create an interest-rate curve (`ratecurve`) for one or more `STIRFuture` instruments. In addition, you can use the `irbootstrap` optional name-value input argument `ConvexityAdjustment` to specify a convexity adjustment for the `STIRFuture` instruments.

For more information on these workflows, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `STIRFuture` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
STIRFutureInst = fininstrument(InstrumentType,QuotedPrice=quoted_stir_price,
Maturity=maturity_date,RateEndDate=rate_end_date)
STIRFutureInst = fininstrument( ___ ,Name=Value)
```

Description

`STIRFutureInst = fininstrument(InstrumentType,QuotedPrice=quoted_stir_price, Maturity=maturity_date,RateEndDate=rate_end_date)` creates a `STIRFuture` object for one or more STIR future instruments by specifying `InstrumentType`, `QuotedPrice`, `Maturity`, and `EndDate`.

`STIRFutureInst = fininstrument( ___ ,Name=Value)` sets optional properties on page 11-2508 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `STIRFutureInst =`

`fininstrument("STIRFuture",QuotedPrice=99.5,Maturity=datetime(2022,12,15),RateEndDate=datetime(2022,6,15))` creates a STIR future instrument. You can specify multiple name-value arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"STIRFuture"` | string array with values of `"STIRFuture"` | character vector with value `'STIRFuture'` | cell array of character vectors with values of `'STIRFuture'`

Instrument type, specified as a string with the value of `"STIRFuture"`, a character vector with the value of `'STIRFuture'`, an `NINST`-by-`1` string array 'with values of `"STIRFuture"`, or an `NINST`-by-`1` cell array of character vectors with values of `'STIRFuture'`.

Data Types: `char` | `cell` | `string`

`STIRFuture` Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `STIRFutureInst = fininstrument("STIRFuture",QuotedPrice=99.5,Maturity=datetime(2022,12,15),RateEndDate=datetime(2022,6,15))`

Required `STIRFuture` Name-Value Arguments

`QuotedPrice` — STIR future quoted price

scalar numeric | numeric vector

STIR future quoted price, specified as `QuotedPrice` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Maturity` — STIR future maturity date

datetime array | string array | date character vector

STIR future maturity date, specified as `Maturity` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `STIRFuture` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`RateEndDate` — STIR future underlying rate end date

datetime array | string array | date character vector

STIR future underlying rate end date, specified as `RateEndDate` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `STIRFuture` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `RateEndDate` property is stored as a datetime.

Optional STIRFuture Name-Value Arguments

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, specified as `Basis` and a scalar integer or an `NINST`-by-`1` vector of integers for the following:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Notional` — Notional principal amount
`100` (default) | scalar numeric | numeric vector

Notional principal amount, specified as `Notional` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`BusinessDayConvention` — Business day convention for cash flow dates
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day convention for cash flow dates, specified as `BusinessDayConvention` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.

- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as `Holidays` and dates using an `NINST`-by-`1` vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
STIRFutureInst = fininstrument("STIRFuture",Maturity=datetime(2022,12,15),QuotedPrice=99.5,Exerci
```

To support existing code, `STIRFuture` also accepts serial date numbers as inputs, but they are not recommended.

`ProjectionCurve` — Projection curve used to price STIR future
`ratecurve.empty` (default) | `ratecurve` object | vector of `ratecurve` objects

Projection curve used to price STIR future, specified as `ProjectionCurve` and a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects. These objects must be created using `ratecurve`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `object`

`Name` — User-defined name for instrument
`""` (default) | string | character vector

User-defined name for the instrument, specified as `Name` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`QuotedPrice` — STIR future quoted price
scalar numeric | numeric vector

STIR Future quoted price, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Maturity` — STIR future maturity date
datetime | vector of datetimes

STIR future maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`RateEndDate` — STIR future underlying rate end date
datetime | vector of datetimes

STIR future underlying rate end date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Notional` — Notional principal amount
`100` (default) | scalar numeric | numeric vector

Notional principal amount, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`BusinessDayConvention` — Business day convention for cash flow dates
`"actual"` (default) | scalar string | string array

Business day convention for cash flow dates, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`ProjectionCurve` — Projection curve used to price STIR future
`ratecurve.empty` (default) | `ratecurve` object | vector of `ratecurve` objects

Projection curve used to price STIR future, returned as a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects.

Data Types: `object`

`Name` — User-defined name for instrument
`""` (default) | string | string array

User-defined name for the instrument, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

cashflows	Compute cash flow for FixedBond, FloatBond, Swap, FRA, STIRFuture, OISFuture, OvernightIndexedSwap, or Deposit instrument

Examples

Price STIR Future Instrument Using `ratecurve` and Discount Pricer

This example shows the workflow to price a `STIRFuture` instrument when you use a `ratecurve` object and a `Discount` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `STIRFuture` instrument.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `STIRFuture` Instrument Object

Use `fininstrument` to create a `STIRFuture` instrument object.

```
STIRFuture = fininstrument("STIRFuture",Maturity=datetime(2022,9,15),QuotedPrice=99.5,RateEndDate
```

```
STIRFuture =
  STIRFuture with properties:

             QuotedPrice: 99.5000
                   Basis: 2
             RateEndDate: 15-Dec-2022
                Maturity: 15-Sep-2022
                Notional: 500
    BusinessDayConvention: "actual"
                Holidays: NaT
          ProjectionCurve: [0x0 ratecurve]
                     Name: "stir_future_instrument"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",DiscountCurve=myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `STIRFuture` Instrument

Use `price` to compute the price and sensitivities for the `STIRFuture` instrument.

```
[Price, outPR] = price(outPricer,STIRFuture,["all"])
```

```
Price = 97.3030

outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    ________

    97.303    0.041513
```

Price Multiple STIR Future Instruments Using `ratecurve` and Discount Pricer

This example shows the workflow to price multiple `STIRFuture` instruments when you use a `ratecurve` object and a `Discount` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `STIRFuture` instrument.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:
```

```
                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create STIRFuture Instrument Object

Use `fininstrument` to create a `STIRFuture` instrument object for three STIR future instruments.

```
STIRFuture = fininstrument("STIRFuture",Maturity=datetime([2022,4,15 ; 2022,5,15 ; 2022,6,15]),Qu
```

```
STIRFuture=3×1 object
  3x1 STIRFuture array with properties:

    QuotedPrice
    Basis
    RateEndDate
    Maturity
    Notional
    BusinessDayConvention
    Holidays
    ProjectionCurve
    Name
```

Create Discount Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",DiscountCurve=myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price STIRFuture Instruments

Use `price` to compute the prices for the `STIRFuture` instruments.

```
Price = price(outPricer,STIRFuture)
```

```
Price = 3×1

   98.2155
   98.8120
```

```
    97.6983
```

More About

STIR Future

A STIR future is a short-term interest-rate future.

A STIR future is a cash settled derivative contract on a specified term interest rate paid on a notional deposit. The price of a STIR future is quoted as 100.00 minus the rate of interest, meaning there is an inverse relationship between the direction in which the underlying interest rate is expected to move and the value of the contract.

Version History

Introduced in R2021b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `STIRFuture` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`OvernightIndexedSwap` | `OISFuture` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Bootstrap ratecurve Object from STIRFuture, Deposit, and Swap BootInstruments” on page 11-2269
“Choose Instruments, Models, and Pricers” on page 1-53

Cliquet

`Cliquet` instrument object

Description

Create and price a `Cliquet` instrument object for one or more Cliquet instruments using this workflow:

1. Use `fininstrument` to create a `Cliquet` instrument object for one or more Cliquet instruments.
2. Use `finmodel` to specify a `BlackScholes`, `Bates`, `Merton`, or `Heston` model for the `Cliquet` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify a `Rubinstein` pricing method for one or more `Cliquet` instruments.
   - When using a `BlackScholes`, `Heston`, `Bates`, or `Merton` model, use `finpricer` to specify an `AssetMonteCarlo` pricing method for one or more `Cliquet` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Cliquet` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
CliquetOpt = fininstrument(InstrumentType,ResetDates=reset_dates)
CliquetOpt = fininstrument( ___ ,Name=Value)
```

Description

`CliquetOpt = fininstrument(InstrumentType,ResetDates=reset_dates)` creates a `Cliquet` instrument object for one or more Cliquet instruments by specifying `InstrumentType` and sets properties on page 11-2517 using the required name-value argument for `ResetDates`.

`CliquetOpt = fininstrument( ___ ,Name=Value)` sets optional properties on page 11-2517 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `CliquetOpt = fininstrument("Cliquet",ResetDates=ResetDates,Name="Cliquet_option")` creates a `Cliquet` option. You can specify multiple name-value arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Cliquet"` | string array with values of `"Cliquet"` | character vector with value `'Cliquet'` | cell array of character vectors with values of `'Cliquet'`

Instrument type, specified as a string with the value of `"Cliquet"`, a character vector with the value of `'Cliquet'`, an `NINST`-by-`1` string array with values of `"Cliquet"`, or an `NINST`-by-`1` cell array of character vectors with values of `'Cliquet'`.

Data Types: `char` | `cell` | `string`

`Cliquet` Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `CliquetOpt = fininstrument("Cliquet",ResetDates=ResetDates,Name="Cliquet_option")`

Required Cliquet Name-Value Arguments

`ResetDates` — Reset dates when option strike is set

vector of datetimes

Reset dates when option strike is set, specified as `ResetDates` and a `1`-by-`NumDates` vector of datetimes. The last element corresponds to the maturity date of the `Cliquet` option.

A cliquet option is a path-dependent, exotic option that periodically settles and then resets its strike price at the level of the underlying asset at the time of settlement. The reset of the strike price is not conditional to the value of the underlying asset at the reset date.

Data Types: `datetime`

Optional Cliquet Name-Value Arguments

`OptionType` — Option type

`"call"` (default) | string with value `"call"` or `"put"` | string array with values of `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values of `'call'` or `'put'`

Option type, specified as `OptionType` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `string`

`ExerciseStyle` — Option exercise style

`"European"` (default) | string with value `"European"` | string array with values of `"European"` | character vector with value `'European'` | cell array of character vectors with values of `'European'`

Option exercise style, specified as `ExerciseStyle` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `string` | `char`

ReturnType — Return calculation type
"absolute" (default) | string with value "absolute" or "relative" | string array with values of "absolute" or "relative" | character vector with value 'absolute' or 'relative' | cell array of character vectors with values of 'absolute' or 'relative'

Option type, specified as ReturnType and a scalar string or character vector or an NINST-by-1 cell array of character vectors or string array.

Data Types: char | string

InitialStrike — Original strike price used for first reset date
0 (default) | nonnegative numeric | vector of nonnegative numeric

Original strike price used for first reset date, specified as InitialStrike and a scalar nonnegative numeric value or an NINST-by-1 vector of nonnegative numeric values.

Data Types: double

LocalCap — Local cap
inf (default) | nonnegative numeric | vector of nonnegative numeric

Local cap, specified as LocalCap and a scalar nonnegative numeric value or an NINST-by-1 vector of nonnegative numeric values.

Data Types: double

LocalFloor — Local floor
0 (default) | nonnegative numeric | vector of nonnegative numeric

Local floor, specified as LocalFloor and a scalar nonnegative numeric value or an NINST-by-1 vector of nonnegative numeric values.

Data Types: double

GlobalCap — Global cap
inf (default) | nonnegative numeric | vector of nonnegative numeric

Global cap, specified as GlobalCap and a scalar nonnegative numeric value or an NINST-by-1 vector of nonnegative numeric values.

Data Types: double

GlobalFloor — Global floor
0 (default) | nonnegative numeric | vector of nonnegative numeric

Global floor, specified as GlobalFloor and a scalar nonnegative numeric value or an NINST-by-1 vector of nonnegative numeric values.

Data Types: double

Name — User-defined name for instrument
" " (default) | string | string array | character vector | cell array of character vectors

User-defined name for one or more instruments, specified as Name and a scalar string or character vector or an NINST-by-1 cell array of character vectors or string array.

Data Types: char | cell | string

Properties

`ResetDates` — Reset dates when option strike is set
vector of datetimes

Reset dates when option strike is set, returned as a 1-by-`NumDates` vector of datetimes.

Data Types: `datetime`

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"`

Option type, returned as a scalar string.

Data Types: `string`

`ReturnType` — Return calculation type
`"absolute"` (default) | string with value `"absolute"` or `"relative"`

Option type, returned as a scalar string.

Data Types: `string`

`InitialStrike` — Original strike price used for first reset date
`0` (default) | nonnegative numeric |

Original strike price used for first reset date, returned as a scalar nonnegative numeric value.

Data Types: `double`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"`

Option exercise style, returned as a scalar string.

Data Types: `string`

`LocalCap` — Local cap
`inf` (default) | nonnegative numeric

Local cap, returned as a scalar nonnegative numeric value.

Data Types: `double`

`LocalFloor` — Local floor
`0` (default) | nonnegative numeric

Local floor, returned as a scalar nonnegative numeric value.

Data Types: `double`

`GlobalCap` — Global cap
`inf` (default) | nonnegative numeric

Global cap, returned as a scalar nonnegative numeric value.

Data Types: `double`

`GlobalFloor` — Global floor
`0` (default) | nonnegative numeric

Global floor, returned as a scalar nonnegative numeric value.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Examples

Price Absolute Return for Cliquet Instrument Using a Black-Scholes Model and Asset Monte Carlo Pricer

This example shows the workflow to price the absolute return for a `Cliquet` instrument when you use a `BlackScholes` model and an `AssetMonteCarlo` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,1,1);
Date = datetime(2021,1,1);
Rates = 0.10;
Basis = 1;
ZeroCurve = ratecurve('zero',Settle,Date,Rates,Basis=Basis)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 1
              Dates: 01-Jan-2021
              Rates: 0.1000
             Settle: 01-Jan-2020
       InterpMethod: "linear"
  ShortExtrapMethod: "next"
   LongExtrapMethod: "previous"
```

Create `Cliquet` Instrument Object

Use `fininstrument` to create a `Cliquet` instrument object.

```
ResetDates =  Settle + years(0:0.25:1);
CliquetOpt = fininstrument("Cliquet",ResetDates=ResetDates,Name="cliquet_option")
```

```
CliquetOpt =
  Cliquet with properties:

      OptionType: "call"
```

```
    ExerciseStyle: "european"
       ResetDates: [01-Jan-2020 00:00:00    01-Apr-2020 07:27:18    ...    ]
         LocalCap: Inf
       LocalFloor: 0
        GlobalCap: Inf
      GlobalFloor: 0
       ReturnType: "absolute"
    InitialStrike: NaN
             Name: "cliquet_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",Volatility=0.1)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.1000
    Correlation: 1
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",DiscountCurve=ZeroCurve,Model=BlackScholesModel,SpotPrice
```

```
outPricer =
  GBMMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 100
    SimulationDates: [02-Jan-2020    03-Jan-2020    04-Jan-2020    ...    ]
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.BlackScholes]
       DividendType: "continuous"
      DividendValue: 0
```

Price Cliquet Instrument

Use `price` to compute the price and sensitivities for the `Cliquet` instrument.

```
[Price, outPR] = price(outPricer,CliquetOpt,"all")
```

```
Price = 13.1885
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma        Lambda     Rho      Theta     Vega
    ______    ________    ___________    ______    ______    _____    ______

    13.189    0.13189    1.2434e-14       1       59.019      0      66.068
```

Price Relative Return for Cliquet Instrument Using a Black-Scholes Model and Asset Monte Carlo Pricer

This example shows the workflow to price a `Cliquet` instrument when you use a `BlackScholes` model and an `AssetMonteCarlo` pricing method. This example demonstrates how variations in caps and floors affect option prices on European Cliquet options.

This example uses three 1-year call cliquet options with quarterly observation dates. The first Cliquet option has no caps or floors, the second Cliquet option has a local floor, and the third Cliquet option has a local cap and a local floor.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,01,01);
Dates = datetime(2021,01,01);
Rate = 0.035;
Compounding = -1;
ZeroCurve = ratecurve('zero',Settle,Dates,Rate,Compounding=Compounding)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: 01-Jan-2021
               Rates: 0.0350
              Settle: 01-Jan-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BSModel = finmodel("BlackScholes",Volatility=0.20)
```

```
BSModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create Cliquet Instrument Objects with Quarterly Observation Dates

Use `fininstrument` to create the first `Cliquet` instrument object with no caps or floors.

```
ResetDates = Settle + years(0:0.25:1);

Cliquet = fininstrument("Cliquet",ResetDates=ResetDates,ReturnType="relative",LocalFloor="-inf",
```

```
Cliquet =
  Cliquet with properties:

       OptionType: "call"
    ExerciseStyle: "european"
       ResetDates: [01-Jan-2020 00:00:00    01-Apr-2020 07:27:18    ...    ]
         LocalCap: Inf
       LocalFloor: -Inf
        GlobalCap: Inf
      GlobalFloor: -Inf
       ReturnType: "relative"
    InitialStrike: NaN
             Name: "Vanilla_Cliquet"
```

Use `fininstrument` to create the second `Cliquet` instrument object with a local floor of 0%.

```
LFCliquet = fininstrument("Cliquet",ResetDates=ResetDates,ReturnType="relative",GlobalFloor="-in
```

```
LFCliquet =
  Cliquet with properties:

       OptionType: "call"
    ExerciseStyle: "european"
       ResetDates: [01-Jan-2020 00:00:00    01-Apr-2020 07:27:18    ...    ]
         LocalCap: Inf
       LocalFloor: 0
        GlobalCap: Inf
      GlobalFloor: -Inf
       ReturnType: "relative"
    InitialStrike: NaN
             Name: "LFCliquet"
```

Use `fininstrument` to create the third `Cliquet` instrument object with a local cap of 7% and a local floor of 0%.

```
LocalCap = 0.07;
LFLCCliquet = fininstrument("Cliquet",ResetDates=ResetDates,ReturnType="relative",LocalCap=Local
```

```
LFLCCliquet =
  Cliquet with properties:

       OptionType: "call"
    ExerciseStyle: "european"
       ResetDates: [01-Jan-2020 00:00:00    01-Apr-2020 07:27:18    ...    ]
         LocalCap: 0.0700
       LocalFloor: 0
        GlobalCap: Inf
      GlobalFloor: -Inf
       ReturnType: "relative"
```

```
    InitialStrike: NaN
             Name: "LFLCCLiquet"
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
SpotPrice = 100;
NumTrials =  5000;
MCPricer = finpricer("AssetMonteCarlo",DiscountCurve=ZeroCurve,Model=BSModel,...
                     SpotPrice=SpotPrice,SimulationDates=[Settle+years(0:0.25:1),Settle+calmonths
```

```
MCPricer =
  GBMMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 100
    SimulationDates: [01-Jan-2020 00:00:00    01-Feb-2020 00:00:00    ...    ]
          NumTrials: 5000
      RandomNumbers: []
              Model: [1x1 finmodel.BlackScholes]
       DividendType: "continuous"
      DividendValue: 0
```

Price Cliquet Instruments

Use `price` to compute the prices for the three `Cliquet` instruments.

```
Price = price(MCPricer,[Cliquet;LFCliquet;LFLCCliquet])
```

```
Price = 3×1

    0.0337
    0.1717
    0.1042
```

The underlying asset has good and poor performances when simulating Cliquet option returns. You can observe the effect of caps and floors on these performances when computing the payoff of the three Cliquet instruments:

- The first Cliquet option has no local floor, so it picks up all the poor performances. Since there is no local cap, none of the returns are capped for this Cliquet option.
- The price of the second Cliquet option is higher than the price of the first Cliquet option. The effect of the local floor on the second Cliquet option is that none of the performances below 0% are considered.
- The price of the third Cliquet option is lower than the price of the second Cliquet option because of the capped performances (returns above 7% are not considered), but it is higher than the price of the first Cliquet option with no local floor, since poor performances below 0% are not considered.

Price Multiple Cliquet Instruments Using Black-Scholes Model and Rubinstein Pricer

This example shows the workflow to price multiple `Cliquet` instruments when you use a `BlackScholes` model and a `Rubinstein` pricing method.

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,Basis=12)
```

```
myRC =
  ratecurve with properties:

                 Type: "zero"
          Compounding: -1
                Basis: 12
                Dates: 15-Sep-2023
                Rates: 0.0350
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Cliquet` Instrument Object

Use `fininstrument` to create a `Cliquet` instrument object for three Cliquet instruments.

```
ResetDates = Settle + years(0:0.25:1);
CliquetOpt = fininstrument("Cliquet",ResetDates=ResetDates,InitialStrike=[140;150;160],ExerciseS
```

```
CliquetOpt=3×1 object
  3x1 Cliquet array with properties:

    OptionType
    ExerciseStyle
    ResetDates
    LocalCap
    LocalFloor
    GlobalCap
    GlobalFloor
    ReturnType
    InitialStrike
    Name
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",Volatility=0.28)
```

```
BlackScholesModel =
  BlackScholes with properties:
```

```
     Volatility: 0.2800
    Correlation: 1
```

Create Rubinstein Pricer Object

Use `finpricer` to create a `Rubinstein` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",DiscountCurve=myRC,Model=BlackScholesModel,SpotPrice=135,Divider
```

```
outPricer =
  Rubinstein with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 135
    DividendValue: 0.0250
     DividendType: "continuous"
```

Price Cliquet Instruments

Use `price` to compute the prices and sensitivities for the three `Cliquet` instruments.

```
[Price, outPR] = price(outPricer,CliquetOpt,"all")
```

```
Price = 3×1

   28.1905
   25.3226
   23.8168


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Rho      Theta
    _____    ________    _______    _______    ______    ______    ______

    28.191    0.59697    0.020662    2.8588    105.38    60.643    -14.62


ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Rho       Theta
    _____    ________    _______    _______    ______    ______    _______

    25.323    0.41949    0.016816    2.2364    100.47    55.367    -11.708


ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Rho      Theta
```

```
    ______    _______    ________    ______    ______    ______    ______

    23.817    0.29729    0.011133    1.6851    93.219    51.616    -7.511
```

More About

Cliquet Option

A cliquet option, also called a "ratchet option," is a series of at-the-money (ATM) options, either puts or calls, where each successive option becomes active when the previous one expires.

A cliquet option is a series of forward start options, all related to each other. Each forward start option represents the advance purchase of a put, or call, option with an at-the-money strike price to be determined at a later date, typically when the option becomes active. A forward start option becomes active at a specified date in the future. The premium is paid in advance, while the time to expiration and the underlying security are established at the time the forward start option is purchased.

For example, a comparison of a European cliquet with a European vanilla option illustrates the behavior of a cliquet option. Assume that a cliquet call and put option has these characteristics:

```
Underlying index = FTST 100 index
Settle = June 19, 2019
Maturity = June 19, 2022
Intial Strike = 3000
% Assume that the underlying asset has the following values at these ResetDates:
ResetDate(1) = Strike = 3300
ResetDate(2) = Strike = 2700
ResetDate(3) = Strike = 2900
Local floor = 0
```

Assume that a vanilla call and put option has these characteristics:

```
Underlying index = FTST 100 index
Settle = June 19, 2019
Maturity = June 19, 2022
Strike = 3000
```

A three-year cliquet call on the FTST with annual resets is a series of three annual at-the-money spot calls. The initial strike is set at 3000. If at the end of year 1, the FTST closes at 3300, the first call matures in-the-money and the holder makes $300 in profit on the one-year start call. The call strike for year 2 is then reset at 3300. If at the end of year 2, the FTST closes at 2700, the call will expire worthless. The call strike for year 3 is then reset at 2700. If at the end of year 3 the underlying asset is trading at 2900, the call matures in-the-money and the holder makes a profit of $200. In summary, the holder has locked $500 in profit.

Year	Strike	Payoff at End of Each Year
1	$3000	$300
2	$3300	$0
3	$2700	$200

On the other hand, a three-year call vanilla option with a strike of 3000 will expire worthless.

A three-year cliquet put on the FTST with annual resets is a series of three annual at-the-money spot puts. The initial strike is set at 3000. If at the end of year 1, the FTST closes at 3300, the first put expires worthless. The put strike for year 2 is then reset at 3300. If at the end of year 2, the FTST closes at 2700, the put matures in-the-money and the holder makes $600 in profit on the second-year start put. The put strike for year 3 is then reset at 2700. If at the end of year 3 the underlying asset is trading at 2900, the put matures worthless. In summary, the holder has locked $600 in profit.

Year	Strike	Payoff at End of Each Year
1	$3000	0
2	$3300	$600
3	$2700	0

On the other hand, a three-year vanilla put option with a strike of $3000 will expire in-the-money with a $100 profit.

Algorithms

A cliquet option is constructed as a series of forward start options. The premium and observation (reset) dates are set in advance and its payoff depends on the returns of the underlying asset at given observation or reset dates. This return can be based in terms of absolute or relative returns. The return during the period [*Tn*-1, *Tn*] is defined as follows:

$$R_n = \begin{Bmatrix} \frac{S_{T_n} - S_{T_{n-1}}}{S_{T_{n-1}}} relative\ return \\ S_{T_n} - S_{T_{n-1}} absolute\ return \end{Bmatrix}$$

Where $n = 1,\ldots,Nobs$ and *Nobs* is the number of observations (reset dates) during the life of the contract, *Sn* is the price of the underlying asset at observation time *n*.

Since the cliquet instrument is built as a series of forward start options, then its payoff is the sum of the returns:

$$\text{Payoff cliquet} = \sum_{i=1}^{n} (Ri)$$

Depending on the underlying asset performance, there would be positive and negative returns, and the presence of caps and floors play a big role in the payoff and price of the cliquet instrument.

If a local cap (LC) and a local floor (LF) of the individual returns are considered, then the payoff of the cliquet option is the sum of the returns, capped and floored by LC and LF, at every observation time *tn*:

$$\text{LCLFCliquetPayoff} = \sum_{i=1}^{n} \max(LF, \min(LC, Ri))$$

At maturity, the sum of these modified local returns might also be globally capped and floored. If a global cap (GC) and a global floor (GF) are also considered, the cliquet option has a final payoff of:

$$\text{GCGFCliquetPayoff} = \max\left[GF, \min(GC, \textstyle\sum_{i=1}^{n} \max(\text{LF}, \min(\text{LC}, \text{RI}))\right]$$

In this case the total sum of all the cliquets is now globally capped and floored.

There are two popular cliquets in the market, the globally capped and locally floored cliquet (GCLF) and the globally floored and locally capped cliquet (GFLC). Their payoffs are defined as follows:

$$\text{GCLFCliquetPayoff} = \min(GC, \sum_{i=1}^{n} \max(LF, Ri))$$

$$\text{GFLCCliquetPayoff} = \max(GF, \sum_{i=1}^{n} \min(LF, Ri))$$

In summary, the payoff of a cliquet instrument is the sum of the capped and floored returns.

Version History

Introduced in R2021b

See Also

Functions

`finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

OvernightIndexedSwap

`OvernightIndexedSwap` instrument object

Description

Create and price an `OvernightIndexedSwap` instrument object for one or more Overnight Indexed Swap (OIS) instruments using this workflow:

1. Use `fininstrument` to create an `OvernightIndexedSwap` instrument object for one or more OIS instruments.
2. Use `ratecurve` to specify a curve model for the `OvernightIndexedSwap` instrument object.
3. Use `finpricer` to specify a `Discount` pricing method for one or more `OvernightIndexedSwap` instruments when using a `ratecurve` object.

Create an `OvernightIndexedSwap` instrument object for one or more OIS instruments to use in curve construction using this workflow:

1. Use `fininstrument` to create an `OvernightIndexedSwap` instrument object for one or more OIS instruments.
2. Use `irbootstrap` to create an interest-rate curve (`ratecurve`) for one or more `OvernightIndexedSwap` instruments.

For more information on these workflows, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for an `OvernightIndexedSwap` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
OvernightIndexedSwapInst = fininstrument(InstrumentType,
Maturity=maturity_date,LegRate=leg_rate)
OvernightIndexedSwapInst = fininstrument( ___ ,Name=Value)
```

Description

`OvernightIndexedSwapInst = fininstrument(InstrumentType, Maturity=maturity_date,LegRate=leg_rate)` creates a `OvernightIndexedSwap` object for one or more OIS instruments by specifying `InstrumentType` and sets the properties on page 11-2533 for the required name-value arguments `Maturity` and `LegRate`. The `OvernightIndexedSwap` instrument supports vanilla Overnight Indexed Swaps, amortizing Overnight Indexed Swaps, and forward Overnight Indexed Swaps.

`OvernightIndexedSwapInst = fininstrument( ___ ,Name=Value)` sets optional properties on page 11-2533 using additional name-value arguments in addition to the required arguments in the

previous syntax. For example, `OvernightIndexedSwapInst = fininstrument("OvernightIndexedSwap",Maturity=datetime(2019,1,30),LegRate=[0.06 0.12],LegType=["fixed","fixed"],Basis=1,Notional=100,StartDate=datetime(2018,1,30),DaycountAdjustedCashFlow=true,BusinessDayConvention="follow",ProjectionCurve=ratecurve,Name="overnight_indexed_swap_instrument")` creates an `OvernightIndexedSwap` instrument with a maturity of January 30, 2019. You can specify multiple name-value arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"OvernigntIndexedSwap"` | string array with values of `"OvernigntIndexedSwap"` | character vector with value `'OvernigntIndexedSwap'` | cell array of character vectors with values of `'OvernigntIndexedSwap'`

Instrument type, specified as a string with the value of `"OvernigntIndexedSwap"`, a character vector with the value of `'OvernigntIndexedSwap'`, an `NINST`-by-`1` string array with values of `"OvernigntIndexedSwap"`, or an `NINST`-by-`1` cell array of character vectors with values of `'OvernigntIndexedSwap'`.

Data Types: `char` | `cell` | `string`

`OvernightIndexedSwap` Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `OvernightIndexedSwapInst = fininstrument("OvernightIndexedSwap",Maturity=datetime(2019,1,30),LegRate=[0.06 0.12],LegType=["fixed","fixed"],Basis=1,Notional=100,StartDate=datetime(2018,1,30),DaycountAdjustedCashFlow=true,BusinessDayConvention="follow",ProjectionCurve=ratecurve,Name="overnight_indexed_swap_instrument")`

Required `OvernightIndexedSwap` Name-Value Arguments

`Maturity` — Swap maturity date

datetime array | string array | date character vector

Swap maturity date, specified as `Maturity` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OvernightIndexedSwap` also accepts serial date numbers as inputs, but they are not recommended.

If you use date characters vector or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`LegRate` — Leg rate in decimal values

matrix

Leg rate in decimal values, specified as `LegRate` and a `NINST`-by-`2` matrix. Each row can be defined as one of the following:

- `[CouponRate Spread]` (fixed-float)
- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

`CouponRate` is the decimal annual rate. `Spread` is the number of basis points in decimals over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.

Data Types: `double`

Optional OvernightIndexedSwap Name-Value Arguments

`LegType` — `OvernightIndexedSwap` leg type
`["fixed","float"]` for each instrument (default) | cell array of character vectors with values `{'fixed','fixed'}`, `{'fixed','float'}`, `{'float','fixed'}`, or `{'float','float'}` | string array with values `["fixed","fixed"]`, `["fixed","float"]`, `["float","fixed"]`, or `["float","float"]`

`OvernightIndexedSwap` leg type, specified as `LegType` and a cell array of character vectors or a string array with the supported values. The `LegType` defines the interpretation of the values entered in `LegRate`.

Data Types: `cell` | `string`

`ProjectionCurve` — Rate curve for projecting floating cash flows
`ratecurve.empty` (default) | scalar `ratecurve` object | vector of `ratecurve` objects

Rate curve for projecting floating cash flows, specified as `ProjectionCurve` and a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects. You must create this object using `ratecurve`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `object`

`Reset` — Frequency of payments per year
`[2 2]` (default) | numeric value of `0`, `1`, `2`, `3`, `4`, `6`, or `12` | matrix

Frequency of payments per year, specified as `Reset` and a scalar or a `NINST`-by-`2` matrix if `Reset` is different for each leg) with one of the following values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day count basis representing the basis for each leg
`[0 0]` (actual/actual) (default) | integer from `0` to `13`

Day count basis representing the basis for each leg, specified as `Basis` and a `NINST`-by-`1` matrix (or `NINST`-by-`2` matrix if `Basis` is different for each leg).

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)

- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)
- 11 — 30/360E (ICMA)
- 12 — actual/365 (ISDA)
- 13 — BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Notional` — Notional principal amount
`100` (default) | scalar numeric | numeric vector

Notional principal amount, specified as `Notional` and a scalar numeric or an `NINST`-by-`1` numeric vector.

`Notional` accepts a scalar for a principal amount (or a `NINST`-by-`2` matrix if `Notional` is different for each leg).

Data Types: `double`

`HistoricalFixing` — Historical fixing data
`timetable.empty` (default) | timetable

Historical fixing data, specified as `HistoricalFixing` and a timetable.

Note If you are creating one or more `OvernightIndexedSwap` instruments and use a timetable, the timetable specification applies to all of the `OvernightIndexedSwap` instruments. `HistoricalFixing` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `timetable`

`ResetOffset` — Lag in rate setting
`[0 0]` (default) | vector

Lag in rate setting, specified as `ResetOffset` and a `NINST`-by-`2` matrix.

Data Types: `double`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day conventions, specified as `BusinessDayConvention` and string (or `NINST`-by-`2` string array if `BusinessDayConvention` is different for each leg) or a character vector (or `NINST`-by-`2` cell array of character vectors if `BusinessDayConvention` is different for each leg). The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as `Holidays` and dates using an `NINST`-by-`1` vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
OvernightIndexedSwapInst = fininstrument("OvernightIndexedSwap",Maturity=datetime(2025,12,15),Leg
```

To support existing code, `OvernightIndexedSwap` also accepts serial date numbers as inputs, but they are not recommended.

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`[true true]` (in effect) (default) | logical with value of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, specified as `EndMonthRule` and a logical value of `true` or `false` using a `NINST`-by-`1` matrix (or `NINST`-by-`2` matrix if `EndMonthRule` is different for each leg).

- If you set `EndMonthRule` to `false`, the software ignores the rule, meaning that a payment date is always the same numerical day of the month.
- If you set `EndMonthRule` to `true`, the software sets the rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`StartDate` — Date `OvernightIndexedSwap` starts payments
`Settle` date (default) | datetime array | string array | date character vector

Date `OvernightIndexedSwap` starts payments, specified as `StartDate` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OvernightIndexedSwap` also accepts serial date numbers as inputs, but they are not recommended.

Use `StartDate` to price a forward `OvernightIndexedSwap`, that is, an `OvernightIndexedSwap` that starts at a future date.

If you use a date character vector or string, the format must be recognizable by `datetime` because the `StartDate` property is stored as a datetime.

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for the instrument, specified as `Name` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Maturity` — Maturity date
scalar datetime | vector of datetimes

Maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`LegRate` — Leg rate
matrix

Leg rate, returned as a `NINST`-by-`2` matrix of decimal values, with each row defined as one of the following:

- `[CouponRate Spread]` (fixed-float)
- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

Data Types: `double`

`LegType` — Leg type
`["fixed","float"]` for each instrument (default) | string array with values `["fixed","fixed"]`, `["fixed","float"]`, `["float","fixed"]`, or `["float","float"]`

Leg type, returned as a string array with the values `["fixed","fixed"]`, `["fixed","float"]`, `["float","fixed"]`, or `["float","float"]`.

Data Types: `string`

`ProjectionCurve` — Rate curve used in generating future cash flows
`ratecurve.empty` (default) | scalar `ratecurve` object | vector of `ratecurve` objects

Rate curve used in projecting the future cash flows, returned as a `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects.

Data Types: `object`

`Reset` — Reset frequency per year for each swap
`[2 2]` (default) | vector

Reset frequency per year for each swap, returned as an 1-by-2 matrix.

Data Types: `double`

Basis — Day count basis
`[0 0]` (actual/actual) (default) | integer from `0` to `13`

Day count basis, returned as an 1-by-2 matrix.

Data Types: `double`

ResetOffset — Lag in rate setting
`[0 0]` (default) | matrix

Lag in rate setting, returned as an `NINST`-by-2 matrix.

Data Types: `double`

Notional — Notional principal amount
`100` (default) | scalar numeric | numeric vector

Notional principal amount, returned as a scalar numeric or an `NINST`-by-1 numeric vector.

Data Types: `double`

HistoricalFixing — Historical fixing data
`timetable.empty` (default) | timetable

Historical fixing data, returned as a timetable.

Data Types: `timetable`

BusinessDayConvention — Business day conventions
`"actual"` (default) | string | string array

Business day conventions, returned as a string or a `NINST`-by-2 string array if `BusinessDayConvention` is different for each leg.

Data Types: `char` | `cell` | `string`

Holidays — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-1 vector of datetimes.

Data Types: `datetime`

EndMonthRule — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`[true true]` (in effect) (default) | logical with value of `true` or `false` | vector of logicals with values of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, returned as an `NINST`-by-1 matrix (or `NINST`-by-2 matrix if `EndMonthRule` is different for each leg).

Data Types: `logical`

StartDate — Date `OvernightIndexedSwap` starts payments
`Settle` date (default) | scalar datetime | vector of datetimes

Date `OvernightIndexedSwap` starts payments, returned as a scalar datetime or an `NINST`-by-1 vector of datetimes.

Data Types: `datetime`

`Name` — User-defined name for instrument
`""` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

cashflows	Compute cash flow for FixedBond, FloatBond, Swap, FRA, STIRFuture, OISFuture, OvernightIndexedSwap, or Deposit instrument

Examples

Price Overnight Indexed Swap Instrument Using `ratecurve` and Discount Pricer

This example shows the workflow to price an `OvernightIndexedSwap` instrument when you use a `ratecurve` object and a `Discount` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `OvernightIndexedSwap` instrument.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create `OvernightIndexedSwap` Instrument Object

Use `fininstrument` to create an `OvernightIndexedSwap` instrument object.

```
OvernightIndexedSwap = fininstrument("OvernightIndexedSwap",Maturity=datetime(2022,9,15),LegRate=
```

```
OvernightIndexedSwap =
  OvernightIndexedSwap with properties:
```

```
                 LegRate: [0.0220 0.0190]
                 LegType: ["float"    "fixed"]
                   Reset: [2 2]
                   Basis: [0 0]
                Notional: 100
         HistoricalFixing: [0x0 timetable]
              ResetOffset: [0 0]
           ProjectionCurve: [1x1 ratecurve]
    BusinessDayConvention: ["actual"    "actual"]
                 Holidays: NaT
             EndMonthRule: [1 1]
                StartDate: NaT
                 Maturity: 15-Sep-2022
                     Name: "overnight_swap_instrument"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",DiscountCurve=myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `OvernightIndexedSwap` Instrument

Use `price` to compute the price and sensitivities for the `OvernightIndexedSwap` instrument.

```
[Price, outPR] = price(outPricer,OvernightIndexedSwap,["all"])
```

```
Price = 3.0226
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    ________

    3.0226    -0.02915
```

Price Multiple Overnight Indexed Swap Instruments Using `ratecurve` and Discount Pricer

This example shows the workflow to price multiple `OvernightIndexedSwap` instruments when you use a `ratecurve` object and a `Discount` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `OvernightIndexedSwap` instrument.

```
Settle = datetime(2020,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `OvernightIndexedSwap` Instrument Object

Use `fininstrument` to create an `OvernightIndexedSwap` instrument object for three Overnight Indexed Swap instruments.

```
OvernightIndexedSwap = fininstrument("OvernightIndexedSwap",Maturity=datetime([2022,9,15 ; 2023,9
```

```
OvernightIndexedSwap=3×1 object
  3x1 OvernightIndexedSwap array with properties:

    LegRate
    LegType
    Reset
    Basis
    Notional
    HistoricalFixing
    ResetOffset
    ProjectionCurve
    BusinessDayConvention
    Holidays
    EndMonthRule
    StartDate
    Maturity
    Name
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",DiscountCurve=myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price OvernightIndexedSwap Instruments

Use `price` to compute the prices for the `OvernightIndexedSwap` instruments.

```
Price = price(outPricer,OvernightIndexedSwap)
```

```
Price = 3×1

   -0.7832
   -0.7336
   -0.2178
```

More About

Overnight Indexed Swap

An Overnight Indexed Swap (OIS) is an interest-rate swap over some fixed term where the periodic floating payment is based on a return calculated from a daily compound interest investment.

The overnight indexed swap denotes an interest-rate swap involving the overnight rate being exchanged for a fixed interest rate. An overnight indexed swap uses an overnight rate index such as the federal funds rate as the underlying rate for the floating leg, while the fixed leg is set at a rate agreed on by both parties.

Version History

Introduced in R2021b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `OvernightIndexedSwap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`STIRFuture` | `OISFuture` | `finmodel` | `finpricer` | `irbootstrap`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Bootstrap ratecurve Object from STIRFuture, Deposit, and Swap BootInstruments” on page 11-2269
“Choose Instruments, Models, and Pricers” on page 1-53

PartialLookback

`PartialLookback` instrument

Description

Create and price a `PartialLookback` instrument object for one or more Partial Lookback instruments using this workflow:

1. Use `fininstrument` to create a `PartialLookback` instrument object for one or more Partial Lookback instruments.
2. Use `finmodel` to specify a `BlackScholes`, `Heston`, `Bates`, or `Merton` model for the `PartialLookback` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify a `HeynenKat` pricing method for one or more `PartialLookback` instruments.
   - When using a `BlackScholes`, `Heston`, `Bates`, or `Merton` model, use `finpricer` to specify an `AssetMonteCarlo` pricing method for one or more `PartialLookback` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `PartialLookback` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
PartialLookbackObj = fininstrument(InstrumentType,ExerciseDate=exercise_date,
Strike=strike_value,MonitorDate=monitor_date)
PartialLookbackObj = fininstrument( ___ ,Name=Value)
```

Description

`PartialLookbackObj = fininstrument(InstrumentType,ExerciseDate=exercise_date, Strike=strike_value,MonitorDate=monitor_date)` creates a `PartialLookback` object for one or more Partial Lookback instruments by specifying `InstrumentType` and sets the properties on page 11-2543 for the required name-value arguments `Strike`, `ExerciseDate`, and `MonitorDate`.

The `PartialLookback` instrument supports fixed-strike and floating-strike partial lookback options. To compute the value of a floating-strike partial lookback option, the `Strike` must be specified as `NaN`. For more information on a `PartialLookback` instrument, see “More About” on page 11-2550.

`PartialLookbackObj = fininstrument( ___ ,Name=Value)` sets optional

properties on page 11-2543 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `LookbackObj =`

`fininstrument("Lookback",Strike=100,ExerciseDate=datetime(2022,1,30),MonitorDate=datetime(2021,1,30),OptionType="put",ExerciseStyle="European",Name="partial_lookback_option")` creates a `PartialLookback` put option with an European exercise. You can specify multiple name-value arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"PartialLookback"` | string array with values of `"PartialLookback"` | character vector with value `'PartialLookback'` | cell array of character vectors with values of `'PartialLookback'`

Instrument type, specified as a string with the value of `"PartialLookback"`, a character vector with the value of `'PartialLookback'`, an `NINST`-by-`1` string array with values of `"PartialLookback"`, or an `NINST`-by-`1` cell array of character vectors with values of `'PartialLookback'`.

Data Types: `char` | `cell` | `string`

`PartialLookback` Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `LookbackObj = fininstrument("Lookback",Strike=100,ExerciseDate=datetime(2022,1,30),MonitorDate=datetime(2021,1,30),OptionType="put",ExerciseStyle="European",Name="partial_lookback_option")`

Required Lookback Name-Value Arguments

`Strike` — Option strike price value

nonnegative numeric | vector of nonnegative values | `NaN`

Option strike price value, specified as `Strike` and a scalar nonnegative numeric or an `NINST`-by-`1` vector of nonnegative values for a fixed-strike `PartialLookback` option. For a floating-strike partial lookback option, specify `Strike` as a `NaN` or an `NINST`-by-`1` vector of `NaN`s.

Data Types: `double`

`ExerciseDate` — Option exercise date

datetime array | string array | date character vector

Option exercise date, specified as `ExerciseDate` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one `ExerciseDate` on the option expiry date.

To support existing code, `PartialLookback` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `ExerciseDate` property is stored as a datetime.

`MonitorDate` — Predetermined lookback monitoring date
datetime array | string array | date character vector

Predetermined lookback monitoring date, specified as `MonitorDate` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

- For a fixed-strike partial lookback, the monitoring period is [`MonitorDate`, `ExerciseDate`]. The `MonitorDate` is the start date for a fixed-strike partial lookback option.
- For a floating-strike partial lookback, the monitoring period is [`Settle`, `MonitorDate`], where `Settle` is < `MonitorDate` < `ExerciseDate`. The `MonitorDate` is the end date for a floating-strike partial lookback option.

To support existing code, `PartialLookback` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `ExerciseDate` property is stored as a datetime.

Optional PartialLookback Name-Value Arguments

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Option type, specified as `OptionType` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` or `"American"` | string array with values `"European"` or `"American"` | character vector with value `'European'` or `'American'` | cell array of character vectors with values `'European'` or `'American'`

Option exercise style, specified as `ExerciseStyle` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `string` | `cell` | `char`

`AssetMinMax` — Maximum or minimum underlying asset price
`NaN` where `SpotPrice` of the underlying asset is used (default) | scalar numeric | numeric vector

Maximum or minimum underlying asset price, specified as `AssetMinMax` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`StrikeScaler` — Degree of partiality for floating-strike partial lookback
`1` (default) | scalar numeric | numeric vector

Degree of partiality for a floating-strike partial lookback, specified as `StrikeScaler` and a scalar numeric or an `NINST`-by-`1` numeric vector. The `StrikeScaler` value indicates the percentage of the `Strike` that is fixed above or below the `AssetMinMax` value.

- For a call floating-strike partial lookback, the `StrikeScaler` is ≥ 1.

- For a put floating-strike partial lookback, 0 < `StrikeScaler` ≤ 1.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as `Name` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Strike` — Option strike price value
nonnegative numeric | vector of nonnegative values

Option strike price value, returned as a scalar nonnegative numeric or an `NINST`-by-`1` vector of nonnegative values.

Data Types: `double`

`ExerciseDate` — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`MonitorDate` — Predetermined lookback monitoring date
datetime | vector of datetimes

Predetermined monitoring date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values `"call"` or `"put"`

Option type, returned as a scalar string or an `NINST`-by-`1` string array with values of `"call"` or `"put"`.

Data Types: `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` or `"American"` | string array with value `"European"` or `"American"`

Option exercise style, returned as a scalar string or an `NINST`-by-`1` string array with values of `"European"` or `"American"`.

Data Types: `string`

`AssetMinMax` — Maximum or minimum underlying asset price
`NaN` where `SpotPrice` of the underlying asset is used (default) | scalar numeric | numeric vector

Maximum or minimum underlying asset price, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`StrikeScaler` — Degree of partiality for partial floating-strike lookback
1 (default) | scalar numeric | numeric vector

Degree of partiality for partial floating-strike lookback, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as an `NINST`-by-`1` string array.

Data Types: `string`

Examples

Price Partial Lookback Instrument Using Black-Scholes Model and Heynen-Kat Pricer

This example shows the workflow to price a floating-strike `PartialLookback` instrument when you use a `BlackScholes` model and a `HeynenKat` pricing method.

Create `PartialLookback` Instrument Object

Use `fininstrument` to create an `PartialLookback` instrument object.

```
PartialLookbackOpt = fininstrument("PartialLookback",ExerciseDate=datetime(2022,9,15),Strike=NaN
```

```
PartialLookbackOpt =
  PartialLookback with properties:

     MonitorDate: 15-Sep-2021
    StrikeScaler: 0.7500
      OptionType: "put"
          Strike: NaN
     AssetMinMax: 98
   ExerciseStyle: "european"
    ExerciseDate: 15-Sep-2022
            Name: "partial_lookback_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",Volatility=0.32)
```

```
BlackScholesModel =
  BlackScholes with properties:

    Volatility: 0.3200
   Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,Basis=12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `HeynenKat` Pricer Object

Use `finpricer` to create a `HeynenKat` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",Model=BlackScholesModel,DiscountCurve=myRC,SpotPrice=100,Divide
```

```
outPricer =
  HeynenKat with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0500
     DividendType: "continuous"
```

Price `PartialLookback` Instrument

Use `price` to compute the price and sensitivities for the `PartialLookback` instrument.

```
[Price, outPR] = price(outPricer,PartialLookbackOpt,["all"])
```

```
Price = 24.8148
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Theta       Rho
```

```
    ______    _______    ________    ______    ______    _______    _______

    24.815    0.27297    0.012438     1.1      131.33    -5.0942    -193.51
```

Price Multiple Partial Lookback Instruments Using Black-Scholes Model and Heynen-Kat Pricer

This example shows the workflow to price multiple floating-strike `PartialLookback` instruments when you use a `BlackScholes` model and a `HeynenKat` pricing method.

Create `PartialLookback` Instrument Object

Use `fininstrument` to create an `PartialLookback` instrument object for three Partial Lookback instruments.

```
PartialLookbackOpt = fininstrument("PartialLookback",ExerciseDate=datetime([2022,9,15 ; 2022,10,1
```

```
PartialLookbackOpt=3×1 object
  3x1 PartialLookback array with properties:

    MonitorDate
    StrikeScaler
    OptionType
    Strike
    AssetMinMax
    ExerciseStyle
    ExerciseDate
    Name
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",Volatility=0.32)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3200
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,Basis=12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
```

```
       Compounding: -1
             Basis: 12
             Dates: 15-Sep-2023
             Rates: 0.0350
            Settle: 15-Sep-2018
      InterpMethod: "linear"
ShortExtrapMethod: "next"
 LongExtrapMethod: "previous"
```

Create HeynenKat Pricer Object

Use `finpricer` to create a `HeynenKat` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",Model=BlackScholesModel,DiscountCurve=myRC,SpotPrice=100,Divider
```

```
outPricer =
  HeynenKat with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0500
     DividendType: "continuous"
```

Price PartialLookback Instruments

Use `price` to compute the prices and sensitivities for the `PartialLookback` instruments.

```
[Price, outPR] = price(outPricer,PartialLookbackOpt,["all"])
```

```
Price = 3×1

   24.8148
   25.2306
   25.6545


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Theta       Rho
    _____    _______    ________    ______    ______    _______    _______

    24.815    0.27297    0.012438     1.1      131.33    -5.0942    -193.51


ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Theta       Rho
    _____    _______    ________    ______    ______    _______    _______
```

```
    25.231    0.27694    0.012349    1.0976    133.05    -5.0265    -198.37

ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Theta      Rho
    ______    _______    ________    ______    ______    _______    ______

    25.655    0.28099    0.012264    1.0953    134.81    -4.9578    -203.4
```

Price Partial Lookback Instrument Using Heston Model and Asset Monte-Carlo Pricer

This example shows the workflow to price a fixed-strike `PartialLookback` instrument when you use a `Heston` model and an `AssetMonteCarlo` pricing method.

Create `PartialLookback` Instrument Object

Use `fininstrument` to create an `PartialLookback` instrument object.

```
PartialLookbackOpt = fininstrument("PartialLookback",ExerciseDate=datetime(2022,9,15),Strike=102
```

```
PartialLookbackOpt =
  PartialLookback with properties:

     MonitorDate: 15-Sep-2021
     StrikeScaler: 1
       OptionType: "call"
           Strike: 102
      AssetMinMax: NaN
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
             Name: "partial_lookback_option"
```

Create `Heston` Model Object

Use `finmodel` to create a `Hestone` model object.

```
HestonModel = finmodel("Heston",V0=0.032,ThetaV=0.1,Kappa=0.003,SigmaV=0.2,RhoSV=-0.9)
```

```
HestonModel =
  Heston with properties:

        V0: 0.0320
    ThetaV: 0.1000
     Kappa: 0.0030
    SigmaV: 0.2000
     RhoSV: -0.9000
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
```

```
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,Basis=12)

myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",DiscountCurve=myRC,Model=HestonModel,SpotPrice=100,simula

outPricer =
  HestonMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 100
    SimulationDates: [15-Oct-2018    15-Nov-2018    15-Dec-2018    ...    ]
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Heston]
       DividendType: "continuous"
      DividendValue: 0
```

Price `PartialLookback` Instrument

Use `price` to compute the price and sensitivities for the `PartialLookback` instrument.

```
[Price, outPR] = price(outPricer,PartialLookbackOpt,["all"])

Price = 19.9479

outPR =
  priceresult with properties:

       Results: [1x8 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×8 table
    Price      Delta       Gamma      Lambda     Rho      Theta     Vega     VegaLT
    _____    _______    _________    ______    _____    ______    _____    ______
```

```
    19.948    0.93159    0.0084898    4.6701    283.87    1.9218    48.04    2.666
```

More About

Partial Lookback Option

A partial lookback option gives the investor the right to exercise the option with the highest (or lowest) price of the underling asset during the partial lookback period.

Partial lookback options are called fractional lookback options because:

- The extreme values (*Smax* and *Smin*) are monitored during a subset of the lives of the options
- Only a fraction of the floating-strike values are in effect

For the latter, the factor λ (lambda) is introduced. The λ factor, represented by the optional name-value argument `StrikeScaler`, is a constant and enables the creation of the fractional floating-strike lookback option where the strike is fixed at some percentage above or below the actual extreme values (*Smax* and *Smin*).

- For calls when $\lambda \geq 1$, the call floating strike increases
- For puts when $0 \leq \lambda \leq 1$, the put floating strike decreases

Financial Instruments Toolbox software supports two types of partial lookback options: fixed and floating. The fixed-strike partial lookback option is similar to a standard fixed-strike lookback option, but the lookback period starts at a predetermined date (*T*) after the settlement date of the option. The payoff for this options is

- *Max*(0, *Smax* - *K*) for a call
- *Max*(0, *K* -*Smin*) for a put

where

Smax is the maximum value of underlying asset during the monitoring period.

Smin is the minimum value of underlying asset during the monitoring period.

K is the strike price.

The floating-strike partial lookback option is similar to a standard floating-strike lookback option, but the lookback period starts at settle and ends at a predetermined date(*T*) before expiration.

The payoff for this options is

- *Max*(0, *S* - λ × *Smin*) for a call
- *Max*(0, λ × *Smax* - *S*) for a put

where

Smax is the maximum value of underlying asset during the monitoring period.

Smin is the minimum value of underlying asset during the monitoring period.

K is the strike price.

S is the price of underlying asset.

λ, represented by `StrikeScaler`, is the degree of partiality.

Version History

Introduced in R2021b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `PartialLookback` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`Lookback` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

ConvertibleBond

`ConvertibleBond` instrument object

Description

Create and price a `ConvertibleBond` instrument object for one of more Convertible Bond instruments using this workflow:

1. Use `fininstrument` to create a `ConvertibleBond` instrument object for one of more Convertible Bond instruments.
2. Use `finmodel` to specify a `BlackScholes` model for the `ConvertibleBond` instrument object.
3. Use `finpricer` to specify a `FiniteDifference` pricing method for one or more `ConvertibleBond` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `ConvertibleBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
ConvertibleBondObj = fininstrument(InstrumentType,'
CouponRate',couponrate_value,'Maturity',maturity_date,'
ConversionRatio',conversion_ratio_value)
ConvertibleBondObj = fininstrument( ___ ,Name,Value)
```

Description

`ConvertibleBondObj = fininstrument(InstrumentType,'CouponRate',couponrate_value,'Maturity',maturity_date,'ConversionRatio',conversion_ratio_value)` creates a `ConvertibleBond` object for one of more Convertible Bond instruments by specifying `InstrumentType` and sets the properties on page 11-2558 for the required name-value pair arguments `CouponRate`, `Maturity`, and `ConversionRatio`.

`ConvertibleBondObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2558 using name-value pair arguments in addition to the required arguments in the previous syntax. For example, `ConvertibleBondObj = fininstrument("ConvertibleBond",'CouponRate',CouponRate,'Maturity',Maturity,'ConversionRatio',ConvRatio,'Period',Period,'Spread',Spread,'CallSchedule',CallSchedule,'CallExerciseStyle',"american")` creates an `ConvertibleBond` instrument with an American exercise and a call schedule. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"ConvertibleBond"` | string array with values of `"ConvertibleBond"` | character vector with value `'ConvertibleBond'` | cell array of character vectors with values of `'ConvertibleBond'`

Instrument type, specified as a string with the value of `"ConvertibleBond"`, a character vector with the value of `'ConvertibleBond'`, an `NINST`-by-`1` string array with values of `"ConvertibleBond"`, or an `NINST`-by-`1` cell array of character vectors with values of `'ConvertibleBond'`.

Data Types: `char` | `cell` | `string`

`ConvertibleBond` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ConvertibleBondObj = fininstrument("ConvertibleBond",'CouponRate',CouponRate, 'Maturity',Maturity,'ConversionRatio',ConvRatio,'Period',Period,'Spread',Spread,'CallSchedule',CallSchedule,'CallExerciseStyle',"american")`

Required `ConvertibleBond` Name-Value Pair Arguments

`CouponRate` — Coupon rate for `ConvertibleBond` object

scalar decimal | decimal vector | timetable

Coupon rate for the `ConvertibleBond` object, specified as the comma-separated pair consisting of `'CouponRate'` as a scalar decimal or an `NINST`-by-`1` vector of decimals for an annual rate or a timetable where the first column is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Note If you are creating one or more `ConvertibleBond` instruments and use a timetable, the timetable specification applies to all of the `ConvertibleBond` instruments. `CouponRate` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `double` | `timetable`

`Maturity` — Maturity date for `ConvertibleBond` object

datetime array | string array | date character vector

Maturity date for the `ConvertibleBond` object, specified as the comma-separated pair consisting of `'Maturity'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `ConvertibleBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`ConversionRatio` — Number of shares convertible from one bond
scalar numeric | numeric vector | timetable

Number of shares convertible from one bond, specified as the comma-separated pair consisting of `'ConversionRatio'` and a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable where the first column is dates and the second column is associated ratios. The date in the first column indicates the last day that the conversion ratio is valid.

Note If you are creating one or more `ConvertibleBond` instruments and use a timetable, the timetable specification applies to all of the `ConvertibleBond` instruments. `ConversionRatio` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `double` | `timetable`

Optional ConvertibleBond Name-Value Pair Arguments

`Spread` — Number of basis points over the reference rate
`0` (default) | scalar numeric | numeric vector

Number of basis points over the reference rate, specified as the comma-separated pair consisting of `'Spread'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`CallSchedule` — Call schedule
`[ ]` (default) | timetable

Call schedule, specified as the comma-separated pair consisting of `'CallSchedule'` and a timetable of call dates and strikes.

If you use a date character vector or date string for the dates in this timetable, the format must be recognizable by `datetime` because the `CallSchedule` property is stored as a datetime.

Note For the `ConvertibleBond` instrument, you can use a `CallSchedule` with a `CallExerciseStyle` and a `PutSchedule` with a `PutExerciseStyle` simultaneously.

Data Types: `timetable`

`CallExerciseStyle` — Call option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with values of `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'` | cell array of character vectors with values of `'European'`, `'American'`, or `'Bermudan'`

Call option exercise style, specified as the comma-separated pair consisting of `'CallExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `string` | `char` | `cell`

`PutSchedule` — Put schedule
`[ ]` (default) | timetable

Put schedule, specified as the comma-separated pair consisting of `'PutSchedule'` and a timetable of call dates and strikes.

If you use a date character vector or date string for dates in this timetable, the format must be recognizable by `datetime` because the `PutSchedule` property is stored as a datetime.

Note For he `ConvertibleBond` instrument, you can use a `CallSchedule` with a `CallExerciseStyle` and a `PutSchedule` with a `PutExerciseStyle` simultaneously.

Data Types: `timetable`

`PutExerciseStyle` — Put option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with values of `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'` | cell array of character vectors with values of `'European'`, `'American'`, or `'Bermudan'`

Put option exercise style, specified as the comma-separated pair consisting of `'PutExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `string` | `cell` | `char`

`Period` — Frequency of payments per year
`2` (default) | integer | vector of integers

Frequency of payments per year, specified as the comma-separated pair consisting of `'Period'` and a scalar integer or an `NINST`-by-`1` vector of integers. Possible values for `Period` are `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and scalar integer or an `NINST`-by-`1` vector of integers using the following values:

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Notional principal amount or principal value schedule, specified as the comma-separated pair consisting of `'Principal'` and a scalar numeric or `NINST`-by-`1` numeric vector or a timetable.

`Principal` accepts a `timetable`, where the first column is dates and the second column is the associated notional principal value. The date indicates the last day that the principal value is valid.

Data Types: `double` | `timetable`

`DaycountAdjustedCashFlow` — Flag indicating whether cash flow adjusts for day count convention
`false` (default) | value of `true` or `false`

Flag indicating whether cash flow adjusts for the day count convention, specified as the comma-separated pair consisting of `'DaycountAdjustedCashFlow'` and a scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Possible values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and dates using an `NINST`-by-`1` vector of a datetime array, string array, or date character vectors. An example follows.

```
H = holidays(datetime('today'),datetime(2025,12,15));
ConvertibleBondObj = fininstrument("ConvertibleBond",'CouponRate',0.34,'Maturity',datetime(2025,1
'ConversionRatio',ConvRatio,'CallSchedule',schedule,'CallExerciseStyle',"american",'Holidays',H)
```

To support existing code, `ConvertibleBond` also accepts serial date numbers as inputs, but they are not recommended.

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`true` (in effect) (default) | value of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar logical value or an `NINST`-by-`1` vector of logical values of `true` or `false`.

- If you set `EndMonthRule` to `false`, the software ignores the rule, meaning that a payment date is always the same numerical day of the month.
- If you set `EndMonthRule` to `true`, the software sets the rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
`NaT` (default) | datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `ConvertibleBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `IssueDate` property is stored as a datetime.

`FirstCouponDate` — Irregular first coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `ConvertibleBond` also accepts serial date numbers as inputs, but they are not recommended.

When you specify both `FirstCouponDate` and `LastCouponDate`, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `FirstCouponDate` property is stored as a datetime.

`LastCouponDate` — Irregular last coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or an `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `ConvertibleBond` also accepts serial date numbers as inputs, but they are not recommended.

If you specify `LastCouponDate` but not `FirstCouponDate`, `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify `LastCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `LastCouponDate` property is stored as a datetime.

`StartDate` — Forward starting date of payments
`NaT` (default) | datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or an `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `ConvertibleBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `StartDate` property is stored as a datetime.

`Name` — User-defined name for instrument
`"  "` (default) | string | character vector

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-1 cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`CouponRate` — Coupon annual rate
scalar decimal | decimal vector | timetable

Coupon annual rate, returned as a scalar decimal or an `NINST`-by-1 or timetable.

Data Types: `double` | `timetable`

`Maturity` — Maturity date
datetime | vector of datetimes

Maturity date, returned as a scalar datetime or an `NINST`-by-1 vector of datetimes.

Data Types: `datetime`

`ConversionRatio` — Number of shares convertible from one bond
scalar numeric | numeric vector | timetable

Number of shares convertible from one bond, returned as a scalar numeric or an `NINST`-by-`1` numeric vector an a timetable.

Data Types: `double` | `timetable`

`Spread` — Number of basis points over the reference rate
scalar numeric | numeric vector

Number of basis points over the reference rate, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`CallSchedule` — Call schedule
timetable

Call schedule, returned as a timetable.

Data Types: `timetable`

`PutSchedule` — Put schedule
timetable

Put schedule, returned as a timetable.

Data Types: `timetable`

`Period` — Coupons per year
`2` (default) | integer | vector of integers

Coupons per year, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Principal` — Notional principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Notional principal amount or principal value schedule, returned as a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

Data Types: `timetable` | `double`

`DaycountAdjustedCashFlow` — Flag indicating whether cash flow adjusts for day count convention
`false` (default) | value of `true` or `false`

Flag indicating whether cash flow adjusted for day count convention, returned as scalar logical or an NINST-by-1 vector of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array

Business day conventions, returned as a scalar string or an NINST-by-1 string array.

Data Types: `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime | vector of datetimes

Holidays used in computing business days, returned as datetimes or an NINST-by-1 vector of datetimes.

Data Types: `datetime`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`true` (in effect) (default) | value of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, returned as a scalar logical or an NINST-by-1 vector of logicals.

Data Types: `logical`

`IssueDate` — Bond issue date
`NaT` (default) | datetime | vector of datetimes

Bond issue date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: `datetime`

`FirstCouponDate` — Irregular first coupon date
`NaT` (default) | datetime | vector of datetimes

Irregular first coupon date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: `datetime`

`LastCouponDate` — Irregular last coupon date
`NaT` (default) | datetime | vector of datetimes

Irregular last coupon date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: `datetime`

`StartDate` — Forward starting date of payments
`NaT` (default) | datetime | vector of datetimes

Forward starting date of payments, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: `datetime`

`CallExerciseStyle` — Call option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermuda"` | string array with values of `"European"`, `"American"`, or `"Bermuda"`

This property is read-only.

Call option exercise style, returned as a scalar string or `NINST`-by-`1` string array with values of `"European"`, `"American"`, or `"Bermuda"`.

Data Types: `string`

`PutExerciseStyle` — Put option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermuda"` | string array with values of `"European"`, `"American"`, or `"Bermuda"`

This property is read-only.

Put option exercise style, returned as a scalar string or an `NINST`-by-`1` string array with values of `"European"`, `"American"`, or `"Bermuda"`.

Data Types: `string`

`Name` — User-defined name for instrument
`"  "` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

setCallExercisePolicy	Set call exercise policy for OptionEmbeddedFixedBond, OptionEmbeddedFloatBond, or ConvertibleBond instrument
setPutExercisePolicy	Set put exercise policy for OptionEmbeddedFixedBond, OptionEmbeddedFloatBond, or ConvertibleBond instrument

Examples

Price Convertible Bond Instrument Using Black-Scholes Model and Finite Difference Pricer

This example shows the workflow to price a `ConvertibleBond` instrument when you use a `BlackScholes` model and a `FiniteDifference` pricing method.

Create `ConvertibleBond` Instrument Object

Use `fininstrument` to create a `ConvertibleBond` instrument object.

```
CouponRate = 0;
Maturity = datetime(2014,10,1);
ConvRatio = 2;
Period = 1;
Spread = 0.05;

CallExDates = datetime(2014,10,1);
CallStrike = 115;
CallSchedule = timetable(CallExDates, CallStrike);
```

```
ConvBond = fininstrument("ConvertibleBond",'CouponRate',CouponRate,'Maturity',Maturity,'Conversio
```

```
ConvBond =
  ConvertibleBond with properties:

                CouponRate: 0
           ConversionRatio: 2
                    Spread: 0.0500
                    Period: 1
                     Basis: 0
              EndMonthRule: 1
                 Principal: 100
  DaycountAdjustedCashFlow: 0
     BusinessDayConvention: "actual"
                  Holidays: NaT
                  Maturity: 01-Oct-2014
                 IssueDate: NaT
           FirstCouponDate: NaT
            LastCouponDate: NaT
                 StartDate: NaT
              CallSchedule: [1x1 timetable]
               PutSchedule: [0x0 timetable]
         CallExerciseStyle: "american"
          PutExerciseStyle: [0x0 string]
                      Name: "Convertible_Bond"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
AssetPrice = 50;
Volatility = 0.3;

BSModel = finmodel("BlackScholes",'Volatility',Volatility)
```

```
BSModel =
  BlackScholes with properties:

    Volatility: 0.3000
   Correlation: 1
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
StartDate = datetime(2014,1,1);
EndDate = datetime(2015,1,1);
Rate = 0.1;

ZeroCurve = ratecurve('zero',StartDate,EndDate,Rate,'Compounding',-1,'Basis',1)
```

```
ZeroCurve =
  ratecurve with properties:

            Type: "zero"
     Compounding: -1
```

```
                Basis: 1
                Dates: 01-Jan-2015
                Rates: 0.1000
               Settle: 01-Jan-2014
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create FiniteDifference Pricer Object

Use `finpricer` to create a `FiniteDifference` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("FiniteDifference",'Model',BSModel,'SpotPrice',AssetPrice,'DiscountCurve',Z
```

```
outPricer =
  FiniteDifference with properties:

     DiscountCurve: [1x1 ratecurve]
             Model: [1x1 finmodel.BlackScholes]
         SpotPrice: 50
    GridProperties: [1x1 struct]
      DividendType: "continuous"
     DividendValue: 0
```

Price ConvertibleBond Instrument

Use `price` to compute the price and sensitivities for the `ConvertibleBond` instrument.

```
[Price, outPR] = price(outPricer,ConvBond,"all")
```

```
Price = 104.3812

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta      Gamma     Lambda      Theta       Rho       Vega
    ______    ______    _______    _______    _______    _______    ______

    104.38    1.3012    0.04195    0.62329    0.72984    -21.883    17.947
```

Price Multiple Convertible Bond Instruments Using Black-Scholes Model and Finite Difference Pricer

This example shows the workflow to price multiple `ConvertibleBond` instruments when you use a `BlackScholes` model and a `FiniteDifference` pricing method.

Create ConvertibleBond Instrument Object

Use `fininstrument` to create a `ConvertibleBond` instrument object for three Convertible Bond instruments.

```
ConvRatio = 2;
Period = 1;
Spread = 0.05;

CallExDates = datetime(2014,10,1);
CallStrike = 115;
CallSchedule = timetable(CallExDates, CallStrike);

ConvBond = fininstrument("ConvertibleBond",'CouponRate',[0 ; 0.1 ; 0.2],'Maturity',datetime([201
```

```
ConvBond=3×1 object
  3x1 ConvertibleBond array with properties:

    CouponRate
    ConversionRatio
    Spread
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    Maturity
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    CallSchedule
    PutSchedule
    CallExerciseStyle
    PutExerciseStyle
    Name
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
AssetPrice = 50;
Volatility = 0.3;

BSModel = finmodel("BlackScholes",'Volatility',Volatility)
```

```
BSModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
StartDate = datetime(2014,1,1);
EndDate = datetime(2015,1,1);
Rate = 0.1;

ZeroCurve = ratecurve('zero',StartDate,EndDate,Rate,'Compounding',-1,'Basis',1)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2015
               Rates: 0.1000
              Settle: 01-Jan-2014
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `FiniteDifference` Pricer Object

Use `finpricer` to create a `FiniteDifference` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("FiniteDifference",'Model',BSModel,'SpotPrice',AssetPrice,'DiscountCurve',Z
```

```
outPricer =
  FiniteDifference with properties:

     DiscountCurve: [1x1 ratecurve]
             Model: [1x1 finmodel.BlackScholes]
         SpotPrice: 50
    GridProperties: [1x1 struct]
      DividendType: "continuous"
     DividendValue: 0
```

Price `ConvertibleBond` Instruments

Use `price` to compute the prices and sensitivities for the `ConvertibleBond` instruments.

```
[Price, outPR] = price(outPricer,ConvBond,"all")
```

```
Price = 3×1

  104.3812
  198.3288
  298.3014


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta      Gamma      Lambda      Theta       Rho       Vega
    ______    _______    ________   ________    ________    _______    ______

    104.38    1.3012     0.04195    0.62329     0.72984     -21.883    17.947

ans=1×7 table
    Price     Delta      Gamma        Lambda     Theta       Rho           Vega
    ______    ______    ___________   _______    _______    ___________    ___________

    198.33      4       1.7053e-13    1.0084     300.82     2.8422e-10     2.8422e-10

ans=1×7 table
    Price     Delta       Gamma        Lambda     Theta         Rho           Vega
    ______    ______    ___________    _______    _______    ____________    ____________

    298.3       6       -1.7053e-13    1.0057     277.96     -1.7053e-09     -5.6843e-10
```

More About

Convertible Bond

A convertible bond is a financial instrument that combines equity and debt features.

A convertible bond is a bond with the embedded option to turn it into a fixed number of shares. The holder of a convertible bond has the right, but not the obligation, to exchange the convertible security for a predetermined number of equity shares at a preset price. The debt component is derived from the coupon payments and the principal. The equity component is provided by the conversion feature.

Convertible bonds have several defining features:

- Coupon — Coupons in convertible bonds are typically lower than coupons in vanilla bonds since investors are willing to take the lower coupon for the opportunity to participate in the company stock via the conversion.
- Maturity — Most convertible bonds are issued with long-stated maturities. Short-term maturity convertible bonds usually do not have call or put provisions.
- Conversion ratio — Conversion ratio is the number of shares that the holder of the convertible bond receives from exercising the call option of the convertible bond. The conversion ratio is the par value of the convertible bond divided by the conversion price of equity.

  For example, a conversion ratio of 25 means a bond can be exchanged for 25 shares of stock. This also implies a conversion price of $40 (1000/25). This, $40, is the price at which the owner would buy the shares. This can be expressed as a ratio or as the conversion price and is specified in the contract along with other provisions.
- Option type:
  - Callable convertible: Convertible bond that is callable by the issuer. The issuer of the bond forces conversion, removing the advantage that conversion is at the discretion of the bondholder. Upon call, the bondholder can either convert the bond or redeem the bond at the call price. This option enables the issuer to control the price of the convertible bond and if necessary refinance the debt with a new, cheaper bond.

- Puttable convertible: Convertible bond with a put feature that allows the bondholder to sell back the bond at a premium on a specific date. This option protects the holder against rising interest rates by reducing the year to maturity.

Tips

After creating a `ConvertibleBond` object, you can modify the `CallSchedule` and `CallExerciseStyle` using `setCallExercisePolicy`. You can modify the `PutSchedule` and `PutExerciseStyle` values using `setPutExercisePolicy`.

Version History

Introduced in R2021a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `ConvertibleBond` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`finmodel` | `finpricer` | `timetable`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

InflationBond

`InflationBond` instrument object

Description

Create and price an `InflationBond` instrument object for one or more Inflation Bond instruments using this workflow:

1. Use `fininstrument` to create an `InflationBond` instrument object for one or more Inflation Bond instruments.
2. Use `ratecurve` to specify an interest-rate model for the `InflationBond` instrument object.
3. Use `inflationcurve` to specify an inflation curve model for the `InflationBond` instrument object.
4. Use `finpricer` to specify an `Inflation` pricing method for one or more `InflationBond` instruments.
5. Use `inflationCashflows` to compute cash flows for each one of the `InflationBond` instruments.

For more detailed information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for an `InflationBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
InflationBond = fininstrument(InstrumentType,'CouponRate',couponrate_value,'
Maturity',maturity_date)
InflationBond = fininstrument( ___ ,Name,Value)
```

Description

`InflationBond = fininstrument(InstrumentType,'CouponRate',couponrate_value,'Maturity',maturity_date)` creates an `InflationBond` object for one or more Inflation Bond instruments by specifying `InstrumentType` and sets the properties on page 11-2573 for the required name-value pair arguments `CouponRate` and `Maturity`.

`InflationBond = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2573 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `InflationBond = fininstrument("InflationBond",'Maturity',Maturity,'CouponRate',CouponRate,'IssueDate',IssueDate)` creates a `InflationBond` option.

Input Arguments

`InstrumentType` — Instrument type

string with value `"InflationBond"` | string array with values of `"InflationBond"` | character vector with value `'InflationBond'` | cell array of character vectors with values of `'InflationBond'`

Instrument type, specified as a string with the value of `"InflationBond"`, a character vector with the value of `'InflationBond'`, an `NINST`-by-`1` string array with values of `"InflationBond"`, or an `NINST`-by-`1` cell array of character vectors with values of `'InflationBond'`.

Data Types: `char` | `cell` | `string`

`InflationBond` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `InflationBond = fininstrument("InflationBond",'Maturity',Maturity,'CouponRate',CouponRate,'IssueDate',IssueDate)`

Required InflationBond Name-Value Pair Arguments

`CouponRate` — `InflationBond` coupon rate

scalar decimal | vector of decimals

`InflationBond` coupon rate, specified as the comma-separated pair consisting of `'CouponRate'` and a scalar decimal or an `NINST`-by-`1` vector of decimals for an annual rate.

Data Types: `double`

`Maturity` — `InflationBond` maturity date

datetime array | string array | date character vector

`InflationBond` maturity date, specified as the comma-separated pair consisting of `'Maturity'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `InflationBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

Optional InflationBond Name-Value Pair Arguments

`Period` — Frequency of payments per year

`2` (default) | scalar numeric with value of `0`, `1`, `2`, `3`, `4`, `6`, or `12` | numeric vector with values of `0`, `1`, `2`, `3`, `4`, `6`, or `12`

Frequency of payments, specified as the comma-separated pair consisting of `'Period'` and a scalar integer or an `NINST`-by-`1` vector of integers. Values for `Period` are `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

Basis — Day count basis
0 (actual/actual) (default) | scalar integer from 0 to 13 | vector of integers from 0 to 13

Day count basis, specified as the comma-separated pair consisting of 'Basis' and scalar integer or an NINST-by-1 vector of integers using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: double

Principal — Initial principal amount
100 (default) | scalar numeric | numeric vector

Initial principal amount, specified as the comma-separated pair consisting of 'Principal' and a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

DaycountAdjustedCashFlow — Flag indicating whether cash flow adjusts for day count convention
false (default) | scalar logical value of true or false | vector of logical values of true or false

Flag indicating whether cash flow is adjusted by day count convention, specified as the comma-separated pair consisting of 'DaycountAdjustedCashFlow' and a scalar logical or an NINST-by-1 vector of logicals with values of true or false.

Data Types: logical

IssueDate — Bond issue date
NaT (default) | datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of 'IssueDate' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, InflationBond also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `IssueDate` property is stored as a datetime.

`Lag` — Indexation lag in months
3 (default) | scalar numeric | numeric vector

Indexation lag in months, specified as the comma-separated pair consisting of `'Lag'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`BusinessDayConvention` — Business day conventions for cash flow dates
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day conventions for cash flow dates, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and dates using an `NINST`-by-`1` vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
InflationBondObj = fininstrument("InflationBond",'CouponRate',0.34,'Maturity',datetime(2025,12,15
```

To support existing code, `InflationBond` also accepts serial date numbers as inputs, but they are not recommended.

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`true` (in effect) (default) | scalar logical values of `true` or `false` | vector of logical values with `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar logical value or an `NINST`-by-`1` vector of logical values of `true` or `false`.

- If you set `EndMonthRule` to `false`, the software ignores the rule, meaning that a payment date is always the same numerical day of the month.
- If you set `EndMonthRule` to `true`, the software sets the rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`FirstCouponDate` — Irregular first coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `InflationBond` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `FirstCouponDate` property is stored as a datetime.

`LastCouponDate` — Irregular last coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `InflationBond` also accepts serial date numbers as inputs, but they are not recommended.

If you specify `LastCouponDate` but not `FirstCouponDate`, `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify `LastCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `LastCouponDate` property is stored as a datetime.

`Name` — User-defined name for instrument
`''` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`CouponRate` — `InflationBond` coupon annual rate
scalar decimal | vector of decimals

`InflationBond` coupon annual rate, returned as a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Maturity` — `InflationBond` maturity date
datetime | vector of datetimes

`InflationBond` maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Period` — Frequency of payments per year
`2` (default) | scalar integer | vector of integers

Frequency of payments per year, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Principal` — Initial principal amount
`100` (default) | scalar numeric | numeric vector

Initial principal amount, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`DaycountAdjustedCashFlow` — Flag indicating whether cash flow adjusts for day count convention
`false` (default) | scalar logical value of `true` or `false` | vector of logical values with `true` or `false`

Flag indicating whether cash flow adjusts for day count convention, returned as scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

Data Types: `logical`

`IssueDate` — Bond issue date
`NaT` (default) | datetime | vector of datetimes

Bond issue date, returned as a datetime or an `NINST`-by-`1` datetime vector.

Data Types: `datetime`

`Lag` — Indexation lag in months
`3` (default) | scalar numeric | numeric vector

Indexation lag in months, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array

Business day conventions, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`true` (in effect) (default) | scalar logical value of `true` or `false` | vector of logicals with value of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, returned as a scalar logical or an `NINST`-by-`1` vector of logicals.

Data Types: `logical`

`FirstCouponDate` — Irregular first coupon date
`NaT` (default) | datetime | vector of datetimes

Irregular first coupon date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`LastCouponDate` — Irregular last coupon date
`NaT` (default) | datetime | vector of datetimes

Irregular last coupon date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

inflationCashflows Compute cash flows for InflationBond instrument

Examples

Price Inflation Bond Instrument Using `inflationcurve` and Inflation Pricer

This example shows the workflow to price an `InflationBond` instrument when you use an `inflationcurve` and an `Inflation` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2021
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2020,10,1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
    InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create `InflationBond` Instrument Object

Use `fininstrument` to create an `InflationBond` instrument object.

```
IssueDate = datetime(2021,1,1);
Maturity = datetime(2026,1,1);
CouponRate = 0.02;

InflationBond = fininstrument("InflationBond", 'IssueDate', IssueDate, 'Maturity', Maturity, 'Cou
```

```
InflationBond =
  InflationBond with properties:

                 CouponRate: 0.0200
```

```
                    Period: 2
                     Basis: 0
                 Principal: 100
    DaycountAdjustedCashFlow: 0
                       Lag: 3
       BusinessDayConvention: "actual"
                   Holidays: NaT
              EndMonthRule: 1
                 IssueDate: 01-Jan-2021
           FirstCouponDate: NaT
            LastCouponDate: NaT
                  Maturity: 01-Jan-2026
                      Name: "inflation_bond_instrument"
```

Create `Inflation` Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price `InflationBond` Instrument

Use `price` to compute the price and sensitivities for the `InflationBond` instrument.

```
[Price, outPR] = price(outPricer, InflationBond)
```

```
Price = 112.1856

outPR =
  priceresult with properties:

       Results: [1x1 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=table
    Price
    ______

    112.19
```

Price Multiple Inflation Bond Instruments Using `inflationcurve` and Inflation Pricer

This example shows the workflow to price multiple `InflationBond` instruments when you use an `inflationcurve` and an `Inflation` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2019,8,1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
     InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create `InflationBond` Instrument Object

Use `fininstrument` to create an `InflationBond` instrument object for three Inflation Bond instruments.

```
IssueDate = datetime([2020,1,1 ; 2019,12,1 ; 2019,11,1]);
Maturity = datetime([2026,1,1 ; 2026,2,1 ; 2026,3,1]);
CouponRate = 0.02;
```

```
InflationBond = fininstrument("InflationBond", 'IssueDate', IssueDate, 'Maturity', Maturity, 'Cou

InflationBond=3×1 object
  3x1 InflationBond array with properties:

    CouponRate
    Period
    Basis
    Principal
    DaycountAdjustedCashFlow
    Lag
    BusinessDayConvention
    Holidays
    EndMonthRule
    IssueDate
    FirstCouponDate
    LastCouponDate
    Maturity
    Name
```

Create `Inflation` Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)

outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price `InflationBond` Instruments

Use `price` to compute the prices and sensitivities for the `InflationBond` instruments.

```
[Price, outPR] = price(outPricer, InflationBond)

Price = 3×1

  112.8769
  113.1022
  113.3434


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData


outPR.Results
```

```
ans=table
    Price
    ______

    112.88


ans=table
    Price
    _____

    113.1


ans=table
    Price
    ______

    113.34
```

More About

Inflation-Indexed Bond

An inflation-indexed bond is a security that guarantees a return higher than the rate of inflation if it is held to maturity. Inflation-indexed securities link their capital appreciation, or coupon payments, to inflation rates

Algorithms

To price an inflation-indexed bond, use an inflation curve and a nominal discount curve (model-free approach), where the cash flows are discounted using the nominal discount curve.

$$I(0,T)P_n(0,T) = I(0)P_r(0,T)$$

$$B_{TIPS}(0,T_M) = \frac{1}{I(T_0)}\sum_{i=1}^{M} cI(0)P_r(0,T_i) + FI(0)P_r(0,T_M)$$

$$= \frac{1}{I(T_0)}\sum_{i=1}^{M} cI(0,T_i)P_n(0,T_i) + FI(0,T_M)P_n(0,T_M)$$

where

- P_n is the nominal zero-coupon bond price.
- P_r is the real zero-coupon bond price.
- k is the fixed inflation rate.
- $I(0,T)$ is the breakeven inflation index for period $(0,T)$.
- $I(0)$ is the inflation index at $(t = 0)$.
- $I(T_0)$ is the base inflation index at the issue date $(t = T_0)$.
- $B_{TIPS}(0,T_M)$ is the inflation-indexed bond price.
- $I(T_{i-1})$ is the inflation index at the start date with some lag (for example, three months).

- *C* is the coupon.
- *F* is the face value.

Version History

Introduced in R2021a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `InflationBond` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Brody, D. C., Crosby, J., and Li, H. "Convexity Adjustments in Inflation-Linked Derivatives." *Risk Magazine*. November 2008, pp. 124–129.

[2] Kerkhof, J. "Inflation Derivatives Explained: Markets, Products, and Pricing." *Fixed Income Quantitative Research*, Lehman Brothers, July 2005.

[3] Treasury Inflation-Protected Securities (TIPS) at: https://www.treasurydirect.gov/instit/marketables/tips/tips.htm.

[4] Zhang, J. X. "Limited Price Indexation (LPI) Swap Valuation Ideas." *Wilmott Magazine*. no. 57, January 2012, pp. 58–69.

See Also

Functions

`YearYearInflationSwap` | `ZeroCouponInflationSwap` | `finpricer`

Topics

“Analyze Inflation-Indexed Instruments” on page 2-132
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

YearYearInflationSwap

`YearYearInflationSwap` instrument object

Description

Create and price a `YearYearInflationSwap` instrument object for one or more Year-on-Year Inflation-Indexed Swap instruments using this workflow:

1. Use `fininstrument` to create a `YearYearInflationSwap` instrument object for one or more Year-on-Year Inflation-Indexed Swap instruments.
2. Use `ratecurve` to specify an interest-rate model for the `YearYearInflationSwap` instrument object.
3. Use `inflationcurve` to specify an inflation curve model.
4. Use `finpricer` to specify an `Inflation` pricing method for one or more `YearYearInflationSwap` instruments.
5. Use `inflationCashflows` to compute cash flows for each one of the `YearYearInflationSwap` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `YearYearInflationSwap` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
YYInflationSwap = fininstrument(InstrumentType,'Maturity',maturity_date,'
Notional',notional_value,'FixedInflationRate',inflation_rate)
YYInflationSwap = fininstrument( ___ ,Name,Value)
```

Description

`YYInflationSwap = fininstrument(InstrumentType,'Maturity',maturity_date,'Notional',notional_value,'FixedInflationRate',inflation_rate)` creates a `YearYearInflationSwap` object for one or more Year-on-Year Inflation-Indexed Swap instruments by specifying `InstrumentType` and sets the properties on page 11-2583 for the required name-value pair arguments `Maturity`, `Notional`, and `FixedInflationRate`.

`YYInflationSwap = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2583 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `YYInflationSwap = fininstrument("YearYearInflationSwap",'Maturity',Maturity,'FixedInflationRate',FixedInflationRate,'Notional',Notional,'Basis',4,'Lag',4)` creates a `YearYearInflationSwap` option. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"YearYearInflationSwap"` | string array with values of `"YearYearInflationSwap"` | character vector with value `'YearYearInflationSwap'` | cell array of character vectors with values of `'YearYearInflationSwap'`

Instrument type, specified as a string with the value of `"YearYearInflationSwap"`, a character vector with the value of `'YearYearInflationSwap'`, an `NINST`-by-`1` string array with values of `"YearYearInflationSwap"`, or an `NINST`-by-`1` cell array of character vectors with values of `'YearYearInflationSwap'`.

Data Types: `char` | `cell` | `string`

`YearYearInflationSwap` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `YYInflationSwap = fininstrument("YearYearInflationSwap",'Maturity',Maturity,'FixedInflationRate',FixedInflationRate,'Notional',Notional,'Basis',4,'Lag',4)`

Required YearYearInflationSwap Name-Value Pair Arguments

`Maturity` — Swap maturity date

datetime array | string array | date character vector

Swap maturity date, specified as the comma-separated pair consisting of `'Maturity'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `YearYearInflationSwap` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`Notional` — Notional amount

scalar numeric | numeric decimal

Notional amount, specified as the comma-separated pair consisting of `'Notional'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`FixedInflationRate` — Inflation rate

scalar decimal | vector of decimals

Inflation rate, specified as the comma-separated pair consisting of `'FixedInflationRate'` and a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

Optional YearYearInflationSwap Name-Value Pair Arguments

`Basis` — Day count basis for the fixed leg
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis for the fixed leg, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer or an `NINST`-by-`1` vector of integers for the following:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Lag` — Indexation lag in months
`3` (default) | scalar numeric | numeric vector

Indexation lag in months, specified as the comma-separated pair consisting of `'Lag'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | character vector

User-defined name for the instrument, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Maturity` — Swap maturity date
scalar datetime | vector of datetimes

Swap maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Notional` — Notional amount
scalar numeric | numeric vector

Notional amount, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`FixedInflationRate` — Inflation rate
scalar decimal | vector of decimals

Inflation rate, returned as a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Basis` — Day count basis for fixed leg
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis for fixed leg, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Lag` — Indexation lag in months
`3` (default) | scalar numeric | numeric vector

Indexation lag in months, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | scalar string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

inflationCashflows Compute cash flows for YearYearInflationSwap instrument

Examples

Price Year-on-Year Inflation-Indexed Swap Instrument Using `inflationcurve` and Inflation Pricer

This example shows the workflow to price a `YearYearInflationSwap` instrument when you use an `inflationcurve` object and an `Inflation` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
```

```
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)

ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2021
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2020,10,1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)

myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
     InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create `YearYearInflationSwap` Instrument Object

Use `fininstrument` to create a `YearYearInflationSwap` instrument object.

```
Maturity = datetime(2025,1,1);
FixedInflationRate = 0.015;
Notional = 2000;

YYInflationSwap = fininstrument("YearYearInflationSwap",'Maturity',Maturity,'FixedInflationRate'

YYInflationSwap =
  YearYearInflationSwap with properties:

             Notional: 2000
    FixedInflationRate: 0.0150
                Basis: 0
                  Lag: 3
             Maturity: 01-Jan-2025
                 Name: "YYInflationSwap_instrument"
```

Create Inflation Pricer Object

Use finpricer to create an Inflation pricer object and use the ratecurve object with the 'DiscountCurve' name-value pair argument and the inflationcurve object with the 'InflationCurve' name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price YearYearInflationSwap Instrument

Use price to compute the price and sensitivities for the YearYearInflationSwap instrument.

```
[Price,outPR] = price(outPricer,YYInflationSwap,"all")
```

```
Price = 12.5035
```

```
outPR =
  priceresult with properties:

       Results: [1x1 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=table
    Price
    ______

    12.504
```

Price Multiple Year-on-Year Inflation-Indexed Swap Instruments Using inflationcurve and Inflation Pricer

This example shows the workflow to price multiple YearYearInflationSwap instruments when you use an inflationcurve object and an Inflation pricing method.

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:
```

```
                Type: "zero"
        Compounding: -1
              Basis: 0
              Dates: [10x1 datetime]
              Rates: [10x1 double]
             Settle: 15-Jan-2021
       InterpMethod: "linear"
  ShortExtrapMethod: "next"
   LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2019,10,1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
    InflationIndexValues: [10x1 double]
   ForwardInflationRates: [9x1 double]
             Seasonality: [12x1 double]
```

Create `YearYearInflationSwap` Instrument Object

Use `fininstrument` to create a `YearYearInflationSwap` instrument object for three Year-on-Year Inflation-Indexed Swap instruments.

```
Maturity = datetime([2024,1,1 ; 2024,11,1 ; 2024,12,1]);
FixedInflationRate = 0.015;
Notional = [20000 ; 30000 ; 40000];

YYInflationSwap = fininstrument("YearYearInflationSwap",'Maturity',Maturity,'FixedInflationRate'
```

```
YYInflationSwap=3×1 object
  3x1 YearYearInflationSwap array with properties:

    Notional
    FixedInflationRate
    Basis
    Lag
    Maturity
    Name
```

Create `Inflation` Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price YearYearInflationSwap Instruments

Use `price` to compute the prices and sensitivities for the `YearYearInflationSwap` instruments.

```
[Price,outPR] = price(outPricer,YYInflationSwap,"all")
```

```
Price = 3×1

   26.0701
   18.1540
    1.3201


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=table
    Price
    _____

    26.07


ans=table
    Price
    ______

    18.154


ans=table
    Price
    ______

    1.3201
```

More About

Year-on-Year Inflation-Indexed Swap

A year-on-year inflation-indexed swap is a financial contract where, at the end of each accrual period, one party (the inflation receiver) pays a fixed-rate coupon and receives a floating payment linked to a specific inflation index from the other party (the inflation payer).

Algorithms

To price a year-on-year inflation-indexed swap (YYIIS), use an inflation curve and a nominal discount curve (model-free approach), where the cash flows are discounted using the nominal discount curve.

Cash flows for each year $t = \{T_1,...,T_i,...,T_M\}$:

$$FixedLeg = N \times k \times \Delta t_{fixed}$$

$$InflationLeg = N \times \left[\frac{I(T_i)}{I(T_i - 1)} - 1\right] \times \Delta t_{inflation}$$

where

- N is the reference notional of the swap.
- k is the fixed inflation rate.
- Δt_{fixed} is the fixed leg fraction for the period.
- $\Delta t_{inflation}$ is the inflation leg fraction for the period.
- $I(T_i)$ is the inflation index at the period end date with some lag (for example, three months).
- $I(T_{i\text{-}1})$ is the inflation index at the start date with some lag (for example, three months).

Version History

Introduced in R2021a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `YearYearInflationSwap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Brody, D. C., Crosby, J., and Li, H. "Convexity Adjustments in Inflation-Linked Derivatives." *Risk Magazine*. November 2008, pp. 124–129.

[2] Kerkhof, J. "Inflation Derivatives Explained: Markets, Products, and Pricing." *Fixed Income Quantitative Research*, Lehman Brothers, July 2005.

[3] Zhang, J. X. "Limited Price Indexation (LPI) Swap Valuation Ideas." *Wilmott Magazine*. no. 57, January 2012, pp. 58–69.

See Also

Functions
`InflationBond` | `ZeroCouponInflationSwap` | `finpricer`

Topics
“Analyze Inflation-Indexed Instruments” on page 2-132
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

ZeroCouponInflationSwap

`ZeroCouponInflationSwap` instrument object

Description

Create and price a `ZeroCouponInflationSwap` instrument object for one or more Zero-Coupon Inflation Swap instruments using this workflow:

1. Use `fininstrument` to create a `ZeroCouponInflationSwap` instrument object for one or more Zero-Coupon Inflation Swap instruments.
2. Use `ratecurve` to specify an interest-rate model for the `ZeroCouponInflationSwap` instrument object.
3. Use `inflationcurve` to specify an inflation curve model.
4. Use `finpricer` to specify an `Inflation` pricing method for one or more `ZeroCouponInflationSwap` instruments.
5. Use `inflationCashflows` to compute cash flows for each one of the `ZeroCouponInflationSwap` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `ZeroCouponInflationSwap` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
ZCInflationSwap = fininstrument(InstrumentType,'Maturity',maturity_date,'
Notional',notional_value,'FixedInflationRate',inflation_rate)
ZCInflationSwap = fininstrument( ___ ,Name,Value)
```

Description

`ZCInflationSwap = fininstrument(InstrumentType,'Maturity',maturity_date,'Notional',notional_value,'FixedInflationRate',inflation_rate)` creates a `ZeroCouponInflationSwap` object for one or more Zero-Coupon Inflation Swap instruments by specifying `InstrumentType` and sets the properties on page 11-2594 for the required name-value pair arguments `Maturity`, `Notional`, and `FixedInflationRate`.

`ZCInflationSwap = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2594 using name-value pairs in addition to the required arguments in the previous syntax. For example, `ZCInflationSwap = fininstrument("ZeroCouponInflationSwap", 'Maturity',Maturity,'Notional',Notional,'FixedInflationRate',FixedInflationRate,'StartDate',StartDate)` creates a `ZeroCouponInflationSwap` instrument. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"ZeroCouponInflationSwap"` | string array with values of `"ZeroCouponInflationSwap"` | character vector with value `'ZeroCouponInflationSwap'` | cell array of character vectors with values of `'ZeroCouponInflationSwap'`

Instrument type, specified as a string with the value of `"ZeroCouponInflationSwap"`, a character vector with the value of `'ZeroCouponInflationSwap'`, an `NINST`-by-`1` string array with values of `"ZeroCouponInflationSwap"`, or an `NINST`-by-`1` cell array of character vectors with values of `'ZeroCouponInflationSwap'`.

Data Types: `char` | `cell` | `string`

`ZeroCouponInflationSwap` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ZCInflationSwap = fininstrument("ZeroCouponInflationSwap", 'Maturity',Maturity,'Notional',Notional,'FixedInflationRate',FixedInflationRate,'StartDate',StartDate)`

Required `ZeroCouponInflationSwap` Name-Value Pair Arguments

`Maturity` — Swap maturity date

datetime array | string array | date character vector

Swap maturity date, specified as the comma-separated pair consisting of `'Maturity'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `ZeroCouponInflationSwap` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`Notional` — Notional amount

scalar numeric | numeric vector

Notional amount, specified as the comma-separated pair consisting of `'Notional'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`FixedInflationRate` — Inflation rate

decimal | vector of decimals

Inflation rate, specified as the comma-separated pair consisting of `'FixedInflationRate'` and a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

Optional ZeroCouponInflationSwap Name-Value Pair Arguments

`StartDate` — Date swap starts

`Settle` date (default) | datetime array | string array | date character vector

Date swap starts, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. If not specified, when pricing the `ZeroCouponInflationSwap` instrument, the `Inflation` pricer uses the `Settle` date of the `DiscountCurve` as the `StartDate`.

To support existing code, `ZeroCouponInflationSwap` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `StartDate` property is stored as a datetime.

`Basis` — Day count basis for the fixed leg

`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis for the fixed leg, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer or an `NINST`-by-`1` vector of integers for the following:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Lag` — Indexation lag in months

`3` (default) | scalar numeric | numeric vector

Indexation lag in months, specified as the comma-separated pair consisting of `'Lag'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Name` — User-defined name for instrument

`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for the instrument, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Maturity` — Swap maturity date
scalar datetime | vector of datetimes

Swap maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Notional` — Notional amount
scalar numeric | numeric vector

Notional amount, returned as a scalar numeric or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`FixedInflationRate` — Inflation rate
scalar decimal | vector of decimals

Inflation rate, returned as a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`StartDate` — Date swap starts
`Settle` date (default) | scalar datetime | vector of datetimes

Date swap starts, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Basis` — Day count basis for fixed leg
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis for fixed leg, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Lag` — Indexation lag in months
`3` (default) | scalar numeric | numeric vector

Indexation lag in months, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string

User-defined name for the instrument, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

inflationCashflows Compute cash flows for ZeroCouponInflationSwap instrument

Examples

Price Zero Coupon Inflation Swap Instrument Using `inflationcurve` and Inflation Pricer

This example shows the workflow to price a `ZeroCouponInflationSwap` instrument when you use an `inflationcurve` object and an `Inflation` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2021
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2020, 10, 1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
     InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create ZeroCouponInflationSwap Instrument Object

Use `fininstrument` to create a `ZeroCouponInflationSwap` instrument object.

```
StartDate = datetime(2021,1,1);
Maturity = datetime(2022,10,1);
FixedInflationRate = 0.015;
Notional = 2000;

ZCInflationSwap = fininstrument("ZeroCouponInflationSwap",'StartDate',StartDate,'Maturity',Matur
```

```
ZCInflationSwap =
  ZeroCouponInflationSwap with properties:

             Notional: 2000
    FixedInflationRate: 0.0150
                Basis: 0
                  Lag: 3
            StartDate: 01-Jan-2021
             Maturity: 01-Oct-2022
                 Name: "zero_coupon_inflation_swap_instrument"
```

Create Inflation Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price ZeroCouponInflationSwap Instrument

Use `price` to compute the price and sensitivities for the `ZeroCouponInflationSwap` instrument.

```
[Price,outPR] = price(outPricer,ZCInflationSwap,"all")
```

```
Price = 9.5675

outPR =
  priceresult with properties:

       Results: [1x1 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=table
    Price
    ______
```

```
    9.5675
```

Price Multiple Zero Coupon Inflation Swap Instruments Using `inflationcurve` and Inflation Pricer

This example shows the workflow to price multiple `ZeroCouponInflationSwap` instruments when you use an `inflationcurve` object and an `Inflation` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2021,12,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Dec-2021
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2020, 10, 1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
    InflationIndexValues: [10x1 double]
   ForwardInflationRates: [9x1 double]
             Seasonality: [12x1 double]
```

Create `ZeroCouponInflationSwap` Instrument Object

Use `fininstrument` to create a `ZeroCouponInflationSwap` instrument object.

```
StartDate = datetime([2021,5,1 ; 2021,6,1 ; 2021,7,1]);
Maturity = datetime([2022,10,1 ; 2022,11,1 ;2022,12,1]);
FixedInflationRate = 0.015;
Notional = [20000 ; 30000 ; 40000] ;

ZCInflationSwap = fininstrument("ZeroCouponInflationSwap",'StartDate',StartDate,'Maturity',Matur
```

```
ZCInflationSwap=3×1 object
  3x1 ZeroCouponInflationSwap array with properties:

    Notional
    FixedInflationRate
    Basis
    Lag
    StartDate
    Maturity
    Name
```

Create `Inflation` Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price `ZeroCouponInflationSwap` Instruments

Use `price` to compute the prices and sensitivities for the `ZeroCouponInflationSwap` instruments.

```
[Price,outPR] = price(outPricer,ZCInflationSwap,"all")
```

```
Price = 3×1

   59.4576
   80.6037
   89.4137


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=table
    Price
    ______
```

```
    59.458


ans=table
    Price
    _____

    80.604


ans=table
    Price
    _____

    89.414
```

More About

Zero Coupon Inflation Swap

A zero coupon inflation swap is a type of derivative in which a fixed-rate payment on a notional amount is exchanged for a payment at the rate of inflation.

Algorithms

To price a zero-coupon inflation-indexed swap (ZCIS), use an inflation curve and a nominal discount curve (model-free approach), where the cash flows are discounted using the nominal discount curve.

Cash flows at maturity $t = T_M$:

$FixedLeg = N \times [(^1 - 1]$

$InflationLeg = N \times \left[\frac{I(T_M)}{I_0} - 1\right]$

where

- N is the reference notional of the swap.
- k is the fixed inflation rate.
- M is the number of years for the life of the swap.
- $I(T_M)$ is the inflation index at the maturity date with some lag (for example, three months).
- I_0 is the inflation index at the start date with some lag (for example, three months).

Version History

Introduced in R2021a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `ZeroCouponInflationSwap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Brody, D. C., Crosby, J., and Li, H. "Convexity Adjustments in Inflation-Linked Derivatives." *Risk Magazine*. November 2008, pp. 124–129.

[2] Kerkhof, J. "Inflation Derivatives Explained: Markets, Products, and Pricing." *Fixed Income Quantitative Research*, Lehman Brothers, July 2005.

[3] Zhang, J. X. "Limited Price Indexation (LPI) Swap Valuation Ideas." *Wilmott Magazine*. no. 57, January 2012, pp. 58–69.

See Also

Functions

`InflationBond` | `YearYearInflationSwap` | `finpricer`

Topics

“Analyze Inflation-Indexed Instruments” on page 2-132
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Inflation

Create `Inflation` pricer object for `InflationBond`, `YearYearInflationSwap`, or `ZeroCouponInflationSwap` instrument using `inflationcurve` model

Description

Create and price an `InflationBond`, `YearYearInflationSwap`, or `ZeroCouponInflationSwap` instrument object with an `inflationcurve` model and an `Inflation` pricing method using this workflow:

1. Use `fininstrument` to create an `InflationBond`, `YearYearInflationSwap`, or `ZeroCouponInflationSwap` instrument object.
2. Use `inflationcurve` to specify an inflation curve object.
3. Create an interest-rate curve object using `ratecurve`.
4. Use `finpricer` to specify the `Inflation` pricer object for the `InflationBond`, `YearYearInflationSwap`, or `ZeroCouponInflationSwap` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for an `InflationBond`, `YearYearInflationSwap`, or `ZeroCouponInflationSwap` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
InflationPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
InflationCurve',inflationcurve_obj)
```

Description

`InflationPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'InflationCurve',inflationcurve_obj)` creates an `Inflation` pricer object by specifying `PricerType` and the required name-value pair arguments `DiscountCurve` and `InflationCurve` to set properties on page 11-2602 using name-value pairs. For example, `InflationPricerObj = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)` creates an `Inflation` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Inflation"` | character vector with value `'Inflation'`

Pricer type, specified as a string with the value of `"Inflation"` or a character vector with the value of `'Inflation'`.

Data Types: `char` | `string`

`Inflation` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `InflationPricerObj = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a previously created `ratecurve` object.

Data Types: `object`

`InflationCurve` — Inflation curve
`inflationcurve` object

Inflation curve, specified as the comma-separated pair consisting of `'InflationCurve'` and the name of the previously created `inflationcurve`.

Data Types: `object`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`InflationCurve` — Inflation curve
`inflationcurve` object

Inflation curve, returned as an `inflationcurve` object.

Data Types: `object`

Object Functions

price Compute price for inflation instrument with Inflation pricer

Examples

Use Inflation Pricer and `inflationcurve` to Price Inflation Bond Instrument

This example shows the workflow to price an `InflationBond` instrument when you use an `inflationcurve` object and an `Inflation` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2021,1,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2021
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `inflationcurve` Object

Create an `inflationcurve` object using `inflationcurve`.

```
BaseDate = datetime(2020,10,1);
InflationTimes = [0 calyears([1 2 3 4 5 7 10 20 30])]';
InflationIndexValues = [100 102 103.5 105 106.8 108.2 111.3 120.1 130.4 150.2]';
InflationDates = BaseDate + InflationTimes;
myInflationCurve = inflationcurve(InflationDates,InflationIndexValues)
```

```
myInflationCurve =
  inflationcurve with properties:

                   Basis: 0
                   Dates: [10x1 datetime]
    InflationIndexValues: [10x1 double]
    ForwardInflationRates: [9x1 double]
              Seasonality: [12x1 double]
```

Create `InflationBond` Instrument Object

Use `fininstrument` to create an `InflationBond` instrument object.

```
IssueDate = datetime(2021,1,1);
Maturity = datetime(2026,1,1);
CouponRate = 0.02;

InflationBond = fininstrument("InflationBond",'IssueDate',IssueDate,'Maturity',Maturity,'CouponRa
```

```
InflationBond =
  InflationBond with properties:

                 CouponRate: 0.0200
```

```
                    Period: 2
                     Basis: 0
                 Principal: 100
    DaycountAdjustedCashFlow: 0
                       Lag: 3
       BusinessDayConvention: "actual"
                   Holidays: NaT
              EndMonthRule: 1
                 IssueDate: 01-Jan-2021
           FirstCouponDate: NaT
            LastCouponDate: NaT
                  Maturity: 01-Jan-2026
                      Name: "inflation_bond_instrument"
```

Create `Inflation` Pricer Object

Use `finpricer` to create an `Inflation` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument and the `inflationcurve` object with the `'InflationCurve'` name-value pair argument.

```
outPricer = finpricer("Inflation",'DiscountCurve',ZeroCurve,'InflationCurve',myInflationCurve)
```

```
outPricer =
  Inflation with properties:

     DiscountCurve: [1x1 ratecurve]
    InflationCurve: [1x1 inflationcurve]
```

Price `InflationBond` Instrument

Use `price` to compute the price and sensitivities for the `InflationBond` instrument.

```
[Price,outPR] = price(outPricer,InflationBond)
```

```
Price = 112.1856

outPR =
  priceresult with properties:

       Results: [1x1 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=table
    Price
    ______

    112.19
```

Version History

Introduced in R2021a

See Also

Functions
`fininstrument` | `ratecurve`

Topics
“Analyze Inflation-Indexed Instruments” on page 2-132
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Rubinstein

Create `Rubinstein` pricer object for `Cliquet` instrument using `BlackScholes` model

Description

Create and price a `Cliquet` instrument object with a `BlackScholes` model and a `Rubinstein` pricing method using this workflow:

1. Use `fininstrument` to create an `Cliquet` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Cliquet` instrument object.
3. Use `finpricer` to specify a `Rubinstein` pricer object for the `Cliquet` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for an `Cliquet` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
RubinsteinPricerObj = finpricer(PricerType,DiscountCurve=ratecurve_obj,
Model=model,SpotPrice=spotprice_value)
RubinsteinPricerObj = finpricer( ___ ,Name=Value)
```

Description

`RubinsteinPricerObj = finpricer(PricerType,DiscountCurve=ratecurve_obj, Model=model,SpotPrice=spotprice_value)` creates a `Rubinstein` pricer object by specifying `PricerType` and sets the properties on page 11-2608 for the required name-value arguments `DiscountCurve`, `Model`, and `SpotPrice`.

`RubinsteinPricerObj = finpricer( ___ ,Name=Value)` sets optional properties on page 11-3325 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `RubinsteinPricerObj = finpricer("Analytic",DiscountCurve=ratecurve_obj,Model=BSModel,SpotPrice=1000 ,DividendType="continuous",DividendValue=100,PricingMethod="Rubinstein")` creates a `Cliquet` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: `string` | `char`

Rubinstein Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `RubinsteinPricerObj = finpricer("Analytic",DiscountCurve=ratecurve_obj,Model=BSModel,SpotPrice=1000,DividendType="continuous",DividendValue=100,PricingMethod="Rubinstein")`

Required Rubinstein Name-Value Arguments

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as `DiscountCurve` and the name of a previously created `ratecurve` object.

Note Specify a flat `ratecurve` object for `DiscountCurve`. If you use a nonflat `ratecurve` object, the software uses the rate in the `ratecurve` object at `Maturity` and assumes that the value is constant for the life of the equity option.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, specified as `Model` and the name of a previously created `BlackScholes` model object using `finmodel`.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, specified as `SpotPrice` and a scalar nonnegative numeric.

Data Types: `double`

Optional Rubinstein Name-Value Arguments

`DividendType` — Dividend type
`"continuous"` (default) | string with value of `"continuous"` | character vector with value of `'continuous'`

Dividend type, specified as `DividendType` and a string or character vector for a continuous dividend yield.

Data Types: `char` | `string`

`DividendValue` — Dividend yield for underlying stock
`0` (default) | scalar numeric

Dividend yield for the underlying stock, specified as `DividendValue` and a scalar numeric.

Data Types: `double`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"Rubinstein"` | character vector with value `'Rubinstein'`

Analytic pricing method, specified as `PricingMethod` and a character vector or string.

Note The default pricing method for a `BlackScholes` model is a `BlackScholes` pricer.

Data Types: `double`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendType` — Dividend type
`"continuous"` (default) | string with value of `"continuous"`

This property is read-only.

Dividend type, returned as a string.

Data Types: `string`

`DividendValue` — Dividend yield for underlying stock
`0` (default) | numeric

Dividend yield for the underlying stock, returned as a scalar numeric.

Data Types: `double`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"Rubinstein"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Rubinstein Pricer and Black-Scholes Model to Price the Absolure Return for Cliquet Instruments

This example shows the workflow to price the absolute return for three `Cliquet` instruments when you use a `BlackScholes` model and a `Rubinstein` pricing method.

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,Basis=12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create `Cliquet` Instrument Object

Use `fininstrument` to create a `Cliquet` instrument object for three Cliquet instruments.

```
ResetDates = Settle + years(0:0.25:1);
CliquetOpt = fininstrument("Cliquet",ResetDates=ResetDates,InitialStrike=[140;150;160],ExerciseS
```

```
CliquetOpt=3×1 object
  3x1 Cliquet array with properties:

    OptionType
    ExerciseStyle
    ResetDates
    LocalCap
    LocalFloor
    GlobalCap
    GlobalFloor
    ReturnType
    InitialStrike
```

```
    Name
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",Volatility=0.28)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2800
    Correlation: 1
```

Create Rubinstein Pricer Object

Use `finpricer` to create a `Rubinstein` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",DiscountCurve=myRC,Model=BlackScholesModel,SpotPrice=135,Divider
```

```
outPricer =
  Rubinstein with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 135
    DividendValue: 0.0250
     DividendType: "continuous"
```

Price Cliquet Instruments

Use `price` to compute the price and sensitivities for the three `Cliquet` instruments.

```
[Price, outPR] = price(outPricer,CliquetOpt,"all")
```

```
Price = 3×1

   28.1905
   25.3226
   23.8168


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Rho      Theta
    _____    _______    _________    ______    ______    _____    _____
```

```
    28.191    0.59697    0.020662    2.8588    105.38    60.643    -14.62

ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Rho       Theta
    _____    ________    _________    ______    ______    ______    ________

    25.323    0.41949    0.016816    2.2364    100.47    55.367    -11.708

ans=1×7 table
    Price      Delta      Gamma      Lambda     Vega      Rho      Theta
    _____    ________    _________    ______    ______    ______    ______

    23.817    0.29729    0.011133    1.6851    93.219    51.616    -7.511
```

Version History

Introduced in R2021b

See Also

Functions

`fininstrument` | `finmodel` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

parametercurve

Create `parametercurve` object for storing interest-rate curve function

Description

Build a `parametercurve` object using `parametercurve`.

After creating a `parametercurve` object, you can use the associated object functions `discountfactors`, `zerorates`, `forwardrates`, `fitNelsonSiegel`, and `fitSvensson`.

For more detailed information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
parametercurve_obj = parametercurve(Type,Settle,FunctionHandle)
parametercurve_obj = parametercurve( ___ ,Name,Value)
```

Description

`parametercurve_obj = parametercurve(Type,Settle,FunctionHandle)` creates a `parametercurve` object.

`parametercurve_obj = parametercurve( ___ ,Name,Value)` sets properties on page 11-2614 using name-value pairs and any of the arguments in the previous syntax. For example, `parametercurve_obj = parametercurve('zero',datetime(2017,1,30),@(t)polyval([-0.0001 0.003 0.02],t),'Compounding',4,'Basis',5,'Parameters',[-0.0001 0.003 0.02])` creates a `parametercurve` object for a zero curve. You can specify multiple name-value pair arguments.

Input Arguments

Type — Type of interest-rate curve
string with value `"zero"`, `"forward"`, or `"discount"` | character vector with value `'zero'`, `'forward'`, or `'discount'`

Type of interest-rate curve, specified as a scalar string or character vector for one of the supported types.

Data Types: `char` | `string`

Settle — Settlement date for the curve
datetime scalar | string scalar | date character vector

Settlement date for the curve, specified as a scalar datetime, string, or date character vector.

To support existing code, `parametercurve` also accepts serial date numbers as inputs, but they are not recommended.

`FunctionHandle` — Dates corresponding to rate data

function handle

Dates corresponding to the rate data, specified as a function handle. The function handle requires one numeric input (time-to-maturity) and returns one numeric output (interest rate or discount factor). For more information on creating a function handle, see “Create Function Handle”.

Data Types: `function_handle`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
```
parametercurve_obj =
parametercurve('zero',datetime(2017,1,30),@(t)polyval([-0.0001 0.003
0.02],t),'Compounding',4,'Basis',5,'Parameters',[-0.0001 0.003 0.02])
```

`Compounding` — Compounding frequency for curve

`-1` (default) | possible values include: –1, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

Compounding frequency for the curve, specified as the comma-separated pair consisting of `'Compounding'` and a scalar numeric using the supported values: –1, `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day count basis

`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)
- 11 — 30/360E (ICMA)

- 12 — actual/365 (ISDA)
- 13 — BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Parameters` — Curve parameters
`[ ]` (default) | numeric

Curve parameters, specified as the comma-separated pair consisting of `'Parameters'` and a numeric value.

Data Types: `double`

Properties

Type — Type of interest-rate curve
string with value `"zero"`, `"forward"`, or `"discount"`

Instrument type, returned as a string.

Data Types: `string`

`Settle` — Settlement date
datetime

Settlement date, returned as a datetime.

Data Types: `datetime`

`FunctionHandle` — Dates corresponding to rate data
function handle

Function handle that defines the interest-rate curve, returned as a scalar function handle.

Data Types: `function_handle`

`Compounding` — Compounding frequency for curve
`-1` (default) | possible values include: –1, `0`, `1`, `2`, `3`, `4`, `6`, `12`.

Compounding frequency for curve, returned as a scalar numeric.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13`

Day count basis, returned as a scalar integer.

Data Types: `double`

`Parameters` — Curve parameters
`[ ]` (default) | numeric

Curve parameters, returned as a numeric value.

Data Types: `double`

Object Functions

discountfactors	Calculate discount factors for parametercurve object
zerorates	Calculate zero rates for parametercurve object
forwardrates	Calculate forward rates for parametercurve object
fitNelsonSiegel	Fit Nelson-Siegel model to bond market data
fitSvensson	Fit Svensson model to bond market data

Examples

Create `parametercurve` Object

Create a `parametercurve` object using `parametercurve`.

```
PCobj = parametercurve('zero',datetime(2019,9,15),@(t)polyval([-0.0001 0.003 0.02],t),'Compoundi
```

```
PCobj =
  parametercurve with properties:

              Type: "zero"
            Settle: 15-Sep-2019
       Compounding: 4
             Basis: 5
    FunctionHandle: @(t)polyval([-0.0001,0.003,0.02],t)
        Parameters: [-1.0000e-04 0.0030 0.0200]
```

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `parametercurve` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`ratecurve`

Topics

Asian

`Asian` instrument object

Description

Create and price an `Asian` instrument object for one ore more Asian instruments using this workflow:

1. Use `fininstrument` to create an `Asian` instrument object for one or more Asian instruments.
2. Use `finmodel` to specify a `BlackScholes`, `Heston`, `Bates`, or `Merton` model for the `Asian` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify a `Levy`, `KemnaVorst`, `AssetTree`, or `TurnbullWakeman` pricing method for one or more `Asian` instruments.
   - When using a `BlackScholes`, `Heston`, `Bates`, or `Merton` model, use `finpricer` to specify an `AssetMonteCarlo` pricing method for one or more `Asian` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for an `Asian` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
AsianOpt = fininstrument(InstrumentType,'Strike',strike_price,'
ExerciseDate',exercise_date)
AsianOpt = fininstrument( ___ ,Name,Value)
```

Description

`AsianOpt = fininstrument(InstrumentType,'Strike',strike_price,'ExerciseDate',exercise_date)` creates an `Asian` instrument object for one or more Asian instruments by specifying `InstrumentType` and sets the properties on page 11-2620 for the required name-value pair arguments `Strike` and `ExerciseDate`.

The `Asian` instrument supports arithmetic and geometric, fixed-strike, and floating-strike Asian options.

`AsianOpt = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2620 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `AsianOpt = fininstrument("Asian",'Strike',100,'ExerciseDate',datetime(2019,1,30),'Option`

Type','put','ExerciseStyle','European','Name','asian_option') creates an Asian put option with an European exercise. You can specify multiple name-value pair arguments.

Input Arguments

InstrumentType — Instrument type

string with value "Asian" | string array with values of "Asian" | character vector with value 'Asian' | cell array of character vectors with values of 'Asian'

Instrument type, specified as a string with the value of "Asian", a character vector with the value of 'Asian', an NINST-by-1 string array with values of "Asian", or an NINST-by-1 cell array of character vectors with values of 'Asian'.

Data Types: string | char | cell

Asian Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: AsianOpt = fininstrument("Asian",'Strike',100,'ExerciseDate',datetime(2019,1,30),'OptionType','put','ExerciseStyle','European','Name','asian_option")

Required Asian Name-Value Pair Arguments

Strike — Option strike price value

nonnegative value | vector of nonnegative values

Option strike price value, specified as the comma-separated pair consisting of 'Strike' and a scalar nonnegative value or an NINST-by-1 vector of nonnegative values.

Data Types: double

ExerciseDate — Option exercise dates

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of 'ExerciseDate' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

Note For an Asian European option, there is only one ExerciseDate on the option expiry date.

To support existing code, Asian also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by datetime because the ExerciseDate property is stored as a datetime.

Optional Asian Name-Value Pair Arguments

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Option type, specified as the comma-separated pair consisting of `'OptionType'` and a scalar character vector or string or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `string` | `cell`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` | string array with a value `"European"` | character vector with value `'European'` | cell array of character vectors with a value `'European'`

Option exercise style, specified as the comma-separated pair consisting of `'ExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `string` | `char` | `cell`

`AverageType` — Average type
`"arithmetic"` (default) | string with value `"arithmetic"` or `"geometric"` | string array with values `"arithmetic"` or `"geometric"` | character vector with value `'arithmetic'` or `'geometric'` | cell array of character vectors with values `'arithmetic'` or `'geometric'`

Average types, specified as the comma-separated pair consisting of `'AverageType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. Use `"arithmetic"` for an arithmetic average, or `"geometric"` for a geometric average.

Note When you use a `RollGeskeWhaley` pricer, the `AverageType` must be `"geometric"`.

Data Types: `char` | `cell` | `string`

`AveragePrice` — Average price of underlying asset
`0` (default) | scalar numeric | numeric vector

Average price of the underlying asset, specified as the comma-separated pair consisting of `'AveragePrice'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`AverageStartDate` — Start date of averaging period
datetime array | string array | date character vector

Start date of averaging period, specified as the comma-separated pair consisting of `'AverageStartDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `Asian` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `AverageStartDate` property is stored as a `datetime`.

Name — User-defined name for instrument
" " (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of 'Name' and a scalar string or character vector or an NINST-by-1 cell array of character vectors or string array.

Data Types: char | cell | string

Properties

Strike — Option strike price value
nonnegative value | vector of nonnegative values

Option strike price value, returned as a scalar nonnegative value or an NINST-by-1 vector of nonnegative values.

Data Types: double

ExerciseDate — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: datetime

OptionType — Option type
"call" (default) | scalar string with value "call" or "put" | string array with values "call" or "put"

Option type, returned as a scalar string or an NINST-by-1 string array with the values of "call" or "put" .

Data Types: string

ExerciseStyle — Option exercise style
"European" (default) | scalar string with value "European" | string array with values "European"

Option exercise style, returned as a scalar string with the value "European" or NINST-by-1 string array.

Data Types: string

AverageType — Average type
"arithmetic" (default) | scalar string with value "arithmetic" or "geometric" | string array with value "arithmetic" or "geometric"

Average types, returned as a scalar string with the value "arithmetic" for arithmetic average or "geometric" for geometric average or an NINST-by-1 string array.

Data Types: string

AveragePrice — Average price of underlying asset at Settle
0 (default) | scalar numeric | numeric vector

Average price of underlying asset at Settle, returned as a scalar numeric or an NINST-by-1 numeric vector.

Data Types: `double`

`AverageStartDate` — Start date of averaging period
`NaT` (default) | datetime | vector of datetimes

Start date of averaging period, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

Examples

Price Asian Instrument Using a Black-Scholes Model and Turnbull-Wakeman Pricer

This example shows the workflow to price a fixed-strike `Asian` instrument when you use a `BlackScholes` model and a `TurnbullWakeman` pricing method.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',1000,'OptionType',"
```

```
AsianOpt =
  Asian with properties:

          OptionType: "put"
              Strike: 1000
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create ratecurve Object

Create a flat ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create TurnbullWakeman Pricer Object

Use finpricer to create a TurnbullWakeman pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',1000
```

```
outPricer =
  TurnbullWakeman with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 1000
    DividendValue: 0
     DividendType: "continuous"
```

Price Asian Instrument

Use price to compute the price and sensitivities for the Asian instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 56.7068
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma      Lambda      Vega      Theta       Rho
```

```
    ______    _______    _________    _______    ______    _______    _______

    56.707    -0.3155    0.0014381    -5.5637    408.85    -2.9341    -832.53
```

Price Multiple Asian Instruments Using a Black-Scholes Model and Turnbull-Wakeman Pricer

This example shows the workflow to price multiple fixed-strike `Asian` instruments when you use a `BlackScholes` model and a `TurnbullWakeman` pricing method.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object for three Asian instruments.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime([2022,9,15; 2022,10,15; 2022,11,15]),'S
```

```
AsianOpt=3×1 object
  3x1 Asian array with properties:

    OptionType
    Strike
    AverageType
    AveragePrice
    AverageStartDate
    ExerciseStyle
    ExerciseDate
    Name
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
```

```
                Dates: 15-Sep-2023
                Rates: 0.0350
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `TurnbullWakeman` Pricer Object

Use `finpricer` to create a `TurnbullWakeman` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',1000,
```

```
outPricer =
  TurnbullWakeman with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 1000
    DividendValue: 0
     DividendType: "continuous"
```

Price `Asian` Instruments

Use `price` to compute the prices and sensitivities for the `Asian` instruments.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 3×1
10^3 ×

    0.0567
    0.8023
    1.6624


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Lambda      Vega      Theta       Rho
    ______    ________    __________    _______    ______    _______    _______

    56.707    -0.3155    0.0014381    -5.5637    408.85    -2.9341    -832.53


ans=1×7 table
    Price      Delta        Gamma        Lambda      Vega      Theta       Rho
    ______    ________    __________    _______    ______    _______    _______
```

```
    802.32    -0.92568    7.9581e-05    -1.1537    20.935    44.139    -5206.3

ans=1×7 table
    Price      Delta        Gamma        Lambda       Vega       Theta       Rho
    ______    ________    __________    ________    ________    ______    _______

    1662.4    -0.93048    4.5475e-05    -0.55973    0.093861    74.863    -8911.1
```

Price Asian Instrument Using a Black-Scholes Model and Asset Tree Pricer for CRR Binomial Tree

This example shows the workflow to price a fixed-strike `Asian` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method for a Cox-Ross-Rubinstein (CRR) binomial tree.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',1000,'OptionType',"p
```

```
AsianOpt =
  Asian with properties:

          OptionType: "put"
              Strike: 1000
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object for a CRR equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
NumPeriods = 15;
CRRPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',100
```

```
CRRPricer =
  CRRTree with properties:

            Tree: [1x1 struct]
       NumPeriods: 15
            Model: [1x1 finmodel.BlackScholes]
    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 1000
     DividendType: "continuous"
    DividendValue: 0
        TreeDates: [21-Dec-2018 09:36:00    28-Mar-2019 19:12:00    ...    ]
```

```
CRRPricer.Tree
```

```
ans = struct with fields:
    Probs: [2x15 double]
    ATree: {1x16 cell}
     dObs: [15-Sep-2018 00:00:00    21-Dec-2018 09:36:00    ...    ]
     tObs: [0 0.2667 0.5333 0.8000 1.0667 1.3333 1.6000 1.8667 2.1333 ... ]
```

Price Asian Instrument

Use `price` to compute the price and sensitivities for the `Asian` instrument.

```
[Price, outPR] = price(CRRPricer,AsianOpt,["all"])
```

```
Price = 54.9225
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Vega      Lambda       Rho        Theta
    ______    ________    ______    ______    ________    ________    ________

    54.922    -0.32119    0.0581    393.85    -5.8481    -846.57    -2.4325
```

Price Asian Instrument Using a Black-Scholes Model and Asset Tree Pricer for Standard Trinomial Tree

This example shows the workflow to price a fixed-strike `Asian` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method for a Standard Trinomial (STT) tree.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',1000,'OptionType',"p
```

```
AsianOpt =
  Asian with properties:

          OptionType: "put"
              Strike: 1000
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:
```

```
                Type: "zero"
        Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object for a Standard Trinomial tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
NumPeriods = 15;
STTPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',100
```

```
STTPricer =
  STTree with properties:

            Tree: [1x1 struct]
       NumPeriods: 15
           Model: [1x1 finmodel.BlackScholes]
    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 1000
     DividendType: "continuous"
    DividendValue: 0
        TreeDates: [21-Dec-2018 09:36:00    28-Mar-2019 19:12:00    ...    ]
```

```
STTPricer.Tree
```

```
ans = struct with fields:
    ATree: {1x16 cell}
    Probs: {1x15 cell}
     dObs: [15-Sep-2018 00:00:00    21-Dec-2018 09:36:00    ...    ]
     tObs: [0 0.2667 0.5333 0.8000 1.0667 1.3333 1.6000 1.8667 2.1333 ... ]
```

Price Asian Instrument

Use `price` to compute the price and sensitivities for the `Asian` instrument.

```
[Price, outPR] = price(STTPricer,AsianOpt,["all"])
```

```
Price = 54.2450
```

```
outPR =
  priceresult with properties:

      Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Vega      Lambda       Rho       Theta
```

```
    _____    ________    ________    _____    _______    _______    _______

    54.245    -0.32307    0.075269    390.55    -5.9558    -839.02    -2.4161
```

Price Asian Instrument for Foreign Exchange Using Black-Scholes Model and Levy Pricer

This example shows the workflow to price an `Asian` instrument for an arithmetic average currency option when you use a `BlackScholes` model and a `Levy` pricing method. Assume that the current exchange rate is $0.52 and has a volatility of 12% per annum. The annualized continuously compounded foreign risk-free rate is 8% per annum.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',0.65,'OptionType',"p
```

```
AsianOpt =
  Asian with properties:

         OptionType: "put"
             Strike: 0.6500
        AverageType: "arithmetic"
       AveragePrice: 0
   AverageStartDate: NaT
      ExerciseStyle: "european"
       ExerciseDate: 15-Sep-2022
               Name: "asian_fx_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
Sigma = .12;
BlackScholesModel = finmodel("BlackScholes",'Volatility',Sigma)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.1200
    Correlation: 1
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:
```

```
                Type: "zero"
        Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Levy Pricer Object

Use `finpricer` to create a `Levy` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument. When you price currencies using an Asian instrument for an arithmetic average currency option, the `DividendType` must be `'continuous'` and `DividendValue` is the annualized risk-free interest rate in the foreign country.

```
ForeignRate = 0.08;
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',.52,
```

```
outPricer =
  Levy with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 0.5200
    DividendValue: 0.0800
     DividendType: "continuous"
```

Price Asian FX Instrument

Use `price` to compute the price and sensitivities for the `Asian` FX instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 0.1516

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
     Price      Delta       Gamma      Lambda       Vega         Theta          Rho
    _______    ________    _______    _______    ________    __________    _______

    0.15161    -0.78532    0.37534    -2.6935    0.015668    -0.0038317    -1.3974
```

Price Asian Instrument Using a Black-Scholes Model and Asset Monte-Carlo Pricer

This example shows the workflow to price a fixed-strike `Asian` instrument when you use a `BlackScholes` model and an `AssetMonteCarlo` pricing method.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',1000,'OptionType',"p
```

```
AsianOpt =
  Asian with properties:

          OptionType: "put"
              Strike: 1000
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use finpricer to create an AssetMonteCarlo pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BlackScholesModel,'SpotPrice
```

```
outPricer =
  GBMMonteCarlo with properties:

     DiscountCurve: [1x1 ratecurve]
         SpotPrice: 200
    SimulationDates: 15-Sep-2022
         NumTrials: 1000
     RandomNumbers: []
             Model: [1x1 finmodel.BlackScholes]
      DividendType: "continuous"
     DividendValue: 0
```

Price Asian Instrument

Use price to compute the price and sensitivities for the Asian instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 682.3365
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta         Gamma         Lambda       Rho      Theta      Vega
    _____    ________    ___________    ________    _______    _____    _______

    682.34    -0.93511    -5.6843e-14    -0.27409    -3129.1    27.433    -1.2121
```

Price Asian Instrument Using a Merton Model and Asset Monte-Carlo Pricer

This example shows the workflow to price a fixed-strike Asian instrument when you use a Merton model and an AssetMonteCarlo pricing method.

Create Asian Instrument Object

Use fininstrument to create an Asian instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',1000,'OptionType',"p
```

```
AsianOpt =
  Asian with properties:
```

```
          OptionType: "put"
              Strike: 1000
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_option"
```

Create `Merton` Model Object

Use `finmodel` to create a `Merton` model object.

```
MertonModel = finmodel("Merton",'Volatility',0.45,'MeanJ',0.02,'JumpVol',0.07,'JumpFreq',0.09)
```

```
MertonModel =
  Merton with properties:

    Volatility: 0.4500
         MeanJ: 0.0200
       JumpVol: 0.0700
      JumpFreq: 0.0900
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",MertonModel,'SpotPrice',200
```

```
outPricer =
  MertonMonteCarlo with properties:
```

```
      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 200
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Merton]
       DividendType: "continuous"
      DividendValue: 0
```

Price Asian Instrument

Use `price` to compute the price and sensitivities for the `Asian` instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 682.8127

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Lambda       Rho       Theta      Vega
    _____    ________    _____    ________    ________    _____    _______

    682.81    -0.90665      0      -0.26556    -3110.3    25.98    19.316
```

More About

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.

Asian options 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. For more information, see “Asian Option” on page 3-34.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Asian` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`finmodel` | `finpricer`

Topics
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

Barrier

`Barrier` instrument object

Description

Create and price a `Barrier` instrument object for one or more Barrier instruments using this workflow:

1. Use `fininstrument` to create a `Barrier` instrument object for one or more Barrier instruments.
2. Use `finmodel` to specify a `BlackScholes`, `Heston`, `Bates`, or `Merton` model for the `Barrier` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify a `BlackScholes`, `AssetTree`, or `VannaVolga` pricing method for one or more `Barrier` instruments.
   - When using a `BlackScholes`, `Heston`, `Bates`, or `Merton` model, use `finpricer` to specify an `AssetMonteCarlo` or `FiniteDifference` pricing method for one ore more `Barrier` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Barrier` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BarrierOpt = fininstrument(InstrumentType,'Strike',strike_value,'
ExerciseDate',exercise_date,'BarrierValue',barrier_value)
BarrierOpt = fininstrument( ___ ,Name,Value)
```

Description

`BarrierOpt = fininstrument(InstrumentType,'Strike',strike_value,'ExerciseDate',exercise_date,'BarrierValue',barrier_value)` creates a `Barrier` instrument object for one or more Barrier instruments by specifying `InstrumentType` and sets the properties on page 11-2639 for the required name-value pair arguments `Strike`, `ExerciseDate`, and `BarrierValue`.

`BarrierOpt = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2639 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `BarrierOpt = fininstrument("Barrier",'Strike',100,'ExerciseDate',datetime(2019,1,30),'BarrierValue',110,'OptionType',"put",'ExerciseStyle',"European",'BarrierType',"DO`

",'Name',"barrier_option") creates a Barrier put option with an European exercise. You can specify multiple name-value pair arguments.

Input Arguments

InstrumentType — Instrument type

string with value "Barrier" | string with value "Barrier" | string array with values of "Barrier" | character vector with value 'Barrier' | cell array of character vectors with values of 'Barrier'

Instrument type, specified as a string with the value of "Barrier", a character vector with the value of 'Barrier', an NINST-by-1 string array with values "Barrier", or an NINST-by-1 cell array of character vectors with values of 'Barrier'.

Data Types: char | string | cell

Barrier Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: BarrierOpt = fininstrument("Barrier",'Strike',100,'ExerciseDate',datetime(2019,1,30),'BarrierValue',110,'OptionType',"put",'ExerciseStyle',"European",'BarrierType',"DO",'Name',"barrier_option")

Required Barrier Name-Value Pair Arguments

Strike — Option strike value

nonnegative value | vector of nonnegative values

Option strike value, specified as the comma-separated pair consisting of 'Strike' and a scalar nonnegative value or an NINST-by-1 vector of nonnegative values.

Data Types: double

ExerciseDate — Option exercise date

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of 'ExerciseDate' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one ExerciseDate on the option expiry date.

To support existing code, Barrier also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by datetime because the ExerciseDate property is stored as a datetime.

BarrierValue — Barrier level

scalar numeric | numeric vector

Barrier level, specified as the comma-separated pair consisting of `'BarrierLevel'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

Optional Barrier Name-Value Pair Arguments

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Option type, specified as the comma-separated pair consisting of `'OptionType'` and a scalar character vector or string or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` or `"American"` | string array with values `"European"` or `"American"` | character vector with value `'European'` or `'American'` | cell array of character vectors with values `'European'` or `'American'`

Option exercise style, specified as the comma-separated pair consisting of `'ExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Note For a `Barrier` option, the `BlackScholes` pricer supports only `"European"` exercise and the `FiniteDifference` pricer supports an `"American"` or `"European"` exercise.

Data Types: `string` | `char` | `cell`

`BarrierType` — Barrier option type
`"UO"` (default) | string with value `"UI"`, `"UO"`, `"DI"`, or `"DO"` | string array with values `"UI"`, `"UO"`, `"DI"`, or `"DO"` | character vector with value `'UI'`, `'UO'`, `'DI'`, or `'DO'` | cell array of character vectors with values `'UI'`, `'UO'`, `'DI'`, or `'DO'`

Barrier option type, specified as the comma-separated pair consisting of `'BarrierType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array with one of the following values:

- `"UI"` — Up knock-in

  This option becomes effective when the price of the underlying asset passes above the barrier level. If the underlying asset goes above the barrier level during the life of the option, the option holder has the right, but not the obligation, to buy or sell (call or put) the underlying security at the strike price.
- `"UO"` — Up knock-out

  This option gives the option holder the right, but not the obligation, to buy or sell (call or put) the underlying security at the strike price as long as the underlying asset does not go above the barrier level during the life of the option. This option terminates when the price of the underlying security passes above the barrier level. If the spot price of the underlying asset reaches or exceeds the barrier level with an up-and-out option, the rebate is paid.
- `"DI"` — Down knock-in

This option becomes effective when the price of the underlying stock passes below the barrier level. If the underlying security goes below the barrier level during the life of the option, the option holder has the right, but not the obligation, to buy or sell (call or put) the underlying security at the strike price. With a down-and-in option, the rebate is paid if the spot price of the underlying does not reach the barrier level during the life of the option. Note that a `Barrier` instrument using the `FiniteDifference` pricer does not support American knock-in barrier options.

- `"DO"` — Down knock-up

  This option gives the option holder the right, but not the obligation, to buy or sell (call or put) the underlying asset at the strike price as long as the underlying asset does not go below the barrier level during the life of the option. This option terminates when the price of the underlying security passes below the barrier level. If the option is worthless when it expires, the option holder receives a rebate amount.

Option	Barrier Type	Payoff If Barrier Crossed	Payoff If Barrier Not Crossed
Call or Put	Down knock-out	Worthless	Standard Call or Put
Call or Put	Down knock-in	Call or Put	Worthless
Call or Put	Up knock-out	Worthless	Standard Call or Put
Call or Put	Up knock-in	Standard Call or Put	Worthless

Data Types: `char` | `cell` | `string`

`Rebate` — Rebate value
`0` (default) | scalar numeric | numeric vector

Rebate value, specified as the comma-separated pair consisting of `'Rebate'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

- For knock-in options, the `Rebate` is paid at expiry.
- For knock-out options, the `Rebate` is paid when `BarrierValue` is reached.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for the instrument, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Strike` — Option strike value
nonnegative value | vector of nonnegative values

Option strike value, returned as a scalar nonnegative value or an `NINST`-by-`1` vector of nonnegative values.

Data Types: `double`

`ExerciseDate` — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values `"call"` or `"put"`

Option type, returned as a scalar string or an `NINST`-by-`1` string array with the values of `"call"` or `"put"`.

Data Types: `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` or `"American"` | string array with values `"European"` or `"American"`

Option exercise style, returned as a scalar string or `NINST`-by-`1` string array with the values of `"European"` or `"American"`.

Data Types: `string`

`BarrierSpec` — Barrier option type
`"UO"` European (default) | string with value `"UI"`, `"UO"`, `"DI"`, or `"DO"` | string array with values `"UI"`, `"UO"`, `"DI"`, or `"DO"`

Barrier option type, returned as a scalar string or `NINST`-by-`1` string array with the values of `"UI"`, `"UO"`, `"DI"`, or `"DO"`.

Data Types: `string`

`BarrierValue` — Barrier level
scalar numeric | numeric vector

Barrier level, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Rebate` — Rebate value
`0` (default) | scalar numeric | numeric vector

Rebate value, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

Examples

Price Barrier Instrument Using Black-Scholes Model and Finite Difference Pricer

This example shows the workflow to price an `Barrier` instrument when you use a `BlackScholes` model and a `FiniteDifference` pricing method.

Create `Barrier` Instrument Object

Use `fininstrument` to create an `Barrier` instrument object.

```
BarrierOpt = fininstrument("Barrier",'Strike',45,'ExerciseDate',datetime(2019,1,1),'OptionType','
```

```
BarrierOpt =
  Barrier with properties:

       OptionType: "call"
           Strike: 45
      BarrierType: "do"
     BarrierValue: 40
           Rebate: 0
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2019
             Name: "barrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.30)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2023,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 1
              Dates: 01-Jan-2023
              Rates: 0.0350
             Settle: 01-Jan-2018
       InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create FiniteDifference Pricer Object

Use finpricer to create a FiniteDifference pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("FiniteDifference",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPri
```

```
outPricer =
  FiniteDifference with properties:

     DiscountCurve: [1x1 ratecurve]
             Model: [1x1 finmodel.BlackScholes]
         SpotPrice: 50
    GridProperties: [1x1 struct]
      DividendType: "continuous"
     DividendValue: 0
```

Price Barrier Instrument

Use price to compute the price and sensitivities for the Barrier instrument.

```
[Price, outPR] = price(outPricer,BarrierOpt,["all"])
```

```
Price = 8.5014

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Lambda     Theta      Rho       Vega
    ______    _______    _________    ______    _______    ______    ______

    8.5014    0.85673    0.0057199    5.0388    -1.8461    26.238    6.1837
```

Price Multiple Barrier Instruments Using Black-Scholes Model and Finite Difference Pricer

This example shows the workflow to price multiple Barrier instruments when you use a BlackScholes model and a FiniteDifference pricing method.

Create Barrier Instrument Object

Use fininstrument to create an Barrier instrument object for three Barrier instruments.

```
BarrierOpt = fininstrument("Barrier",'Strike',45,'ExerciseDate',datetime([2019,1,1; 2019,2,1 ; 20
```

```
BarrierOpt=3×1 object
  3x1 Barrier array with properties:

    OptionType
```

```
    Strike
    BarrierType
    BarrierValue
    Rebate
    ExerciseStyle
    ExerciseDate
    Name
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.30)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2023,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2023
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create FiniteDifference Pricer Object

Use `finpricer` to create a `FiniteDifference` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("FiniteDifference",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPric
```

```
outPricer =
  FiniteDifference with properties:

     DiscountCurve: [1x1 ratecurve]
             Model: [1x1 finmodel.BlackScholes]
         SpotPrice: 50
    GridProperties: [1x1 struct]
```

```
      DividendType: "continuous"
     DividendValue: 0
```

Price Barrier Instruments

Use `price` to compute the prices and sensitivities for the `Barrier` instruments.

```
[Price, outPR] = price(outPricer,BarrierOpt,["all"])
```

```
Price = 3×1

    8.5014
    9.7112
    9.9901


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Lambda     Theta      Rho       Vega
    _____    ________    _________    ______    ________    _____    ______

    8.5014    0.85673    0.0057199    5.0388    -1.8461    26.238    6.1837


ans=1×7 table
    Price      Delta       Gamma       Lambda     Theta      Rho       Vega
    _____    ________    _________    ______    ________    _____    ______

    9.7112    0.73186    0.020793     3.7681    -3.2754    29.014    16.885


ans=1×7 table
    Price      Delta       Gamma       Lambda     Theta      Rho       Vega
    _____    ________    _________    ______    ________    _____    ______

    9.9901    0.7296     0.020326     3.6516    -3.2151    30.872    17.803
```

Price Barrier Instrument Using Black-Scholes Model and Asset Tree Pricer for EQP Binomial Tree

This example shows the workflow to price an `Barrier` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method using an Equal Probability (EQP) tree.

Create Barrier Instrument Object

Use `fininstrument` to create an `Barrier` instrument object.

```
BarrierOpt = fininstrument("Barrier",'Strike',45,'ExerciseDate',datetime(2019,1,1),'OptionType','
```

```
BarrierOpt =
  Barrier with properties:

       OptionType: "call"
           Strike: 45
      BarrierType: "do"
     BarrierValue: 40
           Rebate: 0
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2019
             Name: "barrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.30)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2023,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2023
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetTree` Pricer Object

Use `finpricer` to create an `AssetTree` pricer object with an EQP equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
NumPeriods = 15;
EQPPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',1000
```

```
EQPPricer =
  EQPTree with properties:
```

```
            Tree: [1x1 struct]
       NumPeriods: 15
            Model: [1x1 finmodel.BlackScholes]
    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 1000
     DividendType: "continuous"
    DividendValue: 0
        TreeDates: [25-Jan-2018 08:00:00    18-Feb-2018 16:00:00    ...    ]
```

```
EQPPricer.Tree
```

```
ans = struct with fields:
    Probs: [2x15 double]
    ATree: {1x16 cell}
     dObs: [01-Jan-2018 00:00:00    25-Jan-2018 08:00:00    ...    ]
     tObs: [0 0.0667 0.1333 0.2000 0.2667 0.3333 0.4000 0.4667 0.5333 ... ]
```

Price `Barrier` Instrument

Use `price` to compute the price and sensitivities for the `Barrier` instrument.

```
[Price, outPR] = price(EQPPricer,BarrierOpt,["all"])
```

```
Price = 956.5478
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta      Gamma          Vega         Lambda     Rho      Theta
    ______    _____    __________    ___________    ______    _____    _______

    956.55      1      9.3133e-18    -6.8212e-09    1.0454    43.45    -1.5208
```

```
outPR.PricerData.PriceTree
```

```
ans = struct with fields:
     PTree: {1x16 cell}
    ExTree: {1x16 cell}
      tObs: [0 0.0667 0.1333 0.2000 0.2667 0.3333 0.4000 0.4667 0.5333 ... ]
      dObs: [01-Jan-2018    25-Jan-2018    18-Feb-2018    ...    ]
     Probs: [2x15 double]
```

Price Barrier Instrument Using Black-Scholes Model and Asset Tree Pricer for Standard Trinomial Tree

This example shows the workflow to price an `Barrier` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method using a Standard Trinomial (STT) tree.

Create `Barrier` Instrument Object

Use `fininstrument` to create an `Barrier` instrument object.

```
BarrierOpt = fininstrument("Barrier",'Strike',45,'ExerciseDate',datetime(2019,1,1),'OptionType','
```

```
BarrierOpt =
  Barrier with properties:

       OptionType: "call"
           Strike: 45
      BarrierType: "do"
     BarrierValue: 40
           Rebate: 0
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2019
             Name: "barrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.30)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2023,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 1
             Dates: 01-Jan-2023
             Rates: 0.0350
            Settle: 01-Jan-2018
      InterpMethod: "linear"
  ShortExtrapMethod: "next"
```

```
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object with a Standard Trinomial (STT) equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
NumPeriods = 15;
STTPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',1000
```

```
STTPricer =
  STTree with properties:

             Tree: [1x1 struct]
       NumPeriods: 15
            Model: [1x1 finmodel.BlackScholes]
    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 1000
     DividendType: "continuous"
    DividendValue: 0
        TreeDates: [25-Jan-2018 08:00:00    18-Feb-2018 16:00:00    ...    ]
```

```
STTPricer.Tree
```

```
ans = struct with fields:
    ATree: {1x16 cell}
    Probs: {1x15 cell}
     dObs: [01-Jan-2018 00:00:00    25-Jan-2018 08:00:00    ...    ]
     tObs: [0 0.0667 0.1333 0.2000 0.2667 0.3333 0.4000 0.4667 0.5333 ... ]
```

Price Barrier Instrument

Use `price` to compute the price and sensitivities for the `Barrier` instrument.

```
[Price, outPR] = price(STTPricer,BarrierOpt,["all"])
```

```
Price = 956.5444
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta       Gamma          Vega       Lambda     Rho       Theta
    ______    _____    ___________    _________    ______    ______    _______

    956.54      1      -1.9331e-17    -0.20023    1.0454    44.112    -1.514
```

```
outPR.PricerData.PriceTree
```

```
ans = struct with fields:
     PTree: {1x16 cell}
```

```
    ExTree: {1x16 cell}
      tObs: [0 0.0667 0.1333 0.2000 0.2667 0.3333 0.4000 0.4667 0.5333 ... ]
      dObs: [01-Jan-2018    25-Jan-2018    18-Feb-2018    ...    ]
     Probs: {1x15 cell}
```

Price Barrier Instrument Using Black-Scholes Model and Asset Monte-Carlo Pricer

This example shows the workflow to price an `Barrier` instrument when you use a `BlackScholes` model and a `AssetMonteCarlo` pricing method.

Create `Barrier` Instrument Object

Use `fininstrument` to create an `Barrier` instrument object.

```
BarrierOpt = fininstrument("Barrier",'Strike',45,'ExerciseDate',datetime(2019,1,1),'OptionType','
```

```
BarrierOpt =
  Barrier with properties:

       OptionType: "call"
           Strike: 45
      BarrierType: "do"
     BarrierValue: 40
           Rebate: 0
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2019
             Name: "barrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.30)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2023,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
          Compounding: -1
```

```
                Basis: 1
                Dates: 01-Jan-2023
                Rates: 0.0350
               Settle: 01-Jan-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BlackScholesModel,'SpotPrice
```

```
outPricer =
  GBMMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 200
    SimulationDates: 01-Jan-2019
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.BlackScholes]
       DividendType: "continuous"
      DividendValue: 0
```

Price Barrier Instrument

Use `price` to compute the price and sensitivities for the `Barrier` instrument.

```
[Price, outPR] = price(outPricer,BarrierOpt,["all"])
```

```
Price = 156.6270
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta        Gamma        Lambda     Rho     Theta     Vega
    ______    ______    ___________    ______    _____    _____    _______

    156.63    1.0004    -7.6028e-12    1.2774    43.45      0      0.67904
```

Price Barrier Instrument Using Heston Model and Asset Monte-Carlo Pricer

This example shows the workflow to price an `Barrier` instrument when you use a `Heston` model and an `AssetMonteCarlo` pricing method.

Create `Barrier` Instrument Object

Use `fininstrument` to create an `Barrier` instrument object.

```
BarrierOpt = fininstrument("Barrier",'Strike',45,'ExerciseDate',datetime(2019,1,1),'OptionType','
```

```
BarrierOpt =
  Barrier with properties:

       OptionType: "call"
           Strike: 45
      BarrierType: "do"
     BarrierValue: 40
           Rebate: 0
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2019
             Name: "barrier_option"
```

Create `Heston` Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9)
```

```
HestonModel =
  Heston with properties:

         V0: 0.0320
    ThetaV: 0.1000
     Kappa: 0.0030
    SigmaV: 0.2000
     RhoSV: 0.9000
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2023,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2023
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use finpricer to create an AssetMonteCarlo pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",HestonModel,'SpotPrice',200
```

```
outPricer =
  HestonMonteCarlo with properties:

     DiscountCurve: [1x1 ratecurve]
         SpotPrice: 200
    SimulationDates: 01-Jan-2019
         NumTrials: 1000
     RandomNumbers: []
             Model: [1x1 finmodel.Heston]
      DividendType: "continuous"
     DividendValue: 0
```

Price Barrier Instrument

Use price to compute the price and sensitivities for the Barrier instrument.

```
[Price, outPR] = price(outPricer,BarrierOpt,["all"])
```

```
Price = 156.9962

outPR =
  priceresult with properties:

       Results: [1x8 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×8 table
    Price    Delta       Gamma       Lambda     Rho     Theta     Vega      VegaLT
    _____    _____    __________    ______    _____    _____    ______    _________

     157     1.0022    -1.08e-12     1.2768    43.45      0      2.7882    0.0013677
```

More About

Barrier Option

A barrier option has not only a strike price but also a barrier level and sometimes a rebate.

The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by BarrierValue, during the life of the option. If the option cannot be exercised because the barrier level either has or has not been reached, a fixed rebate amount is paid. For more information, see “Barrier Option” on page 3-20.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Barrier` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`DoubleBarrier` | `finmodel` | `finpricer`

Topics

“Use Deep Learning to Approximate Barrier Option Prices with Heston Model” on page 3-148
“Calibrate Option Pricing Model Using Heston Model” on page 3-143
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

DoubleBarrier

`DoubleBarrier` instrument object

Description

Create and price a `DoubleBarrier` instrument object for one of more Double Barrier instruments using this workflow:

1. Use `fininstrument` to create a `DoubleBarrier` instrument object for one of more Double Barrier instruments.
2. Use `finmodel` to specify a `BlackScholes`, `Heston`, `Bates`, or `Merton` model for the `DoubleBarrier` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify an `IkedaKunitomo` or `VannaVolga` pricing method for one or more `DoubleBarrier` instruments.
   - When using a `BlackScholes`, `Heston`, `Bates`, or `Merton` model, use `finpricer` to specify a `FiniteDifference` or an `AssetMonteCarlo` pricing method for one or more `DoubleBarrier` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `DoubleBarrier` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
DoubleBarrierOpt = fininstrument(InstrumentType,'Strike',strike_value,'
ExerciseDate',exercise_date,'BarrierValue',barrier_value)
DoubleBarrierOpt = fininstrument( ___ ,Name,Value)
```

Description

`DoubleBarrierOpt = fininstrument(InstrumentType,'Strike',strike_value,'ExerciseDate',exercise_date,'BarrierValue',barrier_value)` creates a `DoubleBarrier` instrument object for one of more Double Barrier instruments by specifying `InstrumentType` and sets properties on page 11-2657 using the required name-value pair arguments `Strike`, `ExerciseDate`, and `BarrierValue`.

`DoubleBarrierOpt = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2657 using additional name-value pair arguments in addition to the required arguments in the previous syntax. For example, `DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',100,'ExerciseDate',datetime(2019,1,30),'BarrierValue',110,'OptionType',"put",'ExerciseStyle',"European",'BarrierTyp`

e',"DKI",'Name',"doublebarrier_option") creates a DoubleBarrier put option with a European exercise. You can specify multiple name-value pair arguments.

Input Arguments

InstrumentType — Instrument type

string with value "DoubleBarrier" | string array with values of "DoubleBarrier" | character vector with value 'DoubleBarrier' | cell array of character vectors with values of 'DoubleBarrier'

Instrument type, specified as a string with the value of "DoubleBarrier", a character vector with the value of 'DoubleBarrier', an NINST-by-1 string array with values of "DoubleBarrier", or an NINST-by-1 cell array of character vectors with values of 'DoubleBarrier'.

Data Types: char | cell | string

DoubleBarrier Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',100,'ExerciseDate',datetime(2019,1,30),'BarrierValue',110,'OptionType',"put",'ExerciseStyle',"European",'BarrierType',"DKI",'Name',"doublebarrier_option")

Required DoubleBarrier Name-Value Pair Arguments

Strike — Option strike value

nonnegative value | vector of nonnegative values

Option strike price value, specified as the comma-separated pair consisting of 'Strike' and a scalar nonnegative value or an NINST-by-1 vector of nonnegative values.

Data Types: double

ExerciseDate — Option exercise date

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of 'ExerciseDate' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one ExerciseDate value on the option expiry date.

To support existing code, DoubleBarrier also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by datetime because the ExerciseDate property is stored as a datetime.

BarrierValue — Double barrier value

numeric

Double barrier value, specified as the comma-separated pair consisting of 'BarrierValue' and an NINST-by-1 matrix of numeric values, where each element is a 1-by-2 vector where the first column is Barrier(1)(UB) and the second column is Barrier(2)(LB). Barrier(1) must be greater than Barrier(2).

Data Types: double

Optional DoubleBarrier Name-Value Pair Arguments

OptionType — Option type

"call" (default) | string with value "call" or "put" | string array with values "call" or "put" | character vector with value 'call' or 'put' | cell array of character vectors with values 'call' or 'put'

Option type, specified as the comma-separated pair consisting of 'OptionType' and a scalar string or character vector or an NINST-by-1 cell array of character vectors or string array.

Data Types: char | cell | string

ExerciseStyle — Option exercise style

"European" (default) | string with value "European" or "American" | string array with values "European" or "American" | character vector with value 'European' or 'American' | cell array of character vectors with values 'European' or 'American'

Option exercise style, specified as the comma-separated pair consisting of 'ExerciseStyle' and a scalar string or character vector or an NINST-by-1 cell array of character vectors or string array.

Note For a DoubleBarrier option, the IkedaKunitomo pricer supports only a "European" exercise and the FiniteDifference pricer supports an "American" or "European" exercise.

Data Types: string | cell | char

BarrierType — Double barrier type

"DKO" (default) | string with value of "DKI" or "DKO" | string array with values of "DKI" or "DKO" | character vector with value of 'DKI' or 'DKO' | cell array of character vectors with values of 'DKI' or 'DKO'

Double barrier type, specified as the comma-separated pair consisting of 'BarrierType' and a scalar character vector or string or an NINST-by-1 cell array of character vectors or string array with one of the following values:

- 'DKI' — Double knock-in

  The 'DKI' option becomes effective when the price of the underlying asset reaches one of the barriers. It gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, if the underlying asset goes above or below the barrier levels during the life of the option.

- 'DKO' — Double knock-out

  The 'DKO' option gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, as long as the underlying asset remains between the barrier levels during the life of the option. This option terminates when the price of the underlying asset passes one of the barriers.

Option	Barrier Type	Payoff If Any Barrier Crossed	Payoff If Barriers Not Crossed
Call/Put	Double Knock-in	Standard Call/Put	Worthless
Call/Put	Double Knock-out	Worthless	Standard Call/Put

Data Types: `char` | `cell` | `string`

`Rebate` — Barrier rebate
`[0 0]` (default) | numeric

Barrier rebate, specified as the comma-separated pair consisting of `'Rebate'` and a numeric matrix.

- For knock-in options, the `Rebate` is paid at expiry.
- For knock-out options, the `Rebate` is paid if the Upper Barrier(1)(UB) is hit and the second value is paid if the Lower Barrier(2)(LB) is hit.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for the instrument, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Strike` — Option strike price value
nonnegative value | vector of nonnegative values

Option strike price value, returned as a scalar nonnegative value or an `NINST`-by-`1` vector of nonnegative values.

Data Types: `double`

`ExerciseDate` — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`BarrierValue` — Double barrier value
numeric

Double barrier value, returned as a numeric matrix.

Data Types: `double`

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values `"call"` or `"put"`

Option type, returned as a scalar string or an NINST-by-1 string array with the values `"call"` or `"put"`.

Data Types: `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` or `"American"` | string array with values `"European"` or `"American"`

Option exercise style, returned as a scalar string or an NINST-by-1 string array with the values of `"European"` or `"American"`.

Data Types: `string`

`BarrierType` — Double barrier type
`"DKO"` (default) | string with value of `"DKI"` or `"DKO"` | string array with values of `"DKI"` or `"DKO"`

Double barrier type, returned as a scalar string or an NINST-by-1 string array with the values of `"DKI"` or `"DKO"`.

Data Types: `string`

`Rebate` — Barrier rebate
`[0 0]` (default) | numeric

Barrier rebate, returned as a numeric matrix.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an NINST-by-1 string array.

Data Types: `string`

Examples

Price Double Barrier Instrument Using Black-Scholes Model and Finite Difference Pricer

This example shows the workflow to price an `DoubleBarrier` instrument when you use a `BlackScholes` model and a `FiniteDifference` pricing method.

Create `DoubleBarrier` Instrument Object

Use `fininstrument` to create a `DoubleBarrier` instrument object.

```
DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',75,'ExerciseDate',datetime(2019,1,1),'
```

```
DoubleBarrierOpt =
  DoubleBarrier with properties:

       OptionType: "call"
           Strike: 75
     BarrierValue: [110 80]
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2019
```

```
      BarrierType: "dko"
           Rebate: [0 0]
             Name: "doublebarrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.30)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2023,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2023
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `FiniteDifference` Pricer Object

Use `finpricer` to create a `FiniteDifference` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("FiniteDifference",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPric
```

```
outPricer =
  FiniteDifference with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    GridProperties: [1x1 struct]
      DividendType: "continuous"
     DividendValue: 0
```

Price DoubleBarrier Instrument

Use `price` to compute the price and sensitivities for the `DoubleBarrier` instrument.

```
[Price, outPR] = price(outPricer,DoubleBarrierOpt,["all"])
```

```
Price = 25

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price    Delta    Gamma    Lambda      Theta       Rho    Vega
    _____    _____    _____    ______    __________    ___    ____

     25        1        0        4       2.2737e-13     0      0
```

Price Multiple Double Barrier Instruments Using Black-Scholes Model and Finite Difference Pricer

This example shows the workflow to price multiple `DoubleBarrier` instruments when you use a `BlackScholes` model and a `FiniteDifference` pricing method.

Create DoubleBarrier Instrument Object

Use `fininstrument` to create a `DoubleBarrier` instrument object for three Double Barrier instruments.

```
DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',[75 ; 85 ; 95],'ExerciseDate',datetime
```

```
DoubleBarrierOpt=3×1 object
  3x1 DoubleBarrier array with properties:

    OptionType
    Strike
    BarrierValue
    ExerciseStyle
    ExerciseDate
    BarrierType
    Rebate
    Name
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.30)
```

```
BlackScholesModel =
  BlackScholes with properties:
```

```
     Volatility: 0.3000
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2023,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
               Basis: 1
               Dates: 01-Jan-2023
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create FiniteDifference Pricer Object

Use `finpricer` to create a `FiniteDifference` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("FiniteDifference",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPric
```

```
outPricer =
  FiniteDifference with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    GridProperties: [1x1 struct]
      DividendType: "continuous"
     DividendValue: 0
```

Price DoubleBarrier Instruments

Use `price` to compute the prices and sensitivities for the `DoubleBarrier` instruments.

```
[Price, outPR] = price(outPricer,DoubleBarrierOpt,["all"])
```

```
Price = 3×1

   25.0000
   15.6821
    7.8957
```

```
outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price    Delta    Gamma    Lambda      Theta       Rho    Vega
    _____    _____    _____    _______    __________    ___    ____

     25        1        0        4       2.2737e-13     0      0
```

```
ans=1×7 table
    Price     Delta      Gamma      Lambda     Theta      Rho      Vega
    ______    ______    _______    ______    _______    _____    _______

    15.682    0.7196    0.28626    4.5887    0.88484    6.467    -6.3778
```

```
ans=1×7 table
    Price      Delta       Gamma       Lambda      Theta       Rho       Vega
    ______    _______    __________    ______    _________    ______    _______

    7.8957    0.36913    -0.0020435    4.675     -0.057311    4.3022    -6.9367
```

Price Double Barrier Instrument Using a Heston Model and Asset Monte Carlo Pricer

This example shows the workflow to price a `DoubleBarrier` instrument when you use a `Heston` model and an `AssetMonteCarlo` pricing method.

Create `DoubleBarrier` Instrument Object

Use `fininstrument` to create a `DoubleBarrier` instrument object.

```
DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',75,'ExerciseDate',datetime(2020,9,15),
```

```
DoubleBarrierOpt =
  DoubleBarrier with properties:

       OptionType: "call"
           Strike: 75
     BarrierValue: [110 80]
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2020
      BarrierType: "dko"
           Rebate: [0 0]
             Name: "doublebarrier_option"
```

Create `Heston` Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9)
```

```
HestonModel =
  Heston with properties:

       V0: 0.0320
    ThetaV: 0.1000
     Kappa: 0.0030
    SigmaV: 0.2000
     RhoSV: 0.9000
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
              Dates: 15-Sep-2023
              Rates: 0.0350
             Settle: 15-Sep-2018
       InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",HestonModel,'SpotPrice',102
```

```
outPricer =
  HestonMonteCarlo with properties:

     DiscountCurve: [1x1 ratecurve]
         SpotPrice: 102
    SimulationDates: 15-Sep-2020
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Heston]
       DividendType: "continuous"
      DividendValue: 0
```

Price `DoubleBarrier` Instrument

Use `price` to compute the price and sensitivities for the `DoubleBarrier` instrument.

```
[Price, outPR] = price(outPricer,DoubleBarrierOpt,["all"])

Price = 32.6351

outPR =
  priceresult with properties:

      Results: [1x8 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×8 table
    Price        Delta          Gamma          Lambda         Rho       Theta      Vega        VegaLT
    _____    ___________    ___________    ___________    _______    ______    _______    ________

    32.635    -0.00089196    -0.0025511    -0.0027878    -76.828    1.1334    -0.2616    -0.00298
```

Price Double Barrier Instrument Using a Black-Scholes Model and Asset Monte Carlo Pricer

This example shows the workflow to price a `DoubleBarrier` instrument when you use a `BlackScholes` model and an `AssetMonteCarlo` pricing method.

Create `DoubleBarrier` Instrument Object

Use `fininstrument` to create a `DoubleBarrier` instrument object.

```
DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',100,'ExerciseDate',datetime(2020,8,15),

DoubleBarrierOpt =
  DoubleBarrier with properties:

       OptionType: "call"
           Strike: 100
     BarrierValue: [110 80]
    ExerciseStyle: "american"
     ExerciseDate: 15-Aug-2020
      BarrierType: "dko"
           Rebate: [0 0]
             Name: "doublebarrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes","Volatility",.3)

BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create ratecurve Object

Create a flat ratecurve object using ratecurve.

```
Settle = datetime(2017,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2017
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use finpricer to create an AssetMonteCarlo pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
ExerciseDate = datetime(2020,08,15);
Settle = datetime(2017,09,15);
outPricer = finpricer("AssetMonteCarlo","DiscountCurve",myRC,"Model",BlackScholesModel,'SpotPrice
```

Price DoubleBarrier Instrument

Use price to compute the price and sensitivities for the DoubleBarrier instrument.

```
[Price, outPR] = price(outPricer,DoubleBarrierOpt,["all"])
```

```
Price = 6.9667

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Lambda      Rho       Theta      Vega
    ______    _______    _________    ______    _______    ______    _______

    6.9667    0.26875    -0.096337    3.8576    0.39855    9.5406    -1.2907
```

Price DoubleBarrier Instrument Using Black-Scholes Model and IkedaKunitomo Pricer

This example shows the workflow to price a `DoubleBarrier` instrument when you use a `BlackScholes` model and an `IkedaKunitomo` pricing method.

Create `DoubleBarrier` Instrument Object

Use `fininstrument` to create a `DoubleBarrier` instrument object.

```
DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',100,'ExerciseDate',datetime(2020,8,15),
```

```
DoubleBarrierOpt =
  DoubleBarrier with properties:

      OptionType: "call"
          Strike: 100
     BarrierValue: [110 80]
    ExerciseStyle: "european"
     ExerciseDate: 15-Aug-2020
      BarrierType: "dko"
           Rebate: [0 0]
             Name: "doublebarrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes","Volatility",.3)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2017,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2017
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IkedaKunitomo Pricer Object

Use `finpricer` to create an `IkedaKunitomo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Analytic","DiscountCurve",myRC,"Model",BlackScholesModel,'SpotPrice',100,
```

```
outPricer =
  IkedaKunitomo with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0290
     DividendType: "continuous"
        Curvature: [0.0300 -0.0300]
```

Price DoubleBarrier Instrument

Use `price` to compute the price and sensitivities for the `DoubleBarrier` instrument.

```
[Price, outPR] = price(outPricer,DoubleBarrierOpt,["all"])
```

```
Price = 5.6848e-04

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
      Price          Delta          Gamma        Lambda        Vega          Theta           Rho
    __________    ___________    ___________    _______    _________    _________    ____________

    0.00056848    -3.7713e-05    -4.2071e-06    -6.6339    -0.031332    0.0008912    -0.00035113
```

Price Double Barrier Instrument Using Black-Scholes Model and Vanna Volga Pricer

This example shows the workflow to price a `DoubleBarrier` instrument when you use a `BlackScholes` model and a `VannaVolga` pricing method.

Create DoubleBarrier Instrument Object

Use `fininstrument` to create a `DoubleBarrier` instrument object.

```
DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',100,'ExerciseDate',datetime(2020,8,15),
```

```
DoubleBarrierOpt =
  DoubleBarrier with properties:

       OptionType: "call"
```

```
          Strike: 100
     BarrierValue: [110 80]
    ExerciseStyle: "european"
     ExerciseDate: 15-Aug-2020
      BarrierType: "dko"
           Rebate: [0 0]
             Name: "doublebarrier_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes","Volatility",0.02)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.0200
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create VannaVolga Pricer Object

Use `finpricer` to create a `VannaVolga` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
VolRR = -0.0045;
VolBF = 0.0037;
RateF = 0.0210;
outPricer = finpricer("VannaVolga","DiscountCurve",myRC,"Model",BlackScholesModel,'SpotPrice',100
```

```
outPricer =
  VannaVolga with properties:

    DiscountCurve: [1x1 ratecurve]
```

```
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
     DividendType: "continuous"
    DividendValue: 0.0210
     VolatilityRR: -0.0045
     VolatilityBF: 0.0037
```

Price DoubleBarrier Instrument

Use `price` to compute the price and sensitivities for the `DoubleBarrier` instrument.

```
[Price, outPR] = price(outPricer,DoubleBarrierOpt,["all"])
```

```
Price = 1.6450
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta     Gamma     Lambda     Vega      Theta      Rho
    _____    _______    ______    ______    ______    _______    ______

    1.645    0.82818    75.662    50.346    14.697    -1.3145    74.666
```

More About

Double Barrier Option

A double barrier option is similar to the standard single barrier option except that they have two barrier levels: a lower barrier (LB) and an upper barrier (UB).

The payoff for a double barrier option depends on whether the underlying asset remains between the barrier levels during the life of the option. Double barrier options are less expensive than single barrier options as the probability of being knocked out is higher. Because of this, double barrier options allow investors to achieve reduction in the option premiums and match an investor’s belief about the future movement of the underlying price process.

There are two types of double barrier options:

- Double knock-in

  This option becomes effective when the price of the underlying asset reaches one of the barriers. It gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, if the underlying asset goes above or below the barrier levels during the life of the option.
- Double knock-out

  This option gives the option holder the right but not the obligation to buy or sell the underlying security at the strike price, as long as the underlying asset remains between the barrier levels

during the life of the option. This option terminates when the price of the underlying asset passes one of the barriers.

The payoff for this type of option depends on whether the underlying asset crosses the predetermined trigger value (barrier level), indicated by `BarrierValue`, during the life of the option. If the option cannot be exercised because the barrier level either has or has not been reached, a fixed rebate amount is paid. For more information, see “Double Barrier Option” on page 3-21.

Tips

After creating an `DoubleBarrier` instrument object with an `ExerciseStyle` set to `"American"`, you can modify the `ExerciseStyle` property to change it to `"European"` using dot notation.

```
DoubleBarrier.ExerciseStyle = "European"
```

Because a European option has a scalar `Strike` and `ExerciseDate` value and an American option has a 2-element vector for `Strike` and `ExerciseDate` values, when you change to `ExerciseStyle` from `"American"` to `"European"`, the `Strike` and `ExerciseDate` values become the last element in the 2-element vector for the `Strike` and `ExerciseDate` values.

Version History

Introduced in R2020b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `DoubleBarrier` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`Barrier` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

Touch

`Touch` instrument object

Description

Create and price a `Touch` instrument object for one or more Touch instruments using this workflow:

1. Use `fininstrument` to create a `Touch` instrument object for one or more Touch instruments.
2. Use `finmodel` to specify a `BlackScholes`, `Bates`, `Merton`, or `Heston` model for the `Touch` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify a `BlackScholes` or a `VannaVolga` pricing method for one or more `Barrier` instruments.
   - When using a `BlackScholes`, `Heston`, `Bates`, or `Merton` model, use `finpricer` to specify an `AssetMonteCarlo` pricing method for one or more `Touch` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Touch` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
TouchOpt = fininstrument(InstrumentType,'ExerciseDate',exercise_date,'
BarrierValue',barrier_value,'PayoffValue',payoff_value)
TouchOpt = fininstrument( ___ ,Name,Value)
```

Description

`TouchOpt = fininstrument(InstrumentType,'ExerciseDate',exercise_date,'BarrierValue',barrier_value,'PayoffValue',payoff_value)` creates a `Touch` instrument object for one or more Touch instruments by specifying `InstrumentType` and sets properties on page 11-2673 using the required name-value pair arguments `ExerciseDate`, `BarrierValue`, and `PayoffValue`.

`TouchOpt = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2673 using additional name-value pair arguments in addition to the required arguments in the previous syntax. For example, `TouchOpt = fininstrument("Touch",'ExerciseDate',datetime(2019,1,30),'BarrierValue',110,'PayoffValue',130,'BarrierType',"OT",'PayoffType',"Expiry",'Name',"Touch_option")` creates a `Touch` option with an expiry payoff type. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Touch"` | string array with values of `"Touch"` | character vector with value `'Touch'` | cell array of character vectors with values of `'Touch'`

Instrument type, specified as a string with the value of `"Touch"`, a character vector with the value of `'Touch'`, an `NINST`-by-`1` string array with values of `"Touch"`, or an `NINST`-by-`1` cell array of character vectors with values of `'Touch'`.

Data Types: `char` | `cell` | `string`

`Touch` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TouchOpt = fininstrument("Touch",'ExerciseDate',datetime(2019,1,30),'BarrierValue',110,'PayoffValue',130,'BarrierType',"OT",'PayoffType',"Expiry",'Name',"Touch_option")`

Required `Touch` Name-Value Pair Arguments

`ExerciseDate` — Option exercise date

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of `'ExerciseDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `Touch` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `ExerciseDate` property is stored as a datetime.

`BarrierValue` — Barrier level

scalar numeric | numeric vector

Barrier level, specified as the comma-separated pair consisting of `'BarrierValue'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`PayoffValue` — Option payoff value

scalar numeric | numeric vector

Option payoff value, specified as the comma-separated pair consisting of `'PayoffValue'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

Optional Touch Name-Value Pair Arguments

`BarrierType` — Barrier type
`"OT"` (default) | string with value `"OT"` or `"NT"` | string array with values `"OT"` or `"NT"` | character vector with value `'OT'` or `'NT'` | cell array of character vectors with values `'OT'` or `'NT'`

Barrier type, specified as the comma-separated pair consisting of `'BarrierType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array with one of the following values:

- `'OT'` — One-touch

  The one-touch option provides a payoff if the underlying asset ever trades at or beyond the `BarrierValue`. Otherwise, the `PayoffValue` is zero.
- `'NT'` — No-touch

  The no-touch option provides a payoff if the underlying asset never trades at or beyond the `BarrierValue`. Otherwise, the `PayoffValue` is zero.

Data Types: `char` | `cell` | `string`

`PayoffType` — Payoff type
`"Hit"` (default) | string with value `"Hit"` or `"Expiry"` | string array with values `"Hit"` or `"Expiry"` | character vector with value `'Hit'` or `'Expiry'` | cell array of character vectors with values `'Hit'` or `'Expiry'`

Payoff type, specified as the comma-separated pair consisting of `'PayoffType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. You can specify `"Expiry"` only when you specify `'OT'` as the `BarrierType`.

Note When you use a `BlackScholes` pricer, only the `"Hit"` `PayoffType` is supported.

Data Types: `char` | `cell` | `string`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`ExerciseDate` — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`BarrierValue` — Barrier level
scalar numeric | numeric vector

Barrier level, returned as a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

PayoffValue — Option payoff
scalar numeric | numeric vector

Option payoff, returned as a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

BarrierType — Barrier type
"OT" (default) | string with value "OT" or "NT" | string array with values "OT" or "NT"

Barrier type, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

PayoffType — Payoff type
"Hit" (default) | string with value "Hit" or "Expiry" | string array with values "Hit" or "Expiry"

Option type, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Name — User-defined name for instrument
" " (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Examples

Price Touch Instrument Using a Black-Scholes Model and Asset Monte Carlo Pricer

This example shows the workflow to price a Touch instrument when you use a BlackScholes model and an AssetMonteCarlo pricing method.

Create Touch Instrument Object

Use fininstrument to create a Touch instrument object.

```
TouchOpt = fininstrument("Touch",'ExerciseDate',datetime(2022,9,15),'BarrierValue',100,'PayoffVa
```

```
TouchOpt =
  Touch with properties:

    ExerciseDate: 15-Sep-2022
    BarrierValue: 100
     PayoffValue: 110
     BarrierType: "ot"
      PayoffType: "expiry"
            Name: "touch_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BlackScholesModel,'SpotPrice
```

```
outPricer =
  GBMMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 102
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.BlackScholes]
       DividendType: "continuous"
      DividendValue: 0
```

Price Touch Instrument

Use `price` to compute the price and sensitivities for the `Touch` instrument.

```
[Price, outPR] = price(outPricer,TouchOpt,["all"])

Price = 91.1862

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×7 table
    Price      Delta      Gamma      Lambda       Rho      Theta      Vega
    _____    ________    _________    _______    _______    _____    _____

    91.186    -2.1825    0.038281    -2.4413    -415.45    2.7374    35.998
```

Price Multiple Touch Instruments Using Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price multiple `Touch` instruments when you use a `BlackScholes` model and a `BlackScholes` pricing method.

Create Touch Instrument Object

Use `fininstrument` to create a `Touch` instrument object for three Touch instruments.

```
TouchOpt = fininstrument("Touch",'ExerciseDate',datetime([2022,9,15 ; 2022,10,15 ; 2022,11,15]),

TouchOpt=3×1 object
  3x1 Touch array with properties:

    ExerciseDate
    BarrierValue
    PayoffValue
    BarrierType
    PayoffType
    Name
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.28)

BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2800
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                 Type: "zero"
          Compounding: -1
                Basis: 12
                Dates: 15-Sep-2023
                Rates: 0.0350
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackScholes Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',135,
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 135
    DividendValue: 0.0450
     DividendType: "continuous"
```

Price Touch Instruments

Use `price` to compute the prices and sensitivities for the `Touch` instruments.

```
[Price, outPR] = price(outPricer,TouchOpt,["all"])
```

```
Price = 3×1

  136.5553
   99.8742
   63.6835


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta      Gamma      Lambda     Vega     Theta       Rho
```

```
    ______    ______    _________    ______    ______    ______    _______

    136.56    2.2346    0.005457    2.2092    30.812    3.9013    -465.89

ans=1×7 table
    Price     Delta      Gamma       Lambda     Vega       Theta         Rho
    ______    ______    _________    ______    ______    _________    _______

    99.874    1.8197    0.008319    2.4597    120.98    0.0043188    -138.47

ans=1×7 table
    Price     Delta       Gamma       Lambda     Vega      Theta       Rho
    ______    ______    _________    ______    ______    _______    ______

    63.683    1.3221    0.0099462    2.8028    182.58    -3.0963    72.793
```

Price Touch Instrument Using Heston Model and Asset Monte Carlo Pricer

This example shows the workflow to price a `Touch` instrument when you use a `Heston` model and an `AssetMonteCarlo` pricing method.

Create Touch Instrument Object

Use `fininstrument` to create a `Touch` instrument object.

```
TouchOpt = fininstrument("Touch",'ExerciseDate',datetime(2022,9,15),'BarrierValue',110,'PayoffVa
```

```
TouchOpt =
  Touch with properties:

    ExerciseDate: 15-Sep-2022
    BarrierValue: 110
     PayoffValue: 140
     BarrierType: "ot"
      PayoffType: "expiry"
            Name: "touch_option"
```

Create Heston Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9)
```

```
HestonModel =
  Heston with properties:

        V0: 0.0320
    ThetaV: 0.1000
     Kappa: 0.0030
    SigmaV: 0.2000
     RhoSV: 0.9000
```

Create ratecurve Object

Create a flat ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use finpricer to create an AssetMonteCarlo pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",HestonModel,'SpotPrice',112
```

```
outPricer =
  HestonMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 112
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Heston]
       DividendType: "continuous"
      DividendValue: 0
```

Price Touch Instrument

Use price to compute the price and sensitivities for the Touch instrument.

```
[Price, outPR] = price(outPricer,TouchOpt,["all"])
```

```
Price = 63.5247
```

```
outPR =
  priceresult with properties:

       Results: [1x8 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×8 table
    Price      Delta      Gamma      Lambda       Rho       Theta      Vega      VegaLT
    ______    ________    ______    ________    ________    ______    ______    ______

    63.525    -7.2363    1.0541    -12.758    -320.21    3.5527    418.94    8.1498
```

Price Touch Instrument Using Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price a `Touch` instrument when you use a `BlackScholes` model and a `BlackScholes` pricing method.

Create Touch Instrument Object

Use `fininstrument` to create a `Touch` instrument object.

```
TouchOpt = fininstrument("Touch",'ExerciseDate',datetime(2022,9,15),'BarrierValue',140,'PayoffVa
```

```
TouchOpt =
  Touch with properties:

    ExerciseDate: 15-Sep-2022
    BarrierValue: 140
     PayoffValue: 170
     BarrierType: "ot"
      PayoffType: "expiry"
            Name: "touch_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.28)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2800
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 12
```

```
                Dates: 15-Sep-2023
                Rates: 0.0350
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackScholes Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',135,
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 135
    DividendValue: 0.0450
     DividendType: "continuous"
```

Price Touch Instrument

Use `price` to compute the price and sensitivities for the `Touch` instrument.

```
[Price, outPR] = price(outPricer,TouchOpt,["all"])
```

```
Price = 136.5553

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta      Gamma      Lambda     Vega     Theta       Rho
    ______    ______    _________    ______    ______    ______    _______

    136.56    2.2346    0.005457    2.2092    30.812    3.9013    -465.89
```

More About

Touch Option

A touch option (also known as a binary barrier option or American digital) is a path-dependent option in which the existence and payment of the options depend on the movement of the underlying spot through their option life.

The one-touch (no-touch) option provides a payoff if the underlying spot ever (never) trades at or beyond the barrier level and otherwise it is zero. For more information, see “One-Touch and Double One-Touch Options” on page 3-30.

Version History

Introduced in R2020b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Touch` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`DoubleTouch` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

DoubleTouch

`DoubleTouch` instrument object

Description

Create and price a `DoubleTouch` instrument object for one of more Double Touch instruments using this workflow:

1. Use `fininstrument` to create a `DoubleTouch` instrument object for one of more Double Touch instruments.
2. Use `finmodel` to specify a `BlackScholes`, `Bates`, `Merton`, or `Heston` model for the `DoubleTouch` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify a `BlackScholes` or `VannaVolga` pricing method for one or more `DoubleTouch` instruments.
   - When using a `BlackScholes`, `Heston`, `Bates`, or `Merton` model, use `finpricer` to specify an `AssetMonteCarlo` pricing method for one or more `DoubleTouch` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `DoubleTouch` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
DoubleTouchOpt = fininstrument(InstrumentType,'ExerciseDate',exercise_date,'
BarrierValue',barrier_value,'PayoffValue',payoff_value)
DoubleTouchOpt = fininstrument( ___ ,Name,Value)
```

Description

`DoubleTouchOpt = fininstrument(InstrumentType,'ExerciseDate',exercise_date,'BarrierValue',barrier_value,'PayoffValue',payoff_value)` creates a `DoubleTouch` object for one of more Double Touch instruments by specifying `InstrumentType` and sets properties on page 11-2686 using the required name-value pair arguments `ExerciseDate`, `BarrierValue`, and `PayoffValue`.

`DoubleTouchOpt = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2686 using additional name-value pair arguments in addition to the required arguments in the previous syntax. For example, `DoubleTouchOpt = fininstrument("DoubleTouch",'Strike',100,'ExerciseDate',datetime(2019,1,30),'BarrierValue',110,'PayoffValue',150,'BarrierType',"DOT",'PayoffType',"Expiry"`

,'Name',"DoubleTouch_option") creates a DoubleTouch option with a payoff type of Expiry. You can specify multiple name-value pair arguments.

Input Arguments

InstrumentType — Instrument type

string with value "DoubleTouch" | string array with values of "DoubleTouch" | character vector with value 'DoubleTouch' | cell array of character vectors with values of 'DoubleTouch'

Instrument type, specified as a string with the value of "DoubleTouch", a character vector with the value of 'DoubleTouch', an NINST-by-1 string array with values of "DoubleTouch", or an NINST-by-1 cell array of character vectors with values of 'DoubleTouch'.

Data Types: char | cell | string

DoubleTouch Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: DoubleTouchOpt = fininstrument("DoubleTouch",'Strike',100,'ExerciseDate',datetime(2019,1,30),'BarrierValue',110,'OptionType',"put",'ExerciseStyle',"European",'BarrierType',"DO",'Name',"DoubleTouch_option")

Required DoubleTouch Name-Value Pair Arguments

ExerciseDate — Option exercise date

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of 'ExerciseDate' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, DoubleTouch also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by datetime because the ExerciseDate property is stored as a datetime.

BarrierValue — Option barrier levels

numeric

Option barrier levels, specified as the comma-separated pair consisting of 'BarrierValue' and an NINST-by-2 matrix of numeric values, where the first column is Upper Barrier(1)(UB) and the second column is Lower Barrier(2)(LB). Barrier(1) must be greater than Barrier(2).

Data Types: double

PayoffValue — Payoff value

numeric

Payoff value, specified as the comma-separated pair consisting of 'PayoffValue' and an NINST-by-1 matrix of numeric values, where each element is a 1-by-2 vector in which the first column is Barrier(1)(UB) and the second column is Barrier(2)(LB). Barrier(1) must be greater than Barrier(2).

Note The payoff value is calculated for the point in time that the `BarrierValue` is reached. The payoff is either cash or nothing. If you specify a double no-touch option using `BarrierType`, the payoff is at the maturity of the option.

Data Types: `double`

Optional DoubleTouch Name-Value Pair Arguments

`BarrierType` — Double barrier type

`"DOT"` (default) | string with value `"DOT"`, `"DNT"`, `"UNT-LOT"`, or `"UOT-LNT"` | string array with values `"DOT"`, `"DNT"`, `"UNT-LOT"`, or `"UOT-LNT"` | character vector with value `'DOT'`, `'DNT'`, `'UNT-LOT'`, or `'UOT-LNT'` | cell array of character vectors with values `'DOT'`, `'DNT'`, `'UNT-LOT'`, or `'UOT-LNT'`

Double barrier type, specified as the comma-separated pair consisting of `'BarrierType'` and a string or character vector or an `NINST`-by-`1` cell array of character vectors or string array with one of the following values:

- `'DOT'` — Double one-touch. The double one-touch option defines two `BarrierValue` values. A double one-touch option provides a `PayoffValue` if the underlying asset ever touches either the upper or lower `BarrierValue` values.
- `'DNT'` — Double no-touch. The double no-touch option defines two `BarrierValue` values. A double no-touch option provides a `PayoffValue` if the underlying asset ever never touches either the upper or lower `BarrierValue` values.
- `'UNT-LOT'` — Upper `BarrierValue` is No Touch and Lower `BarrierValue` is one Touch.
- `'UOT-LNT'` — Upper `BarrierValue` is One Touch and Lower `BarrierValue` is No Touch.

Data Types: `char` | `cell` | `string`

`PayoffType` — Payoff type

`"Hit"` (default) | string with value `"Hit"` or `"Expiry"` | string arrays with values `"Hit"` or `"Expiry"` | character vector with value `'Hit'` or `'Expiry'` | cell array of character vectors with values `'Hit'` or `'Expiry'`

Payoff type, specified as the comma-separated pair consisting of `'PayoffType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. You cannot use specify `"Expiry"` when using a `BarrierType` of `'DNT'`.

Note When you use a `BlackScholes` pricer, only the `"Expiry"` `PayoffType` is supported.

Data Types: `char` | `cell` | `string`

`Name` — User-defined name for instrument

`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for the instrument, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`ExerciseDate` — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`BarrierValue` — Barrier level
numeric

Barrier level, returned as a numeric matrix.

Data Types: `double`

`PayoffValue` — Option payoff
numeric

Option payoff, returned as a numeric matrix.

Data Types: `double`

`BarrierType` — Double barrier type
`"DOT"` (default) | string with value `"DOT"`, `"DNT"`, `"UNT-LOT"`, or `"UOT-LNT"` | string array with values `"DOT"`, `"DNT"`, `"UNT-LOT"`, or `"UOT-LNT"`

Double barrier type, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

`PayoffType` — Payoff type
`"Hit"` (default) | string with value `"Hit"` or `"Expiry"` | string array with values `"Hit"` or `"Expiry"`

Option type, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

Examples

Price Double Touch Instrument Using a Black-Scholes Model and Asset Monte Carlo Pricer

This example shows the workflow to price a `DoubleTouch` instrument when you use a `BlackScholes` model and an `AssetMonteCarlo` pricing method.

Create `DoubleTouch` Instrument Object

Use `fininstrument` to create a `DoubleTouch` instrument object.

```
DoubleTouchOpt = fininstrument("DoubleTouch",'ExerciseDate',datetime(2022,9,15),'BarrierValue',[1
```

```
DoubleTouchOpt =
  DoubleTouch with properties:

    ExerciseDate: 15-Sep-2022
    BarrierValue: [110 90]
     PayoffValue: 50
     BarrierType: "dot"
      PayoffType: "expiry"
            Name: "doubletouch_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BlackScholesModel,'SpotPrice
```

```
outPricer =
  GBMMonteCarlo with properties:
```

```
      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 102
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.BlackScholes]
       DividendType: "continuous"
      DividendValue: 0
```

Price DoubleTouch Instrument

Use `price` to compute the price and sensitivities for the `DoubleTouch` instrument.

```
[Price, outPR] = price(outPricer,DoubleTouchOpt,["all"])
```

```
Price = 43.3860

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price       Delta        Gamma       Lambda       Rho       Theta      Vega
    _____    _________    _________    ________    _______    ______    ______

    43.386    0.0043916    0.0018346    0.010325    -173.28    1.4722    1.8176
```

Price Multiple Double Touch Instruments Using a Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price multiple `DoubleTouch` instruments when you use a `BlackScholes` model and a `BlackScholes` pricing method.

Create DoubleTouch Instrument Object

Use `fininstrument` to create a `DoubleTouch` instrument object for three Double Touch instruments.

```
DoubleTouchOpt = fininstrument("DoubleTouch",'ExerciseDate',datetime([2022,9,15 ; 2022,10,15 ; 20
```

```
DoubleTouchOpt=3×1 object
  3x1 DoubleTouch array with properties:

    ExerciseDate
    BarrierValue
    PayoffValue
    BarrierType
    PayoffType
    Name
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.28)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2800
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
              Dates: 15-Sep-2023
              Rates: 0.0350
             Settle: 15-Sep-2018
       InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackScholes Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',100,
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0450
     DividendType: "continuous"
```

Price DoubleTouch Instruments

Use `price` to compute the prices and sensitivities for the `DoubleTouch` instruments.

```
[Price, outPR] = price(outPricer,DoubleTouchOpt,["all"])
```

```
Price = 3×1

   52.6903
   66.9920
   67.7447


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta       Gamma       Lambda      Vega      Theta      Rho
    _____    _______    __________    _______    _______    _____    _______

    52.69    -3.4708    -0.0041339    -6.5871    -1.3469      0      -35.883


ans=1×7 table
    Price      Delta       Gamma       Lambda      Vega      Theta      Rho
    ______    _______    _________    _______    _______    _____    _______

    66.992    -4.4128    -0.005258    -6.5871    -1.7125      0      -45.623


ans=1×7 table
    Price      Delta       Gamma       Lambda      Vega      Theta      Rho
    ______    _______    __________    _______    _______    _____    _______

    67.745    -4.4624    -0.0053149    -6.5871    -1.7318      0      -46.135
```

Price Double Touch Instrument Using a Bates Model and Asset Monte Carlo Pricer

This example shows the workflow to price a `DoubleTouch` instrument when you use a `Bates` model and an `AssetMonteCarlo` pricing method.

Create `DoubleTouch` Instrument Object

Use `fininstrument` to create a `DoubleTouch` instrument object.

```
DoubleTouchOpt = fininstrument("DoubleTouch",'ExerciseDate',datetime(2022,9,15),'BarrierValue',[1
```

```
DoubleTouchOpt =
  DoubleTouch with properties:

    ExerciseDate: 15-Sep-2022
    BarrierValue: [115 95]
     PayoffValue: 40
     BarrierType: "dot"
      PayoffType: "expiry"
            Name: "doubletouch_option"
```

Create Bates Model Object

Use `finmodel` to create a `Bates` model object.

```
BatesModel = finmodel("Bates",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9,'Mea
```

```
BatesModel =
  Bates with properties:

          V0: 0.0320
      ThetaV: 0.1000
       Kappa: 0.0030
      SigmaV: 0.2000
       RhoSV: 0.9000
       MeanJ: 0.1100
     JumpVol: 0.0230
    JumpFreq: 0.0200
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BatesModel,'SpotPrice',102,
```

```
outPricer =
  BatesMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 102
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Bates]
       DividendType: "continuous"
```

```
      DividendValue: 0
```

Price DoubleTouch Instrument

Use `price` to compute the price and sensitivities for the `DoubleTouch` instrument.

```
[Price, outPR] = price(outPricer,DoubleTouchOpt,["all"])
```

```
Price = 34.7743

outPR =
  priceresult with properties:

       Results: [1x8 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×8 table
    Price     Delta    Gamma    Lambda      Rho       Theta     Vega     VegaLT
    ______    _____    _____    ______    _______    ______    ____    ______

    34.774      0        0        0       -139.07    1.2179     0        0
```

Price Double Touch Instrument Using a Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price a `DoubleTouch` instrument when you use a `BlackScholes` model and a `BlackScholes` pricing method.

Create DoubleTouch Instrument Object

Use `fininstrument` to create a `DoubleTouch` instrument object.

```
DoubleTouchOpt = fininstrument("DoubleTouch",'ExerciseDate',datetime(2022,9,15),'BarrierValue',[1
```

```
DoubleTouchOpt =
  DoubleTouch with properties:

    ExerciseDate: 15-Sep-2022
    BarrierValue: [115 95]
     PayoffValue: 70
     BarrierType: "unt-lot"
      PayoffType: "expiry"
            Name: "doubletouch_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.28)
```

```
BlackScholesModel =
  BlackScholes with properties:
```

```
     Volatility: 0.2800
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackScholes Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',100,
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0450
     DividendType: "continuous"
```

Price DoubleTouch Instrument

Use `price` to compute the price and sensitivities for the `DoubleTouch` instrument.

```
[Price, outPR] = price(outPricer,DoubleTouchOpt,["all"])
```

```
Price = 52.6903
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta       Gamma        Lambda      Vega      Theta      Rho
    _____    ________    ___________    _______    _______    _____    ________

    52.69    -3.4708    -0.0041339    -6.5871    -1.3469      0       -35.883
```

More About

Double Touch Option

Double touch and double no-touch options work the same way as a `Touch` option, but have two barriers.

Double touch and double no-touch option provides a payoff if the underlying spot ever (never) touches either the upper or lower barriers levels. For more information, see “One-Touch and Double One-Touch Options” on page 3-30.

Version History

Introduced in R2020b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `DoubleTouch` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`Touch` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

Binary

`Binary` instrument object

Description

Create and price a `Binary` instrument object for one or more Binary instruments using this workflow:

1. Use `fininstrument` to create a `Binary` instrument object for one or more Binary instruments.
2. Use `finmodel` to specify a `BlackScholes` or `Bachelier` model for the `Binary` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify a `BlackScholes` or `AssetMonteCarlo` pricing method for one or more `Binary` instruments.
   - When using a `Bachelier` model, use `finpricer` to specify an `AssetMonteCarlo` pricing method for one or more `Binary` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Binary` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BinaryOpt = fininstrument(InstrumentType,'Strike',strike_value,'
ExerciseDate',exercise_date,'PayoffValue',payoff_value)
BinaryOpt = fininstrument( ___ ,Name,Value)
```

Description

`BinaryOpt = fininstrument(InstrumentType,'Strike',strike_value,'ExerciseDate',exercise_date,'PayoffValue',payoff_value)` creates a `Binary` instrument object for one or more Binary instruments by specifying `InstrumentType` and sets properties on page 11-2697 using the required name-value pair arguments `Strike`, `ExerciseDate`, and `PayoffValue`.

`BinaryOpt = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2697 using additional name-value pair arguments in addition to the required arguments in the previous syntax. For example, `BinaryOpt = fininstrument("Binary",'Strike',100,'ExerciseDate',datetime(2019,1,30),'PayoffValue',110,'OptionType',"put",'Name',"binary_option")` creates a `Binary` put option with a `PayoffValue` of 110. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Binary"` | string array with values of `"Binary"` | character vector with value `'Binary'` | cell array of character vectors with values of `'Binary'`

Instrument type, specified as a string with the value of `"Binary"`, a character vector with the value of `'Binary'`, an `NINST`-by-`1` string array with values of `"Binary"`, or an `NINST`-by-`1` cell array of character vectors with values of `'Binary'`.

Data Types: `char` | `cell` | `string`

`Binary` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BinaryOpt = fininstrument("Binary",'Strike',100,'ExerciseDate',datetime(2019,1,30),'PayoffValue',110,'OptionType',"put",'Name',"binary_option")`

Required Binary Name-Value Pair Arguments

`Strike` — Option strike price value

nonnegative value | vector of nonnegative values

Option strike price value, specified as the comma-separated pair consisting of `'Strike'` and a scalar nonnegative value or an `NINST`-by-`1` vector of nonnegative values.

Data Types: `double`

`ExerciseDate` — Option exercise date

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of `'ExerciseDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `Binary` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `ExerciseDate` property is stored as a datetime.

`PayoffValue` — Option payoff value

scalar numeric | numeric vector

Option payoff value, specified as the comma-separated pair consisting of `'PayoffValue'` and a scalar numeric value or an `NINST`-by-`1` numeric vector.

Data Types: `double`

Optional Binary Name-Value Pair Arguments

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Option type, specified as the comma-separated pair consisting of `'OptionType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` | string array with a value `"European"` | character vector with value `'European'` | cell array of character vectors with a value `'European'`

Option exercise style, specified as the comma-separated pair consisting of `'ExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `string` | `char` | `cell`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for the instrument, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Strike` — Option strike price value
nonnegative value | nonnegative value | vector of nonnegative values

Option strike price value, returned as a scalar nonnegative value or an `NINST`-by-`1` vector of nonnegative values.

Data Types: `double`

`ExerciseDate` — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`PayoffValue` — Option payoff value
scalar numeric | numeric vector

Option payoff value, returned as a scalar numeric value or an `NINST`-by-`1` vector of numeric values.

Data Types: `double`

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values `"call"` or `"put"`

Option type, returned as a scalar string or an NINST-by-1 string array with the values of "call" or "put".

Data Types: string

ExerciseStyle — Option exercise style
"European" (default) | string with value "European" | string array with values "European"

This property is read-only.

Option exercise style, returned as a scalar string or an NINST-by-1 string array with the value of "European".

Data Types: string

Name — User-defined name for instrument
" " (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Examples

Price Binary Instrument Using Black-Scholes Model and Asset Monte Carlo Pricer

This example shows the workflow to price a Binary instrument when you use a BlackScholes model and an AssetMonteCarlo pricing method.

Create Binary Instrument Object

Use fininstrument to create a Binary instrument object.

```
BinaryOpt = fininstrument("Binary",'ExerciseDate',datetime(2022,9,15),'Strike',1000,'PayoffValue
```

```
BinaryOpt =
  Binary with properties:

       OptionType: "put"
     ExerciseDate: 15-Sep-2022
           Strike: 1000
      PayoffValue: 130
    ExerciseStyle: "european"
             Name: "binary_option"
```

Create BlackScholes Model Object

Use finmodel to create a BlackScholes model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
```

```
    Correlation: 1
```

Create ratecurve Object

Create a flat ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use finpricer to create an AssetMonteCarlo pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BlackScholesModel,'SpotPrice
```

```
outPricer =
  GBMMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 102
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.BlackScholes]
       DividendType: "continuous"
      DividendValue: 0
```

Price Binary Instrument

Use price to compute the price and sensitivities for the Binary instrument.

```
[Price, outPR] = price(outPricer,BinaryOpt,["all"])
```

```
Price = 113.0166
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta     Gamma     Lambda      Rho       Theta     Vega
    ______    _____     _____     ______    _______    ______    ____

    113.02      0         0         0       -451.98    3.9582     0
```

Price Multiple Binary Instruments Using Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price multiple `Binary` instruments when you use a `BlackScholes` model and a `BlackScholes` pricing method.

Create `Binary` Instrument Object

Use `fininstrument` to create a `Binary` instrument object with three Binary instruments.

```
BinaryOpt = fininstrument("Binary",'ExerciseDate',datetime([2022,9,15 ; 2022,10,15 ; 2022,11,15])
```

```
BinaryOpt=3×1 object
  3x1 Binary array with properties:

    OptionType
    ExerciseDate
    Strike
    PayoffValue
    ExerciseStyle
    Name
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.28)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2800
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
```

```
          Compounding: -1
                Basis: 12
                Dates: 15-Sep-2023
                Rates: 0.0350
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackScholes Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',800,
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 800
    DividendValue: 0.0450
     DividendType: "continuous"
```

Price Binary Instruments

Use `price` to compute the prices and sensitivities for the `Binary` instruments.

```
[Price, outPR] = price(outPricer,BinaryOpt,["all"])
```

```
Price = 3×1

   87.4005
  109.9703
  111.9328


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta         Gamma        Lambda      Vega      Theta       Rho
    _____    _________    ___________    _______    _______    ______    _______

    87.4     -0.075973    -3.1264e-05    -0.6954    -23.084    3.2599    -592.61


ans=1×7 table
    Price       Delta         Gamma         Lambda       Vega       Theta       Rho
    ______    _________    ___________    ________    _______    ______    _______
```

```
    109.97    -0.014137    -4.4054e-05    -0.10284    -32.196    4.8405    -495.01

ans=1×7 table
    Price       Delta          Gamma         Lambda        Vega       Theta       Rho
    _____    __________    ___________    _________    _______    ______    _______

    111.93    -0.0027668    -1.279e-05    -0.019775    -9.4868    4.2144    -475.57
```

Price Binary Instrument Using Merton Model and Asset Monte Carlo Pricer

This example shows the workflow to price a `Binary` instrument when you use a `Merton` model and an `AssetMonteCarlo` pricing method.

Create `Binary` Instrument Object

Use `fininstrument` to create a `Binary` instrument object.

```
BinaryOpt = fininstrument("Binary",'ExerciseDate',datetime(2022,9,15),'Strike',1000,'PayoffValue
```

```
BinaryOpt =
  Binary with properties:

       OptionType: "put"
     ExerciseDate: 15-Sep-2022
           Strike: 1000
      PayoffValue: 130
    ExerciseStyle: "european"
             Name: "binary_option"
```

Create `Merton` Model Object

Use `finmodel` to create a `Merton` model object.

```
MertonModel = finmodel("Merton",'Volatility',0.45,'MeanJ',0.02,'JumpVol',0.07,'JumpFreq',0.09)
```

```
MertonModel =
  Merton with properties:

    Volatility: 0.4500
         MeanJ: 0.0200
       JumpVol: 0.0700
      JumpFreq: 0.0900
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",MertonModel,'SpotPrice',102
```

```
outPricer =
  MertonMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 102
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Merton]
       DividendType: "continuous"
      DividendValue: 0
```

Price `Binary` Instrument

Use `price` to compute the price and sensitivities for the `Binary` instrument.

```
[Price, outPR] = price(outPricer,BinaryOpt,["all"])
```

```
Price = 112.4515
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta     Gamma     Lambda      Rho       Theta      Vega
    _____     _____     _____     ______    _______     ______     ____

    112.45      0         0         0       -449.72     3.9384      0
```

Price Binary Instrument Using Bachelier Model and Asset Monte Carlo Pricer

This example shows the workflow to price a `Binary` instrument when you use a `Bachelier` model and an `AssetMonteCarlo` pricing method.

Create `Binary` Instrument Object

Use `fininstrument` to create a `Binary` instrument object.

```
BinaryOpt = fininstrument("Binary",'ExerciseDate',datetime(2022,9,15),'Strike',1000,'PayoffValue
```

```
BinaryOpt =
  Binary with properties:

       OptionType: "put"
     ExerciseDate: 15-Sep-2022
           Strike: 1000
      PayoffValue: 130
    ExerciseStyle: "european"
             Name: "binary_option"
```

Create `Bachelier` Model Object

Use `finmodel` to create a `Bachelier` model object.

```
BachelierModel = finmodel("Bachelier",'Volatility',.2)
```

```
BachelierModel =
  Bachelier with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                 Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BachelierModel,'SpotPrice',1
```

```
outPricer =
  BachelierMonteCarlo with properties:

     DiscountCurve: [1x1 ratecurve]
         SpotPrice: 102
    SimulationDates: 15-Sep-2022
         NumTrials: 1000
     RandomNumbers: []
             Model: [1x1 finmodel.Bachelier]
      DividendType: "continuous"
     DividendValue: 0
```

Price Binary Instrument

Use `price` to compute the price and sensitivities for the `Binary` instrument.

```
[Price, outPR] = price(outPricer,BinaryOpt,["all"])
```

```
Price = 113.0166
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta    Gamma    Lambda      Rho      Theta     Vega
    ______    _____    _____    ______    ________    ______    ____

    113.02      0        0        0       -451.98    3.9582     0
```

Price Binary Instrument Using Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price a `Binary` instrument when you use a `BlackScholes` model and a `BlackScholes` pricing method.

Create Binary Instrument Object

Use `fininstrument` to create a `Binary` instrument object.

```
BinaryOpt = fininstrument("Binary",'ExerciseDate',datetime(2022,9,15),'Strike',1000,'PayoffValue
```

```
BinaryOpt =
  Binary with properties:
```

```
       OptionType: "put"
     ExerciseDate: 15-Sep-2022
           Strike: 1000
      PayoffValue: 130
    ExerciseStyle: "european"
             Name: "binary_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.28)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2800
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackScholes Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',800,
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 800
    DividendValue: 0.0450
```

```
     DividendType: "continuous"
```

Price Binary Instrument

Use `price` to compute the price and sensitivities for the `Binary` instrument.

```
[Price, outPR] = price(outPricer,BinaryOpt,["all"])
```

```
Price = 87.4005
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta         Gamma        Lambda      Vega      Theta       Rho
    _____    _________    ___________    _______    _______    ______    _______

    87.4     -0.075973    -3.1264e-05    -0.6954    -23.084    3.2599    -592.61
```

More About

Binary Option

A binary option is where the buyer receives a payout or loses their investment, depending on whether the option expires in the money.

Binary options depend on the outcome of a "yes or no" proposition, hence the name "binary." Binary options have an expiry date and/or time. At the time of expiry, the price of the underlying asset must be on the correct side of the strike price (based on the trade taken) for the trader to make a profit.

A binary option automatically exercises, meaning the gain or loss on the trade is automatically credited or debited to the trader's account when the option expires.

Version History

Introduced in R2020b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Binary` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`finmodel` | `finpricer`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

Cap

`Cap` instrument object

Description

Create and price a `Cap` instrument object for one or more Cap instruments using this workflow:

1. Use `fininstrument` to create a `Cap` instrument object for one or more Cap instruments.
2. Use `finmodel` to specify a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `Black`, `Normal`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model for the `Cap` instrument object.
3. Choose a pricing method.
   - When using a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `Black`, or `Normal` model, use `finpricer` for pricing one or more `Cap` instruments and specify:
     - A `Normal` pricer when using a `Normal` model.
     - A `Black` pricer when using a `Black` model.
     - A `HullWhite` pricer when using a `HullWhite` model.
     - An `IRTree` pricer when using a `BlackKarasinski` or `BlackDermanToy` model.
   - When using a `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model, use `finpricer` to specify an `IRMonteCarlo` pricing method for one or more `Cap` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Cap` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
CapOpt = fininstrument(InstrumentType,'Strike',strike_value,'
Maturity',maturity_date)
CapOpt = fininstrument( ___ ,Name,Value)
```

Description

`CapOpt = fininstrument(InstrumentType,'Strike',strike_value,'Maturity',maturity_date)` creates a `Cap` object for one or more Cap instruments by specifying `InstrumentType` and sets the properties on page 11-2713 for the required name-value pair arguments `Strike` and `Maturity`.

The `Cap` instrument supports vanilla and amortizing caps.

`CapOpt = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2713 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `CapOpt = fininstrument("Cap",'Strike',0.65,'Maturity',datetime(2019,1,30),'Reset',4,'Principal',100,'ResetOffset',1,'Basis',1,'DaycountAdjustedCashFlow',true,'BusinessDayConvention',"follow",'ProjectionCurve',ratecurve_object,'Name',"cap_option")` creates a `Cap` option with a strike of 0.65. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Cap"` | string array with values of `"Cap"` | character vector with value `'Cap'` | cell array of character vectors with values of `'Cap'`

Instrument type, specified as a string with the value of `"Cap"`, a character vector with the value of `'Cap'`, an `NINST`-by-1 string array with values of `"Cap"`, or an `NINST`-by-1 cell array of character vectors with values of `'Cap'`.

Data Types: `char` | `cell` | `string`

Cap Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `CapOpt = fininstrument("Cap",'Strike',0.65,'Maturity',datetime(2019,1,30),'Reset',4,'Principal',100,'ResetOffset',1,'Basis',1,'DaycountAdjustedCashFlow',true,'BusinessDayConvention',"follow",'ProjectionCurve',ratecurve_object,'Name',"cap_option")`

Required Cap Name-Value Pair Arguments

`Strike` — Cap strike price

nonnegative decimal | vector of nonnegative decimals

Cap strike price, specified as the comma-separated pair consisting of `'Strike'` and a scalar nonnegative decimal value or an `NINST`-by-1 nonnegative numeric vector.

Data Types: `double`

`Maturity` — Cap maturity date

datetime array | string array | date character vector

Cap maturity date, specified as the comma-separated pair consisting of `'ExerciseDate'` and a scalar or an `NINST`-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, `Cap` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

Optional Cap Name-Value Pair Arguments

`Reset` — Reset frequency payments per year
`1` (default) | numeric with value of `0`, `1`, `2`, `3`, `4`, `6`, or `12` | numeric vector with values of `0`, `1`, `2`, `3`, `4`, `6`, or `12`

Reset frequency payments per year, specified as the comma-separated pair consisting of `'Reset'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer or an `NINST`-by-`1` vector of integers with the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Principal amount or principal value schedule, specified as the comma-separated pair consisting of `'Principal'` and a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

`Principal` accepts a `timetable`, where the first column is dates and the second column is its associated principal value. The date indicates the last day that the principal value is valid.

Note If you are creating one or more `Cap` instruments and use a timetable, the timetable specification applies to all of the `Cap` instruments. `Principal` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `double` | `timetable`

ResetOffset — Lag in rate setting
0 (default) | scalar numeric | numeric vector

Lag in rate setting, specified as the comma-separated pair consisting of 'ResetOffset' and a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

DaycountAdjustedCashFlow — Flag to adjust cash flows based on actual period day count
false (default) | value of true or false | vector of values of true or false

Flag to adjust cash flows based on the actual period day count, specified as the comma-separated pair consisting of 'DaycountAdjustedCashFlow' and a scalar or an NINST-by-1 vector with values of true or false.

Data Types: logical

BusinessDayConvention — Business day conventions
"actual" (default) | string | string array | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of 'BusinessDayConvention' and a scalar string or character vector or an NINST-by-1 cell array of character vectors or string array for a business day convention. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- "actual" — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- "follow" — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- "modifiedfollow" — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- "previous" — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- "modifiedprevious" — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: char | cell | string

Holidays — Holidays used in computing business days
NaT (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of 'Holidays' and dates using an NINST-by-1 vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
CapOpt = fininstrument("Cap",'Strike',100,'Maturity',datetime(2025,12,15),'Holidays',H)
```

To support existing code, Cap also accepts serial date numbers as inputs, but they are not recommended.

`ProjectionCurve` — Rate curve used in generating future cash flows
`ratecurve.empty` (default) | `ratecurve` object | vector of `ratecurve` objects

Rate curve used in projecting the future cash flows, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects. These objects must be created using `ratecurve`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `object`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Strike` — Option strike price value
nonnegative value | vector of nonnegative values

Option strike price value, returned as a scalar nonnegative value or an `NINST`-by-`1` vector of nonnegative values.

Data Types: `double`

`Maturity` — Cap maturity date
datetime | vector of datetimes

Cap maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Reset` — Reset frequency payments per year
`1` (default) | scalar numeric | numeric vector

Reset frequency payments per year, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Principal` — Principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Principal amount or principal value schedule, returned as a scalar numeric or an `NINST`-by-`1` numeric vector for principal amounts or a timetable for a principal value schedule.

Data Types: `double` | `timetable`

ResetOffset — Lag in rate setting
0 (default) | scalar numeric | numeric vector

Lag in rate setting, returned as a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

DaycountAdjustedCashFlow — Flag to adjust cash flows based on actual period day count
false (default) | value of true or false | vector of values of true or false

Flag to adjust cash flows based on the actual period day count, returned as a scalar logical or an NINST-by-1 vector with values of true or false.

Data Types: logical

BusinessDayConvention — Business day conventions
"actual" (default) | string | string array

Business day conventions, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Holidays — Holidays used in computing business days
NaT (default) | vector of datetimes

Holidays used in computing business days, returned as an NINST-by-1 vector of datetimes.

Data Types: datetime

ProjectionCurve — Rate curve used in generating future cash flows
ratecurve.empty (default) | ratecurve object | vector of ratecurve objects

Rate curve used in projecting the future cash flows, returned as a scalar ratecurve object or an NINST-by-1 vector of ratecurve objects.

Data Types: object

Name — User-defined name for instrument
" " (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Examples

Price Vanilla Cap Instrument Using Hull-White Model and Hull-White Pricer

This example shows the workflow to price a vanilla Cap instrument when using a HullWhite model and a HullWhite pricing method.

Create Cap Instrument Object

Use fininstrument to create a Cap instrument object.

```
CapOpt = fininstrument("Cap",'Strike',0.02,'Maturity',datetime(2019,1,30),'Reset',4,'Principal',1
```

```
CapOpt =
  Cap with properties:

                     Strike: 0.0200
                   Maturity: 30-Jan-2019
                ResetOffset: 0
                      Reset: 4
                      Basis: 8
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                        Name: "cap_option"
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.62,'Sigma',0.99)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.6200
    Sigma: 0.9900
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create HullWhite Pricer Object

Use `finpricer` to create a `HullWhite` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',HullWhiteModel,'DiscountCurve',myRC)

outPricer =
  HullWhite with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.HullWhite]
```

Price Cap Instrument

Use `price` to compute the price for the `Cap` instrument.

```
Price = price(outPricer,CapOpt)

Price = 2.9366
```

Price Multiple Vanilla Cap Instruments Using Hull-White Model and Hull-White Pricer

This example shows the workflow to price multiple vanilla `Cap` instruments when using a `HullWhite` model and a `HullWhite` pricing method.

Create Cap Instrument Object

Use `fininstrument` to create a `Cap` instrument object for three Cap instruments.

```
CapOpt = fininstrument("Cap",'Strike',0.02,'Maturity',datetime([2019,1,30 ; 2019,2,30 ; 2019,3,30

CapOpt=3×1 object
  3x1 Cap array with properties:

    Strike
    Maturity
    ResetOffset
    Reset
    Basis
    Principal
    ProjectionCurve
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    Name
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.62,'Sigma',0.99)

HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.6200
    Sigma: 0.9900
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create HullWhite Pricer Object

Use finpricer to create a HullWhite pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("analytic",'Model',HullWhiteModel,'DiscountCurve',myRC)
```

```
outPricer =
  HullWhite with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.HullWhite]
```

Price Cap Instruments

Use price to compute the prices for the Cap instruments.

```
Price = price(outPricer,CapOpt)
```

```
Price = 3×1

    2.9366
    7.4694
   17.7915
```

Price Vanilla Cap Instrument Using Normal Model and Normal Pricer

This example shows the workflow to price a vanilla Cap instrument when you use a Normal model and a Normal pricing method.

Create ratecurve Object

Create a ratecurve object using ratecurve for the underlying interest-rate curve for the cap instrument.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Cap Instrument Object

Use fininstrument to create a Cap instrument object.

```
CapOpt = fininstrument("Cap",'Maturity',datetime(2022,9,15),'Strike',0.04,'ProjectionCurve',myRC)
```

```
CapOpt =
  Cap with properties:

                    Strike: 0.0400
                  Maturity: 15-Sep-2022
               ResetOffset: 0
                     Reset: 1
                     Basis: 0
                 Principal: 100
            ProjectionCurve: [1x1 ratecurve]
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                  Holidays: NaT
                      Name: ""
```

Create Normal Model Object

Use finmodel to create a Normal model object.

```
NormalModel = finmodel("Normal",'Volatility',0.01)
```

```
NormalModel =
  Normal with properties:
```

```
    Volatility: 0.0100
```

Create Normal Pricer Object

Use `finpricer` to create a `Normal` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',NormalModel)
```

```
outPricer =
  Normal with properties:

    DiscountCurve: [1x1 ratecurve]
            Shift: 0
            Model: [1x1 finmodel.Normal]
```

Price Cap Instrument

Use `price` to compute the price for the `Cap` instrument.

```
[Price, outPR] = price(outPricer, CapOpt)
```

```
Price = 0.0701
```

```
outPR =
  priceresult with properties:

       Results: [1x1 table]
    PricerData: []
```

Price Amortizing Cap Instrument Using Black Model and Black Pricer

This example shows the workflow to price an amortizing `Cap` instrument when you use a `Black` model and a `Black` pricing method.

Create Cap Instrument Object

Use `fininstrument` to create an amortizing `Cap` instrument object.

```
CADates = [datetime(2020,9,1) ; datetime(2023,9,1)];
CAPrincipal = [100; 85];
Principal = timetable(CADates,CAPrincipal);

CapOpt = fininstrument("Cap",'Maturity',datetime(2023,9,1),'Strike',0.015,'Principal',Principal,
```

```
CapOpt =
  Cap with properties:

                    Strike: 0.0150
                  Maturity: 01-Sep-2023
               ResetOffset: 0
                     Reset: 1
                     Basis: 0
                 Principal: [2x1 timetable]
```

```
           ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                        Name: "cap_amortizing_option"
```

Create Black Model Object

Use `finmodel` to create a `Black` model object.

```
BlackModel = finmodel("Black",'Volatility',0.2)
```

```
BlackModel =
  Black with properties:

    Volatility: 0.2000
         Shift: 0
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,1);
Type = 'zero';
ZeroTimes = [calyears([1 2 3 4 5 7 10])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates);
```

Create Black Pricer Object

Use `finpricer` to create a `Black` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackModel,'DiscountCurve',myRC)
```

```
outPricer =
  Black with properties:

           Model: [1x1 finmodel.Black]
    DiscountCurve: [1x1 ratecurve]
```

Price Cap Instrument

Use `price` to compute the price for the `Cap` instrument.

```
Price = price(outPricer,CapOpt)
```

```
Price = 0.3897
```

Price Vanilla Cap Instrument Using Hull-White Model and IRTree Pricer

This example shows the workflow to price a vanilla `Cap` instrument when using a `HullWhite` model and an `IRTree` pricing method.

Create Cap Instrument Object

Use `fininstrument` to create a `Cap` instrument object.

```
CapOpt = fininstrument("Cap",'Strike',0.02,'Maturity',datetime(2020,1,30),'Reset',4,'Principal',1
```

```
CapOpt =
  Cap with properties:

                     Strike: 0.0200
                   Maturity: 30-Jan-2020
                ResetOffset: 0
                      Reset: 4
                      Basis: 8
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "cap_option"
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.01,'Sigma',0.10)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0100
    Sigma: 0.1000
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: [10x1 datetime]
             Rates: [10x1 double]
            Settle: 15-Sep-2018
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
```

```
     LongExtrapMethod: "previous"
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
CFdates = cfdates(Settle, CapOpt.Maturity, CapOpt.Reset, CapOpt.Basis);
outPricer = finpricer("IRTree",'Model',HullWhiteModel,'DiscountCurve',myRC,'TreeDates',CFdates')
```

```
outPricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [6x1 datetime]
           Model: [1x1 finmodel.HullWhite]
   DiscountCurve: [1x1 ratecurve]
```

Price Cap Instrument

Use `price` to compute the price and sensitivities for the `Cap` instrument.

```
[Price, outPR] = price(outPricer,CapOpt,["all"])
```

```
Price = 2.7733

outPR =
  priceresult with properties:

      Results: [1x4 table]
   PricerData: [1x1 struct]


outPR.Results

ans=1×4 table
    Price     Delta      Gamma      Vega
    ______    ______    _______    ______

    2.7733    28.932    -49.227    31.655
```

Price Cap Instrument Using LinearGaussian2F Model and IRMonteCarlo Pricer

This example shows the workflow to price a `Cap` instrument when using a `LinearGaussian2F` model and an `IRMonteCarlo` pricing method.

Create Cap Instrument Object

Use `fininstrument` to create a `Cap` instrument object.

```
CapOpt = fininstrument("Cap","Maturity",datetime(2022,9,15),'Strike',0.01,'Reset',2,'Name',"cap_
```

```
CapOpt =
  Cap with properties:
```

```
                    Strike: 0.0100
                  Maturity: 15-Sep-2022
               ResetOffset: 0
                     Reset: 2
                     Basis: 0
                 Principal: 100
            ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                  Holidays: NaT
                      Name: "cap_option"
```

Create LinearGaussian2F Model Object

Use `finmodel` to create a `LinearGaussian2F` model object.

```
LinearGaussian2FModel = finmodel("LinearGaussian2F",'Alpha1',0.07,'Sigma1',0.01,'Alpha2',0.5,'Sig
```

```
LinearGaussian2FModel =
  LinearGaussian2F with properties:

        Alpha1: 0.0700
        Sigma1: 0.0100
        Alpha2: 0.5000
        Sigma2: 0.0060
    Correlation: -0.7000
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
          Compounding: -1
                Basis: 0
                Dates: [10x1 datetime]
                Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",'Model',LinearGaussian2FModel,'DiscountCurve',myRC,'Simulati
```

```
outPricer =
  G2PPMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jan-2021    ...    ]
              Model: [1x1 finmodel.LinearGaussian2F]
```

Price Cap Instrument

Use `price` to compute the price and sensitivities for the `Cap` instrument.

```
[Price,outPR] = price(outPricer,CapOpt,["all"])
```

```
Price = 1.2156

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta     Gamma          Vega
    ______    ______    _____    _______________

    1.2156    131.37    11048    126.5    -157.38
```

More About

Cap

A cap is a contract that includes a guarantee that sets the maximum interest rate the holder pays, based on an otherwise floating interest rate.

The payoff for a cap is: max(*CurrentRate* − *CapRate*, 0)

For more information, see “Cap” on page 2-12.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Cap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`Floor` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Work with Negative Interest Rates Using Objects” on page 2-22

CDS

CDS instrument object

Description

Create and price a CDS instrument object for one or more CDS instruments using this workflow:

1. Use fininstrument to create a CDS instrument object for one or more CDS instruments.
2. Use defprobcurve to specify a default probability curve for the CDS instrument object.
3. Use finpricer to specify a Credit pricing method for one or more CDS instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a CDS instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
CDSobj = fininstrument(InstrumentType,'Maturity',maturity_date,'
ContractSpread',contractspread_value)
CDSobj = fininstrument( ___ ,Name,Value)
```

Description

`CDSobj = fininstrument(InstrumentType,'Maturity',maturity_date,'ContractSpread',contractspread_value)` creates a CDS object for one or more CDS instruments by specifying InstrumentType and sets the properties on page 11-2729 for the required name-value pair arguments Maturity and ContractSpread.

`CDSobj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2729 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `CDSobj = fininstrument("CDS",'Maturity',datetime(2019,1,30),'ContractSpread',200,'Period',4,'Basis',5,'BusinessDayConvention','follow','Name',"cds_instrument")` creates a CDS instrument with contract spread of 200. You can specify multiple name-value pair arguments.

Input Arguments

InstrumentType — Instrument type

string with value "CDS" | string array with values of "CDS" | character vector with value 'CDS' | cell array of character vectors with values of 'CDS'

Instrument type, specified as a string with the value of "CDS", a character vector with the value of 'CDS', an NINST-by-1 string array with values of "CDS", or an NINST-by-1 cell array of character vectors with values of 'CDS'.

Data Types: char | cell | string

CDS Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: CDSobj = fininstrument("CDS",'Maturity',datetime(2019,1,30),'ContractSpread',200,'Period',4,'Basis',5,'BusinessDayConvention',"follow",'Name',"cds_instrument")

Required CDS Name-Value Pair Arguments

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date, specified as the comma-separated pair consisting of 'Maturity' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, CDS also accepts serial date numbers as inputs, but they are not recommended.

If you use date characters vector or strings, the format must be recognizable by datetime because the Maturity property is stored as a datetime.

ContractSpread — Contract spreads expressed in basis points
scalar numeric | numeric vector

Contract spreads expressed in basis points, specified as the comma-separated pair consisting of 'ContractSpread' and a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

Optional CDS Name-Value Pair Argument

Period — Premium payments per year
4 (default) | scalar numeric with value of 1, 2, 3, 4, 6, 12 | numeric vector with values of 1, 2, 3, 4, 6, 12

Premium payments per year, specified as the comma-separated pair consisting of 'Period' and a scalar numeric or an NINST-by-1 numeric vector with values of 1, 2, 3, 4, 6, or 12.

Data Types: double

Basis — Day count basis
0 (actual/actual) (default) | scalar of positive integers of the set [1...13] | vector of positive integers of the set [1...13]

Day count basis, specified as the comma-separated pair consisting of 'Basis' and a scalar positive integer or an NINST-by-1vector of positive integers using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`BusinessDayConvention` — Business day convention for cash flow dates
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day conventions for cash flow dates, specified as the comma-separated pair consisting of `'BusDayConvention'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`PayAccruedPremium` — Flag for accrued premiums
`true` (default) | scalar value of `true` or `false` | vector with values of `true` or `false`

Flag for accrued premiums, specified as the comma-separated pair consisting of `'PayAccruedPremium'` and a scalar Boolean flag or an `NINST`-by-`1` vector of Boolean flags that are `true` if accrued premiums are paid upon default and `false` otherwise.

Data Types: `logical`

`RecoveryRate` — Recovery rate
`0.4` (default) | scalar decimal | vector of decimals

Recovery rate, specified as the comma-separated pair consisting of `'RecoveryRate'` and a scalar decimal or an `NINST`-by-`1` vector of decimals from `0` to `1`.

Data Types: `double`

`Notional` — Contract notional value
`100` (default) | scalar positive integer | vector of positive integers

Contract notional value, specified as the comma-separated pair consisting of `'Notional'` and a scalar positive integer or an `NINST`-by-`1` vector of positive integers.

Data Types: `double`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes | cell array of character vectors | date string array

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and dates using an `NINST`-by-`1` vector of datetimes, cell array of date character vectors, or date string array. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
CDSobj = fininstrument("CDS",'Maturity',datetime(2025,12,15),'ContractSpread',200,'Holidays',H)
```

To support existing code, `CDS` also accepts serial date numbers as inputs, but they are not recommended.

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Maturity` — Maturity date
datetime | vector of datetimes

Maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`ContractSpread` — Contract spreads expressed in basis points
scalar numeric | numeric vector

Contract spreads expressed in basis points, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Period` — Premium payments per year
`4` (default) | scalar numeric with value of `1`, `2`, `3`, `4`, `6` or `12` | numeric vector with values of `1`, `2`, `3`, `4`, `6` or `12`

Premium payments per year, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day count basis, returned as a scalar positive integer or an `NINST`-by-`1` vector of positive integers.

Data Types: `double`

`BusinessDayConvention` — Business day convention for cash flow dates
`"actual"` (default) | string | string array

Business day conventions for cash flow dates, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

`PayAccruedPremium` — Flag for accrued premiums
`true` (default) | scalar value `true` or `false` | vector of values `true` or `false`

Flag for accrued premiums, returned as a scalar Boolean flag or an `NINST`-by-`1` vector of Boolean flags.

Data Types: `logical`

`RecoveryRate` — Recovery rate
`0.4` (default) | scalar decimal | vector of decimals

Recovery rate, returned as a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Notional` — Contract notional value
`100` (default) | scalar positive integer | vector of positive integers

Contract notional value, returned as a scalar positive integer or an `NINST`-by-`1` vector of positive integers.

Data Types: `double`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

Examples

Price CDS Instrument Using Default Probability Curve and Credit Pricer

This example shows the workflow to price a `CDS` instrument when you use a `defprobcurve` model and a `Credit` pricing method.

Create `CDS` Instrument Object

Use `fininstrument` to create a `CDS` instrument object.

```
CDS = fininstrument("CDS",'Maturity',datetime(2021,9,15),'ContractSpread',15,'Notional',20000,'Pe
```

```
CDS =
  CDS with properties:

          ContractSpread: 15
                Maturity: 15-Sep-2021
                  Period: 4
                   Basis: 3
            RecoveryRate: 0.4000
    BusinessDayConvention: "follow"
                Holidays: NaT
        PayAccruedPremium: 1
                Notional: 20000
                    Name: "CDS_instrument"
```

Create `defprobcurve` Object

Create a `defprobcurve` object using `defprobcurve`.

```
Settle = datetime(2020,9,20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities,'Basis',5)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                 Settle: 20-Sep-2020
                  Basis: 5
                  Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
```

```
ZeroDates = Settle + ZeroTimes;
myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Credit` Pricer Object

Use `finpricer` to create a `Credit` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("credit",'DefaultProbabilityCurve',DefaultProbCurve,'DiscountCurve',myRC)
```

```
outPricer =
  Credit with properties:

             DiscountCurve: [1x1 ratecurve]
                  TimeStep: 10
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price `CDS` Instrument

Use `price` to compute the price for the `CDS` instrument.

```
Price = price(outPricer,CDS)
```

```
Price = 52.7426
```

Price Multiple `CDS` Instruments Using Default Probability Curve and Credit Pricer

This example shows the workflow to price multiple `CDS` instruments when you use a `defprobcurve` model and a `Credit` pricing method.

Create `CDS` Instrument Object

Use `fininstrument` to create a `CDS` instrument object for three CDS instruments.

```
CDS = fininstrument("CDS",'Maturity',datetime([2021,9,15 ; 2021,10,15 ; 2021,11,15]),'ContractSp
```

```
CDS=3×1 object
  3x1 CDS array with properties:

    ContractSpread
    Maturity
    Period
```

```
    Basis
    RecoveryRate
    BusinessDayConvention
    Holidays
    PayAccruedPremium
    Notional
    Name
```

Create `defprobcurve` Object

Create a `defprobcurve` object using `defprobcurve`.

```
Settle = datetime(2020,9,20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities,'Basis',5)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                 Settle: 20-Sep-2020
                  Basis: 5
                  Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Credit` Pricer Object

Use `finpricer` to create a `Credit` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("credit",'DefaultProbabilityCurve',DefaultProbCurve,'DiscountCurve',myRC)
```

```
outPricer =
  Credit with properties:

             DiscountCurve: [1x1 ratecurve]
                  TimeStep: 10
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price CDS Instruments

Use `price` to compute the prices for the `CDS` instruments.

```
Price = price(outPricer,CDS)
```

```
Price = 3×1

   52.7426
   80.2945
  108.0357
```

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `CDS` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`CDSOption` | `finmodel` | `finpricer`

Topics

“Bootstrapping a Default Probability Curve from Credit Default Swaps” on page 8-42
“Price Multiple CDS Option Instruments Using CDS Black Model and CDS Black Pricer” on page 8-46
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

CDSOption

`CDSOption` instrument object

Description

Create and price a `CDSOption` instrument object for one or more CDS Option instruments using this workflow:

1. Use `fininstrument` to create a `CDSOption` instrument object for one or more CDS Option instruments. By default, this creates a single-name CDS option. You can create a CDS index option by specifying the optional name-value argument `AdjustedForwardSpread`.
2. Use `finmodel` to specify a `CDSBlack` model for the `CDSoption` instrument object.
3. Use `finpricer` to specify a `CDSBlack` pricing method for one or more `CDSoption` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods using a `CDSoption` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
CDSOptionObj = fininstrument(InstrumentType,'ExerciseDate',exercise_date,'
Strike',strike_value,'CDS',cds_obj)
CDSOptionObj = fininstrument( ___ ,Name,Value)
```

Description

`CDSOptionObj = fininstrument(InstrumentType,'ExerciseDate',exercise_date,'Strike',strike_value,'CDS',cds_obj)` creates a `CDSOption` object for one or more CDS Option instruments by specifying `InstrumentType` and sets the properties on page 11-2737 for the required name-value pair arguments `ExerciseDate`, `Strike`, and `CDS`.

`CDSOptionObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2737 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `CDSOptionObj = fininstrument("CDSoption",'ExerciseDate',datetime(2019,1,30),'Strike',500,'CDS',cds_object,'Name',"cdsoption_instrument")` creates a `CDSOption` instrument for a single-name CDS option with a strike of 500. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"CDSoption"` | string array with values of `"CDSoption"` | character vector with value `'CDSoption'` | cell array of character vectors with values of `'CDSoption'`

Instrument type, specified as a string with the value of "CDSoption", a character vector with the value of 'CDSoption', an NINST-by-1 string array with values of "CDSoption", or an NINST-by-1 cell array of character vectors with values of 'CDSoption'.

Data Types: char | cell | string

CDSOption Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: CDSOptionObj = fininstrument("CDSoption",'ExerciseDate',datetime(2019,1,30),'Strike',500,'CDS',cds_object,'Name',"cdsoption_instrument")

Required CDSOption Name-Value Pair Arguments

ExerciseDate — Option exercise date
datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of 'ExerciseDate' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, CDSOption also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by datetime because the ExerciseDate property is stored as a datetime.

Strike — Option strike price
scalar nonnegative numeric | vector of nonnegative numeric

Option strike price, specified as the comma-separated pair consisting of 'Strike' and a scalar nonnegative numeric or an NINST-by-1 vector of nonnegative numeric.

Data Types: double

CDS — CDS object
CDS object | vector of CDS objects

CDS object, specified as the comma-separated pair consisting of 'CDS' and a scalar CDS object or an NINST-by-1 vector of CDS objects.

Data Types: object

Optional CDSOption Name-Value Pair Argument

OptionType — Option type
"call" (default) | string with value "call" or "put" | string array with values of "call" or "put" | character vector with value 'call' or 'put' | cell array of character vectors with values of 'call' or 'put'

Option type, specified as the comma-separated pair consisting of 'OptionType' and a scalar string or character vector or an NINST-by-1 cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

`Knockout` — Flag indicating if option is knockout type
`false` (default) | scalar logical with value `true` or `false` | vector of logicals with values of `true` or `false`

Flag indicating if option is knockout type, specified as the comma-separated pair consisting of `'Knockout'` and a scalar logical or an `NINST`-by-`1` vector of logical values.

Data Types: `logical`

`AdjustedForwardSpread` — Adjusted forward spread (in basis points) for pricing CDS index option
`NaN` (single-name CDS option) (default) | scalar numeric | numeric vector

Adjusted forward spread (in basis points) for pricing a CDS index option, specified as the comma-separated pair consisting of `'AdjustedForwardSpread'` and a scalar numeric or an `NINST`-by-`1` numeric vector. For more information on using `'AdjustedForwardSpread'` when pricing a CDS index option, see “Price CDS Index Options Using CDS Black Model and CDS Black Pricer” on page 11-2742.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`ExerciseDate` — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Strike` — Option strike price
scalar nonnegative numeric | vector of nonnegative numeric

Option strike price, returned as a scalar nonnegative numeric or an `NINST`-by-`1` vector of nonnegative numeric values.

Data Types: `double`

`CDS` — `CDS` object
`CDS` object | vector of `CDS` objects

`CDS` object, returned as a scalar `CDS` object or an `NINST`-by-`1` vector of CDS objects.

Data Types: `object`

OptionType — Definition of option
"call" (default) | string with value "call" or "put" | string array with values of "call" or "put"

Definition of option, returned as a scalar string or an NINST-by-1 string array with values of "call" or "put".

Data Types: string

Knockout — Flag indicating if option is knockout type
false (default) | scalar logical with value true or false | vector of logicals with values of true or false

Flag indicating if option is knockout type, returned as a scalar logical or an NINST-by-1 vector of logicals.

Data Types: logical

AdjustedForwardSpread — Adjusted forward spread (in basis points) for pricing CDS index option
NaN (single-name CDS option) (default) | scalar numeric | numeric vector

Adjusted forward spread (in basis points) for pricing a CDS index option, returned as a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

Name — User-defined name for instrument
" " (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Examples

Price CDS Option Instrument Using CDS Black Model and CDS Black Pricer

This example shows the workflow to price a CDSOption instrument for a single-name CDS option when you use a CDSBlack model and a CDSBlack pricing method.

Create CDS Instrument Object

Use fininstrument to create the underlying CDS instrument object.

```
CDSOpt = fininstrument("CDS",'Maturity',datetime(2021,9,15),'ContractSpread',150,'Notional',100,
```

```
CDSOpt =
  CDS with properties:

          ContractSpread: 150
                Maturity: 15-Sep-2021
                  Period: 4
                   Basis: 2
            RecoveryRate: 0.4000
    BusinessDayConvention: "actual"
                Holidays: NaT
```

```
        PayAccruedPremium: 1
                 Notional: 100
                     Name: "CDS_option"
```

Create defprobcurve Object

Create a `defprobcurve` object using `defprobcurve`.

```
Settle = datetime(2020,9,20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities,'Basis',5)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                  Settle: 20-Sep-2020
                   Basis: 5
                   Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                 Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create CDSOption Instrument Object

Use `fininstrument` to create a `CDSOption` instrument object for a single-name CDS option.

```
CDSOptionInst = fininstrument("CDSOption",'ExerciseDate',datetime(2021,8,15),'Strike',20,'CDS',CD
```

```
CDSOptionInst =
  CDSOption with properties:

              OptionType: "put"
```

```
                Strike: 20
             Knockout: 1
    AdjustedForwardSpread: NaN
            ExerciseDate: 15-Aug-2021
                     CDS: [1x1 fininstrument.CDS]
                    Name: "CDSOption_option"
```

Create CDSBlack Model Object

Use `finmodel` to create a `CDSBlack` model object.

```
CDSBlackModel = finmodel("CDSBlack",'SpreadVolatility',.2)
```

```
CDSBlackModel =
  CDSBlack with properties:

    SpreadVolatility: 0.2000
```

Create CDSBlack Pricer Object

Use `finpricer` to create a `CDSBlack` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',CDSBlackModel,'DefaultProbabilityCurve',DefaultProbCurve
```

```
outPricer =
  CDSBlack with properties:

                      Model: [1x1 finmodel.CDSBlack]
              DiscountCurve: [1x1 ratecurve]
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price CDSOption Instrument

Use `price` to compute the price for the `CDSOption` instrument for a single-name CDS option.

```
Price = price(outPricer,CDSOptionInst)
```

```
Price = 3.3016e-04
```

Price Multiple CDS Option Instruments Using CDS Black Model and CDS Black Pricer

This example shows the workflow to price multiple `CDSOption` instruments when you use a `CDSBlack` model and a `CDSBlack` pricing method.

Create CDS Instrument Object

Use `fininstrument` to create the underlying `CDS` instrument object for three CDS Option instruments.

```
CDSOpt = fininstrument("CDS",'Maturity',datetime([2021,9,15 ; 2021,10,15 ; 2021,11,15]),'Contract
```

```
CDSOpt=3×1 object
  3x1 CDS array with properties:
```

```
    ContractSpread
    Maturity
    Period
    Basis
    RecoveryRate
    BusinessDayConvention
    Holidays
    PayAccruedPremium
    Notional
    Name
```

Create `defprobcurve` Object

Create a `defprobcurve` object using `defprobcurve`.

```
Settle = datetime(2020,9,20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities,'Basis',5)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                 Settle: 20-Sep-2020
                  Basis: 5
                  Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: [10x1 datetime]
             Rates: [10x1 double]
            Settle: 15-Sep-2020
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create CDSOption Instrument Object

Use `fininstrument` to create a `CDSOption` instrument object for three single-name CDS options.

```
CDSOptionInst = fininstrument("CDSOption",'ExerciseDate',datetime(2021,8,15),'Strike',20,'CDS',CD
```

```
CDSOptionInst=3×1 object
  3x1 CDSOption array with properties:

    OptionType
    Strike
    Knockout
    AdjustedForwardSpread
    ExerciseDate
    CDS
    Name
```

Create CDSBlack Model Object

Use `finmodel` to create a `CDSBlack` model object.

```
CDSBlackModel = finmodel("CDSBlack",'SpreadVolatility',.2)
```

```
CDSBlackModel =
  CDSBlack with properties:

    SpreadVolatility: 0.2000
```

Create CDSBlack Pricer Object

Use `finpricer` to create a `CDSBlack` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',CDSBlackModel,'DefaultProbabilityCurve',DefaultProbCurve
```

```
outPricer =
  CDSBlack with properties:

                     Model: [1x1 finmodel.CDSBlack]
             DiscountCurve: [1x1 ratecurve]
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price CDSOption Instruments

Use `price` to compute the prices for the three `CDSOption` instruments.

```
Price = price(outPricer,CDSOptionInst)
```

```
Price = 3×1

    0.0003
    0.0384
    0.5941
```

Price CDS Index Options Using CDS Black Model and CDS Black Pricer

This example shows the workflow to use a `CDSOption` instrument to price CDS index options when you use a `CDSBlack` model and a `CDSBlack` pricing method.

Set Up Data for CDS Index

```
% CDS index and option data
Recovery = .4;
Basis = 2;
Period = 4;
CDSMaturity = datetime(2017,6,20);
ContractSpread = 100;
IndexSpread = 140;
BusDayConvention = 'follow';
Settle = datetime(2012,4,13);
OptionMaturity = datetime(2012,6,20);
OptionStrike = 140;
SpreadVolatility = .69;
```

Create `ratecurve` Object for Zero Curve Using `irbootstrap`

Create `ratecurve` object for a zero curve using `irbootstrap`.

```
% Zero curve data
DepRates = [0.004111 0.00563 0.00757 0.01053]';
DepTimes = calmonths([1 2 3 6]');
DepDates = Settle + DepTimes;
nDeposits = length(DepTimes);

SwapRates = [0.01387 0.01035 0.01145 0.01318 0.01508 0.01700 0.01868 ...
    0.02012 0.02132 0.02237 0.02408 0.02564 0.02612 0.02524]';
SwapTimes = calyears([1 2 3 4 5 6 7 8 9 10 12 15 20 30]');
SwapDates = Settle + SwapTimes;
nSwaps = length(SwapTimes);

nInst = nDeposits + nSwaps;

BootInstruments(nInst,1) = fininstrument.FinInstrument;
for ii=1:length(DepDates)
    BootInstruments(ii) = fininstrument("deposit","Maturity",DepDates(ii),"Rate",DepRates(ii));
end

for ii=1:length(SwapDates)
    BootInstruments(ii+nDeposits) = fininstrument("swap","Maturity",SwapDates(ii),"LegRate",[Swa
end

ZeroCurve = irbootstrap(BootInstruments,Settle)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [18x1 datetime]
               Rates: [18x1 double]
              Settle: 13-Apr-2012
```

```
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Bootstrap Default Probability Curve

Use `defprobstrip` to bootstrap default probability curve assuming a flat index spread.

```
ProbDates = datemnth(OptionMaturity,(0:5*12)');
MarketCDSInstruments = fininstrument("cds", ...
    'ContractSpread', ContractSpread, 'Maturity', CDSMaturity);
DefaultProbCurve = defprobstrip(ZeroCurve, MarketCDSInstruments, IndexSpread, 'ProbDates', ProbDa
```

```
DefaultProbCurve =
  defprobcurve with properties:

                 Settle: 13-Apr-2012
                  Basis: 2
                  Dates: [61x1 datetime]
    DefaultProbabilities: [61x1 double]
```

Compute Spot and Forward RPV01s

Compute the spot and forward RPV01s using `cdsrpv01`.

```
ProbData = [datenum(DefaultProbCurve.Dates) DefaultProbCurve.DefaultProbabilities];

% RPV01(t,T)
RPV01_CDSMaturity = cdsrpv01(ZeroCurve,ProbData,Settle,CDSMaturity)
```

```
RPV01_CDSMaturity = 4.7853
```

```
% RPV01(t,t_E,T)
RPV01_OptionExpiryForward = cdsrpv01(ZeroCurve,ProbData,Settle,CDSMaturity,...
    'StartDate',OptionMaturity)
```

```
RPV01_OptionExpiryForward = 4.5972
```

```
% RPV01(t,t_E) = RPV01(t,T) - RPV01(t,t_E,T)
RPV01_OptionExpiry = RPV01_CDSMaturity - RPV01_OptionExpiryForward
```

```
RPV01_OptionExpiry = 0.1882
```

Compute Spot Spreads

Compute the spot spreads using `cdsspread`.

```
% S(t,t_E)
Spread_OptionExpiry = cdsspread(ZeroCurve,ProbData,Settle,OptionMaturity,...
    'Period',Period,'Basis',Basis,'BusDayConvention',BusDayConvention,...
    'PayAccruedPremium',true,'recoveryrate',Recovery)
```

```
Spread_OptionExpiry = 139.8995
```

```
% S(t,T)
Spread_CDSMaturity = cdsspread(ZeroCurve,ProbData,Settle,CDSMaturity,...
    'Period',Period,'Basis',Basis,'BusDayConvention',BusDayConvention,...
    'PayAccruedPremium',true,'recoveryrate',Recovery)
```

```
Spread_CDSMaturity = 139.9999
```

Compute Forward Spread

Compute the forward spread using the spot spreads and RPV01s.

```
% F = S(t,t_E,T)
ForwardSpread = (Spread_CDSMaturity.*RPV01_CDSMaturity - Spread_OptionExpiry.*RPV01_OptionExpiry
```

```
ForwardSpread = 140.0040
```

Compute Front-End Protection

Compute the front-end protection (FEP).

```
FEP = 10000*(1-Recovery)*ZeroCurve.discountfactors(OptionMaturity)*DefaultProbCurve.DefaultProba
```

```
FEP = 26.3108
```

Compute Adjusted Forward Spread

Compute the adjusted forward spread to use when creating an `CDSOption` instrument.

```
AdjustedForwardSpread = ForwardSpread + FEP./RPV01_OptionExpiryForward
```

```
AdjustedForwardSpread = 145.7273
```

Compute CDS Option Prices with Adjusted Forward Spread

Use `fininstrument` to create the underlying `CDS` instrument object for the two CDS Option instruments.

```
CDS = fininstrument("cds",'ContractSpread', ContractSpread, 'Maturity', CDSMaturity)
```

```
CDS =
  CDS with properties:

          ContractSpread: 100
                Maturity: 20-Jun-2017
                  Period: 4
                   Basis: 2
            RecoveryRate: 0.4000
    BusinessDayConvention: "actual"
                Holidays: NaT
        PayAccruedPremium: 1
                Notional: 10000000
                    Name: ""
```

Use `fininstrument` to create a `CDSOption` instrument for two CDS index option instruments.

```
CDSCallOption = fininstrument("cdsoption", 'Strike', OptionStrike, ...
    'ExerciseDate', OptionMaturity, 'OptionType', 'call', 'CDS', CDS, ...
    'Knockout',true, 'AdjustedForwardSpread', AdjustedForwardSpread)
```

```
CDSCallOption =
  CDSOption with properties:

              OptionType: "call"
                  Strike: 140
```

```
                Knockout: 1
    AdjustedForwardSpread: 145.7273
             ExerciseDate: 20-Jun-2012
                      CDS: [1x1 fininstrument.CDS]
                     Name: ""
```

```
CDSPutOption = fininstrument("cdsoption", 'Strike', OptionStrike, ...
    'ExerciseDate', OptionMaturity, 'OptionType', 'put', 'CDS', CDS, ...
    'Knockout',true, 'AdjustedForwardSpread', AdjustedForwardSpread)
```

```
CDSPutOption =
  CDSOption with properties:

               OptionType: "put"
                   Strike: 140
                Knockout: 1
    AdjustedForwardSpread: 145.7273
             ExerciseDate: 20-Jun-2012
                      CDS: [1x1 fininstrument.CDS]
                     Name: ""
```

Create CDSBlack Model Object

Use `finmodel` to create a `CDSBlack` model object.

```
CDSOptionModel = finmodel("cdsblack",'SpreadVolatility',SpreadVolatility)
```

```
CDSOptionModel =
  CDSBlack with properties:

    SpreadVolatility: 0.6900
```

Create CDSBlack Pricer Object

Use `finpricer` to create a `CDSBlack` pricer object and use the `ratecurve` object for the zero curve for the `'DiscountCurve'` name-value pair argument.

```
CDSOptionpricer = finpricer("analytic",'Model',CDSOptionModel,'DiscountCurve',ZeroCurve,'DefaultI
```

```
CDSOptionpricer =
  CDSBlack with properties:

                     Model: [1x1 finmodel.CDSBlack]
             DiscountCurve: [1x1 ratecurve]
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price CDS Index Options

Use `price` to compute the price for the two CDS index options.

```
outPrice = price(CDSOptionpricer, [CDSCallOption;CDSPutOption]);
fprintf('    Payer: %.0f   Receiver: %.0f  \n',outPrice(1),outPrice(2));
```

```
Payer: 92   Receiver: 66
```

More About

Credit Default Swap Option

A credit default swap (CDS) option, or credit default swaption, is a contract that provides the option holder with the right, but not the obligation, to enter into a credit default swap in the future.

CDS options can be either payer swaptions or receiver swaptions. In a payer swaption, the option holder has the right to enter into a CDS in which they are paying premiums, and in a receiver swaption, the option holder is receiving premiums.

A CDS option can be a single-name CDS option or CDS index option. When pricing a CDS index option , use the optional name-value argument `AdjustedForwardSpread`. Unlike a single-name CDS, a CDS portfolio index contains multiple credits. When one or more of the credits default, the corresponding contingent payments are made to the protection buyer but the contract still continues with reduced coupon payments. Considering the fact that the CDS index option does not cancel when some of the underlying credits default before expiry, one might attempt to price CDS index options using the Black's model for non-knockout single-name CDS option. However, Black's model in this form is not appropriate for pricing CDS index options because it does not capture the exercise decision correctly when the strike spread (K) is very high, nor does it ensure put-call parity when (K) is not equal to the contractual spread (O'Kane, 2008).

However, with the appropriate modifications, Black's model for single-name CDS options used with a `CDSOption` instrument and `CDSBlack` pricer can provide a good approximation for CDS index options. While there are some variations in the way the Black's model is modified for CDS index options, they usually involve adjusting the forward spread F, the strike spread K, or both. Here we describe the approach of adjusting the forward spread only. In the Black's model for single-name CDS options, the forward spread F is defined as:

$$F = S(t, t_E, T) = \frac{S(t, T)RPV01(t, T) - S(t, t_E)RPV01(t, t_E)}{RPV01(t, t_E, T)}$$

where

S is the spread.

RPV01 is the risky present value of a basis point (see `cdsrpv01`).

t is the valuation date.

t_E is the option expiry date.

T is the CDS maturity date.

To capture the exercise decision correctly for CDS index options, we use the knockout form of the Black's model and adjust the forward spread to incorporate the *FEP* (front end protection) as follows:

$$F_{Adj} = F + \frac{FEP}{RPV01(t, t_E, T)}$$

with *FEP* defined as

$$FEP = (1 - R)Z(t, t_E)(1 - Q(t, t_E))$$

where

R is the recovery rate.

Z is the discount factor.

Q is the survival probability.

In the `CDSOption` object, forward spread adjustment can be made with the `AdjustedForwardSpread` name-value argument. When computing the adjusted forward spread, you can compute the spreads using `cdsspread` and the RPV01s using `cdsrpv01`.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `CDSOption` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] O'Kane, D. *Modelling Single-name and Multi-name Credit Derivatives*. Wiley, 2008, pp. 156–169.

See Also

Functions

`CDS` | `finmodel` | `finpricer`

Topics

“Price Multiple CDS Option Instruments Using CDS Black Model and CDS Black Pricer” on page 8-46
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

FixedBond

`FixedBond` instrument object

Description

Create and price a `FixedBond` instrument object for one of more Fixed Bond instruments using this workflow:

1. Use `fininstrument` to create a `FixedBond` instrument object for one of more Fixed Bond instruments.
2. Use `ratecurve` to specify a curve model for the `FixedBond` instrument object or use a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model.
3. Choose a pricing method.
   - When using a `ratecurve` use `finpricer` to specify a `Discount` pricing method for one or more `FixedBond` instruments.
   - When using a `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` model, use `finpricer` to specify an `IRTree` pricing method for one or more `FixedBond` instruments.
   - When using a `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model, use `finpricer` to specify an `IRMonteCarlo` pricing method for one or more `FixedBond` instruments.

For more detailed information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `FixedBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
FixedBondObj = fininstrument(InstrumentType,'CouponRate',couponrate_value,'
Maturity',maturity_date)
FixedBondObj = fininstrument( ___ ,Name,Value)
```

Description

`FixedBondObj = fininstrument(InstrumentType,'CouponRate',couponrate_value,'Maturity',maturity_date)` creates a `FixedBond` object for one of more Fixed Bond instruments by specifying `InstrumentType` and sets the properties on page 11-2754 for the required name-value pair arguments `CouponRate` and `Maturity`.

The `FixedBond` instrument supports a vanilla bond, a stepped coupon bond, and an amortizing bond. For more information, see “More About” on page 11-2767.

`FixedBondObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2754 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `FixedBondObj = fininstrument("FixedBond",'CouponRate',0.034,'Maturity',datetime(2019,1,30),'Period',4,'Basis',1,'Principal',100,'FirstCouponDate',datetime(2016,1,30),'EndMonthRule',true,'Name',"fixedbond_instrument")` creates a `FixedBond` option with a coupon rate of 0.34 and a maturity of January 30, 2019. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Fixedbond"` | string array with values of `"Fixedbond"` | character vector with value `'FixedBond'` | cell array of character vectors with values of `'FixedBond'`

Instrument type, specified as a string with the value of `"FixedBond"`, a character vector with the value of `'FixedBond'`, an `NINST`-by-`1` string array with values of `"FixedBond"`, or an `NINST`-by-`1` cell array of character vectors with values of `'FixedBond'`.

Data Types: `char` | `cell` | `string`

`FixedBond` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `FixedBondObj = fininstrument("FixedBond",'CouponRate',0.034,'Maturity',datetime(2019,1,30),'Period',4,'Basis',1,'Principal',100,'FirstCouponDate',datetime(2016,1,30),'EndMonthRule',true,'Name',"fixedbond_instrument")`

Required `FixedBond` Name-Value Pair Arguments

`CouponRate` — `FixedBond` coupon rate

scalar decimal | vector of decimals | timetable

`FixedBond` coupon rate, specified as the comma-separated pair consisting of `'CouponRate'` and a scalar decimal or an `NINST`-by-`1` vector of decimals for an annual rate or a timetable where the first column is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Note If you are creating one or more `FixedBond` instruments and use a timetable, the timetable specification applies to all of the `FixedBond` instruments. `CouponRate` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `double` | `timetable`

`Maturity` — `FixedBond` maturity date

datetime array | string array | date character vector

`FixedBond` maturity date, specified as the comma-separated pair consisting of `'Maturity'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `FixedBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

Optional FixedBond Name-Value Pair Arguments

`Period` — Frequency of payments per year

`2` (default) | scalar numeric value of `0`, `1`, `2`, `3`, `4`, `6`, or `12` | numeric vector with values of `0`, `1`, `2`, `3`, `4`, `6`, or `12`

Frequency of payments, specified as the comma-separated pair consisting of `'Period'` and a scalar integer or an `NINST`-by-`1` vector of integers. Values for `Period` are `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day count basis

`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and scalar integer or an `NINST`-by-`1` vector of integers using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal amount or principal value schedule

`100` (default) | scalar numeric | numeric vector | timetable

Principal amount or principal value schedule, specified as the comma-separated pair consisting of `'Principal'` and a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

`Principal` accepts a `timetable`, where the first column is dates and the second column is the associated notional principal value. The date indicates the last day that the principal value is valid.

Note If you are creating one or more `FixedBond` instruments and use a timetable, the timetable specification applies to all of the `FixedBond` instruments. `Principal` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `double` | `timetable`

`DaycountAdjustedCashFlow` — Flag indicating whether cash flow adjusts for day count convention
`false` (default) | scalar logical value of `true` or `false` | vector of logical values of `true` or `false`

Flag indicating whether cash flow is adjusted by day count convention, specified as the comma-separated pair consisting of `'DaycountAdjustedCashFlow'` and a scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions for cash flow dates
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day conventions for cash flow dates, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and dates using an `NINST`-by-`1` vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
FixedBondObj = fininstrument("FixedBond",'CouponRate',0.34,'Maturity',datetime(2025,12,15),'Holi
```

To support existing code, `FixedBond` also accepts serial date numbers as inputs, but they are not recommended.

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`true` (in effect) (default) | scalar logical value of `true` or `false` | vector of logical values of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar logical value or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

- If you set `EndMonthRule` to `false`, the software ignores the rule, meaning that a payment date is always the same numerical day of the month.
- If you set `EndMonthRule` to `true`, the software sets the rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
`NaT` (default) | datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `FixedBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `IssueDate` property is stored as a datetime.

`FirstCouponDate` — Irregular first coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `FixedBond` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `FirstCouponDate` property is stored as a datetime.

`LastCouponDate` — Irregular last coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, FixedBond also accepts serial date numbers as inputs, but they are not recommended.

If you specify LastCouponDate but not FirstCouponDate, LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at LastCouponDate, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify LastCouponDate, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by datetime because the LastCouponDate property is stored as a datetime.

StartDate — Forward starting date of payments
NaT (default) | datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of 'StartDate' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, FixedBond also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by datetime because the StartDate property is stored as a datetime.

Name — User-defined name for instrument
" " (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of 'Name' and a scalar string or character vector or an NINST-by-1 cell array of character vectors or string array.

Data Types: char | cell | string

Properties

CouponRate — FixedBond coupon annual rate
scalar decimal | vector of decimals | timetable

FixedBond coupon annual rate, returned as a scalar decimal or an NINST-by-1 vector of decimals or a timetable.

Data Types: double | timetable

Maturity — FixedBond maturity date
scalar datetime | vector of datetimes

FixedBond maturity date, returned as a scalar datetime or NINST-by-1 vector of datetimes.

Data Types: datetime

Period — Frequency of payments per year
2 (default) | scalar integer | vector of integers

Frequency of payments per year, returned as a scalar integer or an NINST-by-1 vector of integers.

Data Types: double

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Principal` — Principal amount or principal value schedules
`100` (default) | scalar numeric | numeric vector | timetable

Principal amount or principal value schedules, returned as a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

Data Types: `double`

`DaycountAdjustedCashFlow` — Flag indicating whether cash flow adjusts for day count convention
`false` (default) | scalar logical value of `true` or `false` | vector of logicals with values of `true` or `false`

Flag indicating whether cash flow adjusts for day count convention, returned as scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array

Business day conventions, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`true` (in effect) (default) | scalar logical value of `true` or `false` | vector of logicals with values of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month having 30 or fewer days, returned as a scalar logical or an `NINST`-by-`1` vector of logical values.

Data Types: `logical`

`IssueDate` — Bond issue date
`NaT` (default) | datetime | vector of datetimes

Bond issue date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`FirstCouponDate` — Irregular first coupon date
`NaT` (default) | datetime | vector of datetimes

Irregular first coupon date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`LastCouponDate` — Irregular last coupon date
`NaT` (default) | datetime | vector of datetimes

Irregular last coupon date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`StartDate` — Forward starting date of payments
`NaT` (default) | datetime | vector of datetimes

Forward starting date of payments, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

cashflows	Compute cash flow for FixedBond, FloatBond, Swap, FRA, STIRFuture, OISFuture, OvernightIndexedSwap, or Deposit instrument

Examples

Price Vanilla Fixed Bond Instrument Using `ratecurve` and Discount Pricer

This example shows the workflow to price a vanilla `FixedBond` instrument when you use a `ratecurve` and a `Discount` pricing method.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",'Maturity',datetime(2022,9,15),'CouponRate',0.021,'Period',2,'Ba
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0210
                     Period: 2
                      Basis: 1
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
```

```
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond_instrument"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `FixedBond` Instrument

Use `price` to compute the price and sensitivities for the `FixedBond` instrument.

```
[Price, outPR] = price(outPricer, FixB,["all"])
```

```
Price = 104.5679
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results

ans=1×2 table
    Price       DV01
    ______    ________

    104.57    0.040397
```

Price Multiple Vanilla Fixed Bond Instruments Using ratecurve and Discount Pricer

This example shows the workflow to price multiple vanilla FixedBond instruments when you use a ratecurve and a Discount pricing method.

Create FixedBond Instrument Object

Use fininstrument to create a FixedBond instrument object for three Fixed Bond instruments.

```
FixB = fininstrument("FixedBond",'Maturity',datetime([2022,9,15 ; 2022,10,15 ; 2022,11,15]),'Coup

FixB=3×1 object
  3x1 FixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    Name
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)

myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
               Basis: 0
```

```
                Dates: [10x1 datetime]
                Rates: [10x1 double]
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `FixedBond` Instruments

Use `price` to compute the prices and sensitivities for the `FixedBond` instruments.

```
[Price, outPR] = price(outPricer, FixB,["all"])
```

```
Price = 3×1

  104.5679
  261.4498
  522.9174


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    ________

    104.57    0.040397


ans=1×2 table
    Price     DV01
    ______    _____

    261.45    0.103


ans=1×2 table
    Price      DV01
```

```
    ______    _______

    522.92    0.21013
```

Price Stepped Fixed Bond Instrument Using `ratecurve` and Discount Pricer

This example shows the workflow to price a stepped `FixedBond` instrument when you use a `ratecurve` and a `Discount` pricing method.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a stepped `FixedBond` instrument object.

```
Maturity = datetime(2024,1,1);
Period = 1;
CDates = datetime([2020,1,1 ; 2024,1,1]);
CRates = [.025; .03];
CouponRate = timetable(CDates,CRates);

SBond = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',CouponRate,'Period',Period)
```

```
SBond =
  FixedBond with properties:

                 CouponRate: [2x1 timetable]
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2024
                        Name: ""
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: 1
```

```
                Basis: 0
                Dates: [10x1 datetime]
                Rates: [10x1 double]
               Settle: 01-Jan-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',ZeroCurve)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `FixedBond` Instrument

Use `price` to compute the price and sensitivities for the vanilla `FixedBond` instrument.

```
[Price, outPR] = price(outPricer, SBond,["all"])
```

```
Price = 109.6218
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    ________

    109.62    0.061108
```

Price Amortizing Fixed Bond Instrument Using `ratecurve` and Discount Pricer

This example shows the workflow to price a amortizing `FixedBond` instrument when you use a `ratecurve` and a `Discount` pricing method.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a amortizing `FixedBond` instrument object.

```
Maturity = datetime(2024,1,1);
Period = 1;
```

```
ADates = datetime([2020,1,1 ; 2024,1,1]);
APrincipal = [100; 85];
Principal = timetable(ADates,APrincipal);
Bondamort = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',0.025,'Period',Period,'Pr
```

```
Bondamort =
  FixedBond with properties:

                 CouponRate: 0.0250
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: [2x1 timetable]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2024
                        Name: ""
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding);
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',ZeroCurve)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `FixedBond` Instrument

Use `price` to compute the price and sensitivities for the vanilla `FixedBond` instrument.

```
[Price, outPR] = price(outPricer,Bondamort,["all"])
```

```
Price = 107.1273
```

```
outPR =
  priceresult with properties:
```

```
       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    ________

    107.13    0.054279
```

Price Fixed Bond Instrument Using Hull-White Model and IRMonteCarlo Pricer

This example shows the workflow to price a `FixedBond` instrument when using a `HullWhite` model and an `IRMonteCarlo` pricing method.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond","Maturity",datetime(2022,9,15),"CouponRate",0.05,'Name',"fixed_
```

```
FixB =
  FixedBond with properties:

                  CouponRate: 0.0500
                      Period: 2
                       Basis: 0
                 EndMonthRule: 1
                    Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond"
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.32,'Sigma',0.49)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.3200
    Sigma: 0.4900
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use finpricer to create an IRMonteCarlo pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",'Model',HullWhiteModel,'DiscountCurve',myRC,'SimulationDates
```

```
outPricer =
  HWMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jan-2021    ...    ]
              Model: [1x1 finmodel.HullWhite]
```

Price FixedBond Instrument

Use price to compute the price and sensitivities for the FixedBond instrument.

```
[Price,outPR] = price(outPricer,FixB,["all"])
```

```
Price = 115.0303
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma     Vega
    ______    _______    ______    ____

    115.03    -397.13    1430.4     0
```

Price FixedBond Instrument Using a Hull-White Model and IRTree Pricer

This example shows the workflow to price a `FixedBond` instrument when using a `HullWhite` model and a IRTree pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond","Maturity",datetime(2029,9,15),"CouponRate",.05,"Period",1,"Name
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
              FirstCouponDate: NaT
               LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2029
                        Name: "fixed_bond_instrument"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Type = "zero";
ZeroTimes = [calyears([1:10])]';
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                 Type: "zero"
          Compounding: -1
                Basis: 0
                Dates: [10x1 datetime]
                Rates: [10x1 double]
```

```
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("hullwhite",'Alpha',0.052,'Sigma',0.34)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0520
    Sigma: 0.3400
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("irtree","model",HullWhiteModel,"DiscountCurve",myRC,"TreeDates",ZeroDa
```

```
HWTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [10x1 datetime]
           Model: [1x1 finmodel.HullWhite]
   DiscountCurve: [1x1 ratecurve]
```

```
HWTreePricer.Tree
```

```
ans = struct with fields:
        tObs: [0 1 1.9973 2.9945 3.9918 4.9918 5.9891 6.9863 7.9836 8.9836]
        dObs: [15-Sep-2019    15-Sep-2020    15-Sep-2021    ...    ]
      CFlowT: {1x10 cell}
       Probs: {1x9 cell}
     Connect: {1x9 cell}
     FwdTree: {1x10 cell}
    RateTree: {1x10 cell}
```

Price FixedBond Instrument

Use `price` to compute the price and sensitivities for the `FixedBond` instrument.

```
[Price, outPR] = price(HWTreePricer, FixB,["all"])
```

```
Price = 117.9440
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
```

```
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma        Vega
    _____    _______    _____    __________

    117.94    -964.01    8868.6    -4.2633e-10
```

More About

Fixed-Rate Note

A fixed-rate note is a long-term debt security with a preset interest rate and maturity, by which the interest must be paid.

The principal might or might not be paid at maturity. In Financial Instruments Toolbox, the principal is always paid at maturity. For more information, see “Fixed-Rate Note” on page 2-9.

Vanilla Bond

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment.

Stepped Coupon Bond

A step-up bond and a step-down bond are debt securities with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond.

Bond with an Amortization Schedule

An amortized bond is treated as an asset, with the discount amount being amortized to interest expense over the life of the bond.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `FixedBond` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`FixedBondOption` | `finmodel` | `finpricer` | `timetable`

Topics
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Bond Portfolio Optimization Using Portfolio Object”
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

FixedBondOption

`FixedBondOption` instrument object

Description

Create and price a `FixedBondOption` instrument object for one or more Fixed Bond Option instruments using this workflow:

1. Use `fininstrument` to create a `FixedBondOption` instrument object for one or more Fixed Bond Option instruments.
2. Use `finmodel` to specify a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model for the `FixedBondOption` instrument object.
3. Choose a pricing method.
   - When using a `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` model, use `finpricer` to specify an `IRTree` pricing method for one or more `FixedBondOption` instruments.
   - When using a `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model, use `finpricer` to specify an `IRMonteCarlo` pricing method for one or more `FixedBondOption` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `FixedBondOption` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
FixedBondOptionObj = fininstrument(InstrumentType,'Strike',strike_value,'
ExerciseDate',exercise_date,'Bond',bond_obj)
FixedBondOptionObj = fininstrument( ___ ,Name,Value)
```

Description

`FixedBondOptionObj = fininstrument(InstrumentType,'Strike',strike_value,'ExerciseDate',exercise_date,'Bond',bond_obj)` creates a `FixedBondOption` object for one or more Fixed Bond Option instruments by specifying `InstrumentType` and sets the properties on page 11-2771 for the required name-value pair arguments `Strike`, `ExerciseDate`, and `Bond`.

The `FixedBondOption` instrument supports a European or American option. For more information, see “More About” on page 11-2783.

`FixedBondOptionObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2771 using additional name-value pairs in addition to the required arguments in the previous

syntax. For example, `FixedBondOptionObj = fininstrument("FixedBondOption",'Strike',100,'ExerciseDate',datetime(2019,1,30),'Bond',bond_obj,'OptionType','put','ExerciseStyle',"American",'Name',"fixed_bond_option")` creates a `FixedBondOption` instrument with a strike of 100 and an American exercise. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"FixedBondOption"` | string array with values of `"FixedBondOption"` | character vector with value `'FixedBondOption'` | cell array of character vectors with values of `'FixedBondOption'`

Instrument type, specified as a string with the value of `"FixedBondOption"`, a character vector with the value of `'FixedBondOption'`, an `NINST`-by-`1` string array with values of `"FixedBondOption"`, or an `NINST`-by-`1` cell array of character vectors with values of `'FixedBondOption'`.

Data Types: `char` | `cell` | `string`

`FixedBondOption` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `FixedBondOptionObj = fininstrument("FixedBondOption",'Strike',100,'ExerciseDate',datetime(2019,1,30),'Bond',bond_obj,'OptionType','put','ExerciseStyle',"American",'Name',"fixed_bond_option")`

Required FixedBondOption Name-Value Pair Arguments

`Strike` — Option strike value

scalar nonnegative numeric | nonnegative numeric vector

Option strike value, specified as the comma-separated pair consisting of `'Strike'` and a scalar nonnegative numeric or an `NINST`-by-`1` nonnegative numeric vector.

Data Types: `double`

`ExerciseDate` — Option exercise date

datetime array | string array | date character vector | serial date number

Option exercise date, specified as the comma-separated pair consisting of `'ExerciseDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, date character vectors, or serial date numbers.

- For a European option, there is only one `ExerciseDate` on the option expiry date.
- For a Bermudan option, there is a `1`-by-`NSTRIKES` vector of exercise dates.
- For an American option, the option can be exercised between `Settle` of the `ratecurve` and the single listed `ExerciseDate`.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `ExerciseDate` property is stored as a `datetime`.

Data Types: `double` | `char` | `string` | `datetime`

`Bond` — Underlying `FixedBond` instrument
`FixedBond` object | vector of `FixedBond` objects

Underlying `FixedBond` instrument, specified as the comma-separated pair consisting of `'Bond'` and a scalar `FixedBond` object or an `NINST`-by-`1` vector of `FixedBond` objects.

Data Types: `object`

Optional FixedBondOption Name-Value Pair Arguments

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values of `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values of `'call'` or `'put'`

Option type, specified as the comma-separated pair consisting of `'OptionType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with values of `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'` | cell array of character vectors with values of `'European'`, `'American'`, or `'Bermudan'`

Option exercise style, specified as the comma-separated pair consisting of `'ExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array with values of `"European"`, `"American"`, or `"Bermudan"`.

Data Types: `string` | `cell` | `char`

`Name` — User-defined name for instrument
`"  "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Strike` — Option strike value
scalar nonnegative numeric | vector of nonnegative numeric

Option strike value, returned as a scalar nonnegative numeric or an `NINST`-by-`1` numeric vector of nonnegative values.

Data Types: `double`

`ExerciseDate` — Option exercise date
scalar datetime | vector of datetimes

Option exercise date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`OptionType` — Option type
`"call"` (default) | scalar string with value `"call"` or `"put"` | string array with values of `"call"` or `"put"`

Option type, returned as a scalar string or an `NINST`-by-`1` string array with values of `"call"` or `"put"`.

Data Types: `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | scalar string with value `"European"`, `"American"` | string array with values of `"European"`, `"American"`

Option exercise style, returned as a scalar string or an `NINST`-by-`1` string array with values of `"European"` or `"American"`.

Data Types: `string`

`Bond` — Underlying `FixedBond` instrument
scalar `FixedBond` object | vector of `FixedBond` objects

Underlying `FixedBond` instrument, returned as a scalar `FixedBond` object or an `NINST`-by-`1` vector of `FixedBond` objects.

Data Types: `object`

`Name` — User-defined name for instrument
`"  "` (default) | scalar string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

setExercisePolicy Set exercise policy for FixedBondOption, FloatBondOption, or Vanilla instrument

Examples

Price a FixedBondOption Instrument Using Hull-White Model and Hull-White Tree Pricer

This example shows the workflow to price a `FixedBondOption` instrument when you use a `HullWhite` model and an `IRTree` pricing method.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",'Maturity',datetime(2029,9,15),'CouponRate',.021,'Period',1
```

```
BondInst =
  FixedBond with properties:

                  CouponRate: 0.0210
```

```
                    Period: 1
                     Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2029
                        Name: "bond_instrument"
```

Create `FixedBondOption` Instrument Objects

Use `fininstrument` to create three callable `FixedBondOption` instrument objects with European, American, and Bermudan exercise.

```
FixedBOptionEuro = fininstrument("FixedBondOption",'ExerciseDate',datetime(2025,9,15),'Strike',98
```

```
FixedBOptionEuro =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2025
           Strike: 98
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option_european"
```

```
FixedBOptionAmerican = fininstrument("FixedBondOption",'ExerciseDate',datetime(2025,9,15),'Strike
```

```
FixedBOptionAmerican =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2025
           Strike: 98
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option_american"
```

```
FixedBOptionBermudan = fininstrument("FixedBondOption",'ExerciseDate',[datetime(2025,9,15) , date
```

```
FixedBOptionBermudan =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "bermudan"
     ExerciseDate: [15-Sep-2025    15-Nov-2025]
           Strike: [98 1000]
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option_bermudan"
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calyears([1:10])]';
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create a HullWhite Model Object

Use finmodel to create a HullWhite model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.01,'Sigma',0.05)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0100
    Sigma: 0.0500
```

Create IRTree Pricer Object

Use finpricer to create an IRTree pricer object and use the ratecurve object with the 'DiscountCurve' name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HullWhiteModel,'DiscountCurve',myRC,'TreeDates',ZeroDa
```

```
HWTreePricer =
  HWBKTree with properties:

             Tree: [1x1 struct]
        TreeDates: [10x1 datetime]
            Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

```
HWTreePricer.Tree
```

```
ans = struct with fields:
        tObs: [0 1 1.9973 2.9945 3.9918 4.9918 5.9891 6.9863 7.9836 8.9836]
```

```
        dObs: [15-Sep-2019    15-Sep-2020    15-Sep-2021    ...    ]
      CFlowT: {1x10 cell}
       Probs: {1x9 cell}
     Connect: {1x9 cell}
     FwdTree: {1x10 cell}
    RateTree: {1x10 cell}
```

Price FixedBondOption Instruments

Use `price` to compute the price and sensitivities for the two `FixedBondOption` instruments.

```
[Price, outPR] = price(HWTreePricer,FixedBOptionEuro,["all"])
```

```
Price = 10.7571

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma      Vega
    _____    ________    _____    _____

    10.757    -178.78    2207.3    308.87
```

```
[Price, outPR] = price(HWTreePricer,FixedBOptionAmerican,["all"])
```

```
Price = 19.2984

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma      Vega
    _____    ________    _____    _____

    19.298    -459.06    4977.1    437.32
```

```
[Price, outPR] = price(HWTreePricer,FixedBOptionBermudan,["all"])
```

```
Price = 10.7571

outPR =
  priceresult with properties:

       Results: [1x4 table]
```

```
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma      Vega
    ______    _______    ______    ______

    10.757    -178.78    2207.3    308.87
```

Price Multiple Fixed Bond Option Instruments Using Hull-White Model and Hull-White Tree Pricer

This example shows the workflow to price multiple `FixedBondOption` instruments when you use a `HullWhite` model and an `IRTree` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",'Maturity',datetime(2029,9,15),'CouponRate',.021,'Period',1
```

```
BondInst =
  FixedBond with properties:

                 CouponRate: 0.0210
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2029
                        Name: "bond_instrument"
```

Create FixedBondOption Instrument Objects

Use `fininstrument` to create a `FixedBondOption` instrument object with European exercise for three Fixed Bond Option instruments.

```
FixedBOptionEuro = fininstrument("FixedBondOption",'ExerciseDate',datetime([2025,9,15 ; 2025,10,1
```

```
FixedBOptionEuro=3×1 object
  3x1 FixedBondOption array with properties:

    OptionType
    ExerciseStyle
    ExerciseDate
    Strike
```

```
    Bond
    Name
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calyears([1:10])]';
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create a HullWhite Model Object

Use finmodel to create a HullWhite model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.01,'Sigma',0.05)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0100
    Sigma: 0.0500
```

Create IRTree Pricer Object

Use finpricer to create an IRTree pricer object and use the ratecurve object with the 'DiscountCurve' name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HullWhiteModel,'DiscountCurve',myRC,'TreeDates',ZeroDa
```

```
HWTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [10x1 datetime]
           Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

```
HWTreePricer.Tree
```

```
ans = struct with fields:
        tObs: [0 1 1.9973 2.9945 3.9918 4.9918 5.9891 6.9863 7.9836 8.9836]
        dObs: [15-Sep-2019    15-Sep-2020    15-Sep-2021    ...    ]
      CFlowT: {1x10 cell}
       Probs: {1x9 cell}
     Connect: {1x9 cell}
     FwdTree: {1x10 cell}
    RateTree: {1x10 cell}
```

Price FixedBondOption Instruments

Use `price` to compute the prices and sensitivities for the `FixedBondOption` instruments.

```
[Price, outPR] = price(HWTreePricer,FixedBOptionEuro,["all"])
```

```
Price = 3×1

   10.7571
   10.2111
    9.6508


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma      Vega
    _______    _______    _______    _______

    10.757    -178.78    2207.3    308.87


ans=1×4 table
    Price      Delta     Gamma      Vega
    _______    _______    _______    _______

    10.211    -173.94    2168.8    300.36


ans=1×4 table
    Price      Delta     Gamma      Vega
    _______    _______    _______    _______

    9.6508    -168.87    2127.8    291.34
```

Price a FixedBondOption Instrument on a Stepped FixedBond Using Hull-White Model and Hull-White Tree Pricer

This example shows the workflow to price a `FixedBondOption` instrument on a stepped `FixedBond` instrument when you use a `HullWhite` model and an `IRTree` pricing method.

Create Stepped FixedBond Instrument Object

Use `fininstrument` to create a stepped `FixedBond` instrument object as the underlying bond.

```
Maturity = datetime(2027,1,1);
Period = 1;
CDates = datetime([2022,1,1 ; 2027,1,1]);
CRates = [.022; .027];
CouponRate = timetable(CDates,CRates);

SBond = fininstrument("FixedBond",'Maturity',Maturity,'CouponRate',CouponRate,'Period',Period,'Na
```

```
SBond =
  FixedBond with properties:

                 CouponRate: [2x1 timetable]
                     Period: 1
                      Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2027
                        Name: "stepped_bond_instrument"
```

Create FixedBondOption Instrument Object

Use `fininstrument` to create a `FixedBondOption` instrument object with European exercise.

```
FixedBOption = fininstrument("FixedBondOption",'ExerciseDate',datetime(2026,1,1),'Strike',90,'Bo
```

```
FixedBOption =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 01-Jan-2026
           Strike: 90
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option_european"
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
ZeroRates = [0.0055 0.0063 0.0071 0.0083 0.0099 0.0131 0.0178 0.0262 0.0343 0.0387]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding);
```

Create HullWhite Model Object

Use finmodel to create a HullWhite model object.

```
VolCurve = 0.15;
AlphaCurve = 0.03;

HWModel = finmodel("HullWhite",'Alpha',AlphaCurve,'Sigma',VolCurve)
```

```
HWModel =
  HullWhite with properties:

    Alpha: 0.0300
    Sigma: 0.1500
```

Create IRTree Pricer Object

Use finpricer to create an IRTree pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [10x1 datetime]
           Model: [1x1 finmodel.HullWhite]
   DiscountCurve: [1x1 ratecurve]
```

```
HWTreePricer.Tree
```

```
ans = struct with fields:
        tObs: [0 1 2 3.0027 4.0027 5.0027 6.0027 7.0055 8.0055 9.0055]
        dObs: [01-Jan-2018    01-Jan-2019    01-Jan-2020    ...    ]
      CFlowT: {1x10 cell}
       Probs: {1x9 cell}
     Connect: {1x9 cell}
     FwdTree: {1x10 cell}
    RateTree: {1x10 cell}
```

Price FixedBondOption Instrument

Use price to compute the price and sensitivities for the FixedBondOption instrument.

```
[Price, outPR] = price(HWTreePricer,FixedBOption,"all")
```

```
Price = 12.2717
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta     Gamma      Vega
    _____    _____    _____    _____

    12.272    -130.1    1438.4    100.91
```

Price Fixed Bond Option Instrument Using Hull-White Model and IRMonteCarlo Pricer

This example shows the workflow to price a `FixedBondOption` instrument when using a `HullWhite` model and an `IRMonteCarlo` pricing method.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",'Maturity',datetime(2022,9,15),'CouponRate',.021,'Period',1
```

```
BondInst =
  FixedBond with properties:

                 CouponRate: 0.0210
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "bond_instrument"
```

Create `FixedBondOption` Instrument Object

Use `fininstrument` to create a `FixedBondOption` instrument object.

```
FixedBOptionEuro = fininstrument("FixedBondOption",'ExerciseDate',datetime(2020,3,15),'Strike',98
```

```
FixedBOptionEuro =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Mar-2020
           Strike: 98
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option_european"
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.32,'Sigma',0.49)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.3200
    Sigma: 0.4900
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",'Model',HullWhiteModel,'DiscountCurve',myRC,'SimulationDates
```

```
outPricer =
  HWMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [15-Mar-2019    15-Sep-2019    15-Mar-2020    ...    ]
              Model: [1x1 finmodel.HullWhite]
```

Price FixedBondOption Instrument

Use `price` to compute the price and sensitivities for the `FixedBondOption` instrument.

```
[Price,outPR] = price(outPricer,FixedBOptionEuro,["all"])
```

```
Price = 24.0750
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma      Vega
    _____    _______    _____    _____

    24.075    -166.42    1456.2    20.329
```

More About

Bond Option

A bond option gives the holder 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.

The `FixedBondOption` instrument supports two 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.

For more information, see “Bond Options” on page 2-6.

Tips

After creating a `FixedBondOption` instrument object, you can use `setExercisePolicy` to change the size of the options. For example, if you have the following instrument:

```
FixedBOption = fininstrument("FixedBondOption",'ExerciseDate',datetime(2022,9,15),'Strike',98,'Bo
```

To modify the `FixedBondOption` instrument's size by changing the `ExerciseStyle` from `"European"` to `"American"`, use `setExercisePolicy`:

```
FixedBOption = setExercisePolicy(FixedBOption,[datetime(2021,1,1) datetime(2022,1,1)],100,'Americ
```

Version History

Introduced in R2020a

See Also

Functions

`FixedBond` | `finmodel` | `finpricer`

Topics

“Price Portfolio of Bond and Bond Option Instruments” on page 2-172

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

FloatBond

`FloatBond` instrument object

Description

Create and price a `FloatBond` instrument object using this workflow:

1. Use `fininstrument` to create a `FloatBond` instrument object.
2. Use `ratecurve` to specify a curve model for the `FloatBond` instrument or use a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model.
3. Choose a pricing method.
   - When using a `ratecurve`, use `finpricer` to specify a `Discount` pricing method for one or more `FloatBond` instruments.
   - When using a `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` model, use `finpricer` to specify an `IRTree` pricing method for one or more `FloatBond` instruments.
   - When using a `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model, use `finpricer` to specify an `IRMonteCarlo` pricing method for one or more `FloatBond` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `FloatBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
FloatBondObj = fininstrument(InstrumentType,'Spread',spread_value,'
Maturity',maturity_date)
FloatBondObj = fininstrument( ___ ,Name,Value)
```

Description

`FloatBondObj = fininstrument(InstrumentType,'Spread',spread_value,'Maturity',maturity_date)` creates a `FloatBond` object by specifying `InstrumentType` and sets the properties on page 11-2790 for the required name-value pair arguments `Spread` and `Maturity`.

The `FloatBond` instrument supports a vanilla floating rate note and an amortizing floating rate note. For more information, see “Floating-Rate Note” on page 11-2801.

`FloatBondObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2790 using additional name-value pairs in addition to the required arguments in the previous syntax. For

example, `FloatBondObj = fininstrument("FloatBond",'Spread',0.6,'Maturity',datetime(2019,1,30),'Basis',1,'Principal',100,'FirstCouponDate',datetime(2016,1,30),'EndMonthRule',true,'Name',"float_bond_instrument")` creates a `FloatBond` instrument with a spread of 0.6 and a maturity of January 30, 2019. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"FloatBond"` | string array with values of `"FloatBond"` | character vector with value `'FloatBond'` | cell array of character vectors with values of `'FloatBond'`

Instrument type, specified as a string with the value of `"FloatBond"`, a character vector with the value of `'FloatBond'`, an `NINST`-by-`1` string array with values of `"FloatBond"`, or an `NINST`-by-`1` cell array of character vectors with values of `'FloatBond'`.

Data Types: `char` | `cell` | `string`

`FloatBond` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `FloatBondObj = fininstrument("FloatBond",'Spread',0.6,'Maturity',datetime(2019,1,30),'Basis',1,'Principal',100,'FirstCouponDate',datetime(2016,1,30),'EndMonthRule',true,'Name',"float_bond_instrument")`

Required `FloatBond` Name-Value Pair Arguments

`Spread` — Decimal value over the reference rate

scalar nonnegative decimal | vector of nonnegative decimals

Decimal value over the reference rate, specified as the comma-separated pair consisting of `'Spread'` and a scalar nonnegative decimal or an `NINST`-by-`1` vector of nonnegative decimals.

Data Types: `double`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as the comma-separated pair consisting of `'Maturity'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `FloatBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

Optional `FloatBond` Name-Value Pair Arguments

`Reset` — Frequency of payments per year

2 (default) | scalar integer | vector of integers

Frequency of payments per year, specified as the comma-separated pair consisting of `'Reset'` and a scalar integer or an `NINST`-by-`1` vector of integers. Values for `Reset` are: `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day count basis
`[0 0]` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer or an `NINST`-by-`1` using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Notional principal amount or principal value schedule, specified as the comma-separated pair consisting of `'Principal'` and a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

`Principal` accepts a `timetable`, where the first column is dates and the second column is the associated notional principal value. The date indicates the last day that the principal value is valid.

Note If you are creating one or more `FloatBond` instruments and use a timetable, the timetable specification applies to all of the `FloatBond` instruments. `Principal` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `double` | `timetable`

`ProjectionCurve` — Rate curve for projecting floating cash flows
`ratecurve.empty` (default) | scalar `ratecurve` object | vector of `ratecurve` objects

Rate curve for projecting floating cash flows, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects. You must create this object using `ratecurve`.

Data Types: `object`

`ResetOffset` — Lag in rate setting
`0` (default) | scalar numeric | numeric vector

Lag in rate setting, specified as the comma-separated pair consisting of `'ResetOffset'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`LatestFloatingRate` — Latest floating rate
`[]` (default) | scalar decimal | vector of decimals

Latest floating rate for the `FloatBond` object, specified as the comma-separated pair consisting of `'LatestFloatingRate'` and a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`DaycountAdjustedCashFlow` — Flag to adjust cash flows based on actual period day count
`false` (default) | scalar logical value of `true` or `false` | vector of logical values with `true` or `false`

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'DaycountAdjustedCashFlow'` and a scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and dates using an `NINST`-by-`1` vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
FloatBondObj = fininstrument("floatbond",'Spread',100,'Maturity',datetime(2025,12,15),'Holidays',
```

To support existing code, `FloatBond` also accepts serial date numbers as inputs, but they are not recommended.

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`true` (in effect) (default) | scalar logical with value of `true` or `false` | vector of logicals with values of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

- If you set `EndMonthRule` to `false`, the software ignores the rule, meaning that a payment date is always the same numerical day of the month.
- If you set `EndMonthRule` to `true`, the software sets the rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
`NaT` (default) | datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `FloatBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `IssueDate` property is stored as a datetime.

`FirstCouponDate` — Irregular first coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `FloatBond` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `FirstCouponDate` property is stored as a datetime.

`LastCouponDate` — Irregular last coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `FloatBond` also accepts serial date numbers as inputs, but they are not recommended.

If you specify `LastCouponDate` but not `FirstCouponDate`, `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify `LastCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `LastCouponDate` property is stored as a datetime.

`StartDate` — Forward starting date of payments
`NaT` (default) | datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `FloatBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `StartDate` property is stored as a datetime.

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Spread` — Number of basis points over the reference rate
scalar nonnegative numeric | nonnegative numeric vector

Number of basis points over the reference rate, returned as a scalar nonnegative numeric or an `NINST`-by-`1` nonnegative numeric vector.

Data Types: `double`

`Maturity` — Maturity date
datetime | vector of datetimes

Maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Reset` — Frequency of payment per year
`1` (default) | scalar integer | vector of integers

Coupons per year, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Principal` — Notional principal amount or principal value schedules
`100` (default) | scalar numeric | numeric vector | timetable

Notional principal amount or principal value schedules, returned as a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

Data Types: `timetable` | `double`

`ProjectionCurve` — Rate curve used in generating future cash flows
`ratecurve.empty` (default) | scalar `ratecurve` object | vector of `ratecurve` objects

Rate curve to be used in projecting the future cash flows, returned as a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects.

Data Types: `object`

`ResetOffset` — Lag in rate setting
`0` (default) | scalar numeric | numeric vector

Lag in rate setting, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`LatestFloatingRate` — Latest floating rate for `FloatBond`
`[ ]` (default) | scalar decimal | vector of decimals

Latest floating rate for `FloatBond`, returned as a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`DaycountAdjustedCashFlow` — Flag to adjust cash flows based on actual period day count
`false` (default) | scalar logical value of `true` or `false` | vector of logical values with `true` or `false`

Flag to adjust cash flows based on actual period day count, returned as scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | scalar string | string array

Business day conventions, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: string

Holidays — Holidays used in computing business days
NaT (default) | datetimes

Holidays used in computing business days, returned as an NINST-by-1 vector of datetimes.

Data Types: datetime

EndMonthRule — End-of-month rule flag for generating dates when Maturity is end-of-month date for month with 30 or fewer days
true (in effect) (default) | scalar logical with value of true or false | vector of logicals with values of true or false

End-of-month rule flag for generating dates when Maturity is an end-of-month date for a month with 30 or fewer days, returned as a scalar logical or an NINST-by-1 vector of logical values.

Data Types: logical

IssueDate — Bond issue date
NaT (default) | datetime | vector of datetimes

Bond issue date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: datetime

FirstCouponDate — Irregular first coupon date
NaT (default) | datetime | vector of datetimes

Irregular first coupon date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: datetime

LastCouponDate — Irregular last coupon date
NaT (default) | datetime | vector of datetimes

Irregular last coupon date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: datetime

StartDate — Forward starting date of payments
NaT (default) | datetime | vector of datetimes

Forward starting date of payments, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: datetime

Name — User-defined name for instrument
" " (default) | string | string array

User-defined name for the instrument, returned as a string or an NINST-by-1 string array.

Data Types: string

Object Functions

cashflows — Compute cash flow for FixedBond, FloatBond, Swap, FRA, STIRFuture, OISFuture, OvernightIndexedSwap, or Deposit instrument

Examples

Price Vanilla Float Bond Instrument Using `ratecurve` and Discount Pricer

This example shows the workflow to price a vanilla `FloatBond` instrument when you use a `ratecurve` and a `Discount` pricing method.

Create `FloatBond` Instrument Object

Use `fininstrument` to create a vanilla `FloatBond` instrument object.

```
FloatB = fininstrument("FloatBond",'Maturity',datetime(2022,9,15),'Spread',0.025,'Reset',2,'Basis
```

```
FloatB =
  FloatBond with properties:

                     Spread: 0.0250
            ProjectionCurve: [0x0 ratecurve]
                ResetOffset: 0
                      Reset: 2
                      Basis: 1
               EndMonthRule: 0
                  Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
         LatestFloatingRate: NaN
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
                   Maturity: 15-Sep-2022
                       Name: "float_bond_instrument"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
```

```
                Basis: 0
                Dates: [10x1 datetime]
                Rates: [10x1 double]
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `FloatBond` Instrument

Use `price` to compute the price and sensitivities for the vanilla `FloatBond` instrument.

```
[Price, outPR] = price(outPricer, FloatB,["all"])
```

```
Price = 109.8322
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    _________

    109.83    0.0021981
```

Price Multiple Vanilla Float Bond Instruments Using `ratecurve` and Discount Pricer

This example shows the workflow to price multiple vanilla `FloatBond` instruments when you use a `ratecurve` and a `Discount` pricing method.

Create `FloatBond` Instrument Object

Use `fininstrument` to create a vanilla `FloatBond` instrument object for three Float Bond instruments.

```
FloatB = fininstrument("FloatBond",'Maturity',datetime([2022,9,15 ; 2022,9,15 ; 2022,9,15]),'Spre
```

```
FloatB=3×1 object
  3x1 FloatBond array with properties:

    Spread
    ProjectionCurve
    ResetOffset
    Reset
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    LatestFloatingRate
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    Name
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:
```

```
    DiscountCurve: [1x1 ratecurve]
```

Price FloatBond Instruments

Use `price` to compute the prices and sensitivities for the vanilla `FloatBond` instruments.

```
[Price, outPR] = price(outPricer, FloatB,["all"])
```

```
Price = 3×1

  109.8322
  219.6644
  329.4965


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    _________

    109.83    0.0021981


ans=1×2 table
    Price       DV01
    ______    _________

    219.66    0.0043961


ans=1×2 table
    Price      DV01
    _____    _________

    329.5    0.0065942
```

Price Amortizing Float Bond Instrument Using `ratecurve` and Discount Pricer

This example shows the workflow to price an amortizing `FloatBond` instrument when you use a `ratecurve` and a `Discount` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
```

```
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding);
```

Create FloatBond Instrument Object

Use `fininstrument` to create an amortizing `FloatBond` instrument object.

```
Maturity = datetime(2024,1,1);
Spread = 0.02;
Reset = 1;
ADates = datetime([2020,1,1 ; 2024,1,1]);
APrincipal = [100; 80];
Principal = timetable(ADates,APrincipal);
Floatamort = fininstrument("FloatBond",'Maturity',Maturity,'Spread',Spread,'Reset',Reset,'Projec
```

```
Floatamort =
  FloatBond with properties:

                     Spread: 0.0200
            ProjectionCurve: [1x1 ratecurve]
                ResetOffset: 0
                      Reset: 1
                      Basis: 0
               EndMonthRule: 1
                  Principal: [2x1 timetable]
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
         LatestFloatingRate: NaN
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
                   Maturity: 01-Jan-2024
                       Name: ""
```

Create Discount Pricer Object

Use `finpricer` to create an `Discount` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',ZeroCurve)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price FloatBond Instrument

Use `price` to compute the price and sensitivities for the vanilla `FloatBond` instrument.

```
[Price, outPR] = price(outPricer,Floatamort,["all"])
```

```
Price = 110.1101
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    _________

    110.11    0.0033187
```

Price Float Bond Instrument Using Hull-White Model and IRMonteCarlo Pricer

This example shows the workflow to price a `FloatBond` instrument when using a `HullWhite` model and an `IRMonteCarlo` pricing method.

Create `FloatBond` Instrument Object

Use `fininstrument` to create a `FloatBond` instrument object.

```
FloatB = fininstrument("FloatBond",'Maturity',datetime(2022,9,15),'Spread',0.025,'Reset',2,'Basis
```

```
FloatB =
  FloatBond with properties:

                     Spread: 0.0250
            ProjectionCurve: [0x0 ratecurve]
                ResetOffset: 0
                      Reset: 2
                      Basis: 1
               EndMonthRule: 0
                  Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
         LatestFloatingRate: NaN
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
                   Maturity: 15-Sep-2022
                       Name: "float_bond_instrument"
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.32,'Sigma',0.49)
```

```
HullWhiteModel =
  HullWhite with properties:
```

```
    Alpha: 0.3200
    Sigma: 0.4900
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",'Model',HullWhiteModel,'DiscountCurve',myRC,'SimulationDates
```

```
outPricer =
  HWMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jan-2021    ...    ]
              Model: [1x1 finmodel.HullWhite]
```

Price FloatBond Instrument

Use `price` to compute the price and sensitivities for the `FloatBond` instrument.

```
[Price,outPR] = price(outPricer,FloatB,["all"])
```

```
Price = 109.1227
```

```
outPR =
  priceresult with properties:
```

```
       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma     Vega
    ______    _______    ______    ____

    109.12    -19.033    50.224     0
```

Price Vanilla FloatBond Instrument Using a Hull-White Model and IRTree Pricer

This example shows the workflow to price a vanilla `FloatBond` instrument when using a `HullWhite` model and an IRTree pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding);
```

Create FloatBond Instrument Object

Use `fininstrument` to create a vanilla `FloatdBond` instrument object.

```
Spread = 0.03;
Reset = 1;
Maturity = datetime(2024,1,1);
Period = 1;
Float = fininstrument("FloatBond",'Maturity',Maturity,'Spread',Spread,'Reset',Reset,'ProjectionCu
```

```
Float =
  FloatBond with properties:

                     Spread: 0.0300
            ProjectionCurve: [1x1 ratecurve]
                ResetOffset: 0
                      Reset: 1
                      Basis: 0
               EndMonthRule: 1
                  Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
         LatestFloatingRate: NaN
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
```

```
                Maturity: 01-Jan-2024
                    Name: ""
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;

HWModel = finmodel("HullWhite",'alpha',AlphaCurve,'sigma',VolCurve);
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

             Tree: [1x1 struct]
        TreeDates: [10x1 datetime]
            Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

Price FloatBond Instrument

Use `price` to compute the price and sensitivities for the vanilla `FloatBond` instrument.

```
[Price, outPR] = price(HWTreePricer,Float,["all"])
```

```
Price = 117.4686

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma     Vega
    ______    _______    ______    ____

    117.47    -60.007    315.09     0
```

More About

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.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `FloatBond` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`FloatBondOption` | `finmodel` | `finpricer` | `timetable`

Topics

“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Compute LIBOR Fallback” on page 2-192
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

FloatBondOption

`FloatBondOption` instrument object

Description

Create and price a `FloatBondOption` instrument object for one or more Float Bond Option instruments using this workflow:

1. Use `fininstrument` to create an `FloatBondOption` instrument object for one or more Float Bond Option instruments.
2. Use `finmodel` to specify a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model for the `FloatBondOption` instrument object.
3. Choose a pricing method.
   - When using a `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` model, use `finpricer` to specify an `IRTree` pricing method for one or more `FloatBondOption` instruments.
   - When using a `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model, use `finpricer` to specify an `IRMonteCarlo` pricing method for one or more `FloatBondOption` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods `FloatBondOption` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
FloatBondOptionObj = fininstrument(InstrumentType,'Strike',strike_value,'
ExerciseDate',exercise_date,'Bond',bond_obj)
FloatBondOptionObj = fininstrument( ___ ,Name,Value)
```

Description

`FloatBondOptionObj = fininstrument(InstrumentType,'Strike',strike_value,'ExerciseDate',exercise_date,'Bond',bond_obj)` creates a `FloatBond` object for one or more Float Bond Option instruments by specifying `InstrumentType` and sets properties on page 11-2805 using the required name-value pair arguments `Strike`, `ExerciseDate`, and `Bond`.

`FloatBondOptionObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2805 using additional name-value pair arguments in addition to the required arguments in the previous syntax. For example, `FloatBondOptionObj = fininstrument("FloatBondOption",'Strike',100,'ExerciseDate',datetime(2019,1,30),'Bond',bond_obj,'OptionType','put','ExerciseStyle',"american",'Name',"floa`

t_bond_option") creates a FloatBondOption instrument with a strike of 100 and an American exercise. You can specify multiple name-value pair arguments.

Input Arguments

InstrumentType — Instrument type

string with value "FloatBondOption" | string array with values of "FloatBondOption" | character vector with value 'FloatBondOption' | cell array of character vectors with values of 'FloatBondOption'

Instrument type, specified as a string with the value of "FloatBondOption", a character vector with the value of 'FloatBondOption', an NINST-by-1 string array with values of "FloatBondOption", or an NINST-by-1 cell array of character vectors with values of 'FloatBondOption'.

Data Types: char | cell | string

FloatBondOption Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: FloatBondOptionObj = fininstrument("FloatBondOption",'Strike',100,'ExerciseDate',datetime(2019,1,30),'Bond',bond_obj,'OptionType','put','ExerciseStyle',"american",'Name',"float_bond_option")

Required FloatBondOption Name-Value Pair Arguments

Strike — Option strike value

nonnegative value | vector of nonnegative values

Option strike value, specified as the comma-separated pair consisting of 'Strike' and a scalar nonnegative value or an NINST-by-1 vector of nonnegative values.

Data Types: double

ExerciseDate — Option exercise dates

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of 'ExerciseDate' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, FloatBondOption also accepts serial date numbers as inputs, but they are not recommended.

- For a European option, there is only one ExerciseDate on the option expiry date.
- For a Bermudan option, there is a 1-by-NSTRIKES vector of exercise dates.
- For an American option, the option can be exercised between ValuationDate of the stock tree and the single listed ExerciseDate.

If you use date character vectors or strings, the format must be recognizable by datetime because the Maturity property is stored as a datetime.

Bond — Underlying float bond
`FloatBond` object | vector of `FloatBond` objects

Underlying float bond, specified as the comma-separated pair consisting of `'Bond'` and the name of a `FloatBond` object or an `NINST`-by-`1` vector of `FloatBond` objects.

Data Types: `object`

Optional FloatBondOption Name-Value Pair Arguments

`OptionType` — Definition of option
`"call"` (default) | string with value `"call"` or `"put"` | string array with values of `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values of `'call'` or `'put'`

Definition of option, specified as the comma-separated pair consisting of `'OptionType'` and a scalar character vector or a string or an `NINST`-by-`1` cell array of character vectors or string array using `'call'` or `'put'`.

Data Types: `char` | `cell` | `string`

`ExerciseStyle` — Option type
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with values of `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'` | cell array of character vectors with values of `'European'`, `'American'`, or `'Bermudan'`

Option type, specified as the comma-separated pair consisting of `'ExerciseStyle'` and a scalar character vector or string or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `string` | `cell` | `char`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`InstrumentType` — Instrument type
string with value `"FloatBondOption"` | string array with values of `"FloatBondOption"`

Instrument type, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

`Strike` — Option strike value
nonnegative value | nonnegative value

Option strike value, returned as a scalar nonnegative value or an `NINST`-by-`1` vector of nonnegative values.

Data Types: `double`

ExerciseDate — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: datetime

OptionType — Definition of option
"call" (default) | string with value "call" or "put" | string array with values of "call" or "put"

Definition of option, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

ExerciseStyle — Option type
"European" (default) | string with value "European", "American", or "Bermudan" | string array with values of "European", "American", or "Bermudan"

Option type, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Bond — Underlying float bond
FloatBond object | vector of FloatBond objects

Underlying float bond, returned as a scalar FloatBond object or an NINST-by-1 vector of FloatBond objects.

Data Types: object

Name — User-defined name for instrument
" " (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Object Functions

setExercisePolicy Set exercise policy for FixedBondOption, FloatBondOption, or Vanilla instrument

Examples

Price a Float Bond Option Instrument Using Hull-White Model and IRTree Pricer

This example shows the workflow to price a FloatBondOption instrument when you use a HullWhite model and an IRTree pricing method.

Create FloatBond Instrument Object

Use fininstrument to create a FloatBond instrument object as the underlying bond.

```
BondInst = fininstrument("FloatBond",'Maturity',datetime(2030,9,15),'Spread',0.021,'Name',"bond_
```

```
BondInst =
  FloatBond with properties:
```

```
                    Spread: 0.0210
              ProjectionCurve: [0x0 ratecurve]
                  ResetOffset: 0
                        Reset: 2
                        Basis: 0
                 EndMonthRule: 1
                    Principal: 100
      DaycountAdjustedCashFlow: 0
         BusinessDayConvention: "actual"
            LatestFloatingRate: NaN
                     Holidays: NaT
                    IssueDate: NaT
              FirstCouponDate: NaT
               LastCouponDate: NaT
                    StartDate: NaT
                     Maturity: 15-Sep-2030
                         Name: "bond_instrument"
```

Create `FloatBondOption` Instrument Objects

Use `fininstrument` to create three callable `FloatBondOption` instrument objects with European, American, and Bermudan exercise.

```
FloatBOptionEuro = fininstrument("FloatBondOption",'ExerciseDate',datetime(2029,9,15),'Strike',98
```

```
FloatBOptionEuro =
  FloatBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2029
           Strike: 98
             Bond: [1x1 fininstrument.FloatBond]
             Name: "float_bond_option_european"
```

```
FloatBOptionAmerican = fininstrument("FloatBondOption",'ExerciseDate',datetime(2029,9,15),'Strike
```

```
FloatBOptionAmerican =
  FloatBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2029
           Strike: 98
             Bond: [1x1 fininstrument.FloatBond]
             Name: "float_bond_option_american"
```

```
FloatBOptionBermudan = fininstrument("FloatBondOption",'ExerciseDate',[datetime(2025,9,15) , date
```

```
FloatBOptionBermudan =
  FloatBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "bermudan"
     ExerciseDate: [15-Sep-2025    15-Sep-2029]
           Strike: [98 100]
```

```
              Bond: [1x1 fininstrument.FloatBond]
              Name: "float_bond_option_bermudan"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2024,9,15);
Type = 'zero';
ZeroTimes = [calyears([1:10])]';
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2024
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create a `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.01,'Sigma',0.05)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0100
    Sigma: 0.0500
```

Create `IRTree` Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
CFdates = cfdates(Settle, BondInst.Maturity, BondInst.Reset, BondInst.Basis);
HWTreePricer = finpricer("IRTree",'Model',HullWhiteModel,'DiscountCurve',myRC,'TreeDates',CFdates
```

```
HWTreePricer =
  HWBKTree with properties:

             Tree: [1x1 struct]
        TreeDates: [12x1 datetime]
            Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

```
HWTreePricer.Tree

ans = struct with fields:
        tObs: [0 0.4959 1 1.4959 2 2.4959 3 3.4986 4.0027 4.4986 5.0027 ... ]
        dObs: [15-Sep-2024    15-Mar-2025    15-Sep-2025    ...    ]
      CFlowT: {1x12 cell}
       Probs: {1x11 cell}
     Connect: {1x11 cell}
     FwdTree: {1x12 cell}
    RateTree: {1x12 cell}
```

Price FixedBondOption Instruments

Use `price` to compute the price and sensitivities for the two `FixedBondOption` instruments.

```
[Price, outPR] = price(HWTreePricer,FloatBOptionEuro,["all"])

Price = 3.8040

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×4 table
    Price     Delta     Gamma        Vega
    _____    _______    ______    ___________

    3.804    -20.465    110.75    -2.6645e-11


[Price, outPR] = price(HWTreePricer,FloatBOptionAmerican,["all"])

Price = 14.1700

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×4 table
    Price     Delta     Gamma     Vega
    _____    _______    ______    ____

    14.17    -38.981    160.87     0


[Price, outPR] = price(HWTreePricer,FloatBOptionBermudan,["all"])

Price = 12.0676
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma        Vega
    ______    _______    ______    ___________

    12.068    -39.402    161.55    -2.8422e-10
```

Price Multiple Float Bond Option Instruments Using Hull-White Model and IRTree Pricer

This example shows the workflow to price multiple `FloatBondOption` instruments when you use a `HullWhite` model and an `IRTree` pricing method.

Create FloatBond Instrument Object

Use `fininstrument` to create a `FloatBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FloatBond",'Maturity',datetime(2030,9,15),'Spread',0.021,'Name',"bond_
```

```
BondInst =
  FloatBond with properties:

                     Spread: 0.0210
            ProjectionCurve: [0x0 ratecurve]
                ResetOffset: 0
                      Reset: 2
                      Basis: 0
               EndMonthRule: 1
                  Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
         LatestFloatingRate: NaN
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2030
                        Name: "bond_instrument"
```

Create FloatBondOption Instrument Objects

Use `fininstrument` to create a `FloatBondOption` instrument object with European exercise for three Float Bond Option instruments.

```
FloatBOptionEuro = fininstrument("FloatBondOption",'ExerciseDate',datetime([2030,9,15 ; 2029,09,
```

```
FloatBOptionEuro=3×1 object
  3x1 FloatBondOption array with properties:
```

```
    OptionType
    ExerciseStyle
    ExerciseDate
    Strike
    Bond
    Name
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2024,9,15);
Type = 'zero';
ZeroTimes = [calyears([1:10])]';
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2024
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create a HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.01,'Sigma',0.05)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0100
    Sigma: 0.0500
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
CFdates = cfdates(Settle, BondInst.Maturity, BondInst.Reset, BondInst.Basis);
HWTreePricer = finpricer("IRTree",'Model',HullWhiteModel,'DiscountCurve',myRC,'TreeDates',CFdates
```

```
HWTreePricer =
  HWBKTree with properties:
```

```
            Tree: [1x1 struct]
       TreeDates: [12x1 datetime]
           Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

```
HWTreePricer.Tree
```

```
ans = struct with fields:
        tObs: [0 0.4959 1 1.4959 2 2.4959 3 3.4986 4.0027 4.4986 5.0027 ... ]
        dObs: [15-Sep-2024    15-Mar-2025    15-Sep-2025    ...    ]
      CFlowT: {1x12 cell}
       Probs: {1x11 cell}
     Connect: {1x11 cell}
     FwdTree: {1x12 cell}
    RateTree: {1x12 cell}
```

Price FixedBondOption Instruments

Use `price` to compute the prices and sensitivities for the `FixedBondOption` instruments.

```
[Price, outPR] = price(HWTreePricer,FloatBOptionEuro,["all"])
```

```
Price = 3×1

    1.8081
    2.8617
    3.9097
```

```
outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma        Vega
    _____    _______    _____    __________

    1.8081    -10.854    65.153    4.4409e-12
```

```
ans=1×4 table
    Price      Delta     Gamma        Vega
    _____    _______    _____    __________

    2.8617    -15.751    87.167    -1.7764e-11
```

```
ans=1×4 table
    Price      Delta     Gamma        Vega
    _____    _______    _____    __________
```

```
    3.9097    -20.493    108.64    -7.1054e-11
```

Price Float Bond Option Instrument Using Hull-White Model and IRMonteCarlo Pricer

This example shows the workflow to price a `FloatdBondOption` instrument when using a `HullWhite` model and an `IRMonteCarlo` pricing method.

Create `FloatBond` Instrument Object

Use `fininstrument` to create a `FloatBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FloatBond",'Maturity',datetime(2030,9,15),'Spread',0.021,'Name',"bond_
```

```
BondInst =
  FloatBond with properties:

                    Spread: 0.0210
            ProjectionCurve: [0x0 ratecurve]
                ResetOffset: 0
                      Reset: 2
                      Basis: 0
               EndMonthRule: 1
                  Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
         LatestFloatingRate: NaN
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
                   Maturity: 15-Sep-2030
                       Name: "bond_instrument"
```

Create `FloatBondOption` Instrument Object

Use `fininstrument` to create a `FloatBondOption` instrument object.

```
FloatBOptionEuro = fininstrument("FloatBondOption",'ExerciseDate',datetime(2020,3,15),'Strike',98
```

```
FloatBOptionEuro =
  FloatBondOption with properties:

      OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Mar-2020
           Strike: 98
             Bond: [1x1 fininstrument.FloatBond]
             Name: "float_bond_option_european"
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.32,'Sigma',0.49)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.3200
    Sigma: 0.4900
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",'Model',HullWhiteModel,'DiscountCurve',myRC,'SimulationDates
```

```
outPricer =
  HWMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [15-Mar-2019    15-Sep-2019    15-Mar-2020    ...    ]
              Model: [1x1 finmodel.HullWhite]
```

Price FloatBondOption Instrument

Use `price` to compute the price and sensitivities for the `FloatBondOption` instrument.

```
[Price,outPR] = price(outPricer,FloatBOptionEuro,["all"])
```

```
Price = 18.2369
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma     Vega
    _____    ________    _____    _____

    18.237    -104.22    788.7    -13.949
```

More About

Floating-Rate Note Option

A floating-rate note option gives the option holder the right to sell the option back to the issuer (put) or to redeem an option (call) at a specific price and on a specific date.

Financial Instruments Toolbox 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 only exercise it on predetermined dates, usually monthly

For more information, see “Bond Options” on page 2-6.

Tips

After creating a `FloatBondOption` instrument object, you can use `setExercisePolicy` to change the size of the options. For example, consider the following instrument:

```
FloatBOption = fininstrument("FloatBondOption",'ExerciseDate',datetime(2029,9,15),'Strike',98,'Bo
```

To modify the size of the `FloatBondOption` instrument object by changing the `ExerciseStyle` from `"European"` to `"American"`, use `setExercisePolicy`:

```
FloatBOption = setExercisePolicy(FloatBOption,[datetime(2021,1,1) datetime(2022,1,1)],100,'Americ
```

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `FloatBondOption` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`FloatBond` | `finmodel` | `finpricer`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Floor

`Floor` instrument object

Description

Create and price a `Floor` instrument object for one of more Floor instruments using this workflow:

1. Use `fininstrument` to create a `Floor` instrument object for one of more Floor instruments.
2. Use `finmodel` to specify a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `Black`, `Normal`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model for the `Floor` instrument object.
3. Choose a pricing method.
   - When using a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `Black`, or `Normal` model, use `finpricer` for pricing one or more `Floor` instruments and specify:
     - A `Normal` pricer when using a `Normal` model.
     - A `Black` pricer when using a `Black` model.
     - A `HullWhite` pricer when using a `HullWhite` model.
     - An `IRTree` pricer when using a `BlackKarasinski` or `BlackDermanToy` model.
   - When using a `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model, use `finpricer` to specify an `IRMonteCarlo` pricing method for one or more `Floor` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Floor` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
FloorOpt = fininstrument(InstrumentType,'Strike',strike_value,'
Maturity',maturity_date)
FloorOpt = fininstrument( ___ ,Name,Value)
```

Description

`FloorOpt = fininstrument(InstrumentType,'Strike',strike_value,'Maturity',maturity_date)` creates a `Floor` object for one of more Floor instruments by specifying `InstrumentType` and sets the properties on page 11-2821 for the required name-value pair arguments`Strike` and `Maturity`.

The `Floor` instrument supports vanilla and amortizing floors.

`FloorOpt = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2821 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `FloorOpt = fininstrument("floor",'Strike',100,'Maturity',datetime(2019,1,30),'Reset',4,'Principal',100,'ResetOffset',1,'Basis',4,'DaycountAdjustedCashFlow',true,'BusinessDayConvention',"follow",'ProjectionCurve',ratecurve_object,'Name',"floor_option")` creates a `Floor` instrument with a strike of 100 and a maturity of January 30, 2019. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Floor"` | string array with values of `"Floor"` | character vector with value `'Floor'` | cell array of character vectors with values of `'Floor'`

Instrument type, specified as a string with the value of `"Floor"`, a character vector with the value of `'Floor'`, an `NINST`-by-`1` string array with values of `"Floor"`, or an `NINST`-by-`1` cell array of character vectors with values of `'Floor'`.

Data Types: `char` | `cell` | `string`

`Floor` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `FloorOpt = fininstrument("floor",'Strike',100,'Maturity',datetime(2019,1,30),'Reset',4,'Principal',100,'ResetOffset',1,'Basis',4,'DaycountAdjustedCashFlow',true,'BusinessDayConvention',"follow",'ProjectionCurve',ratecurve_object,'Name',"floor_option")`

Required `Floor` Name-Value Pair Arguments

`Strike` — Option strike price value

scalar nonnegative decimal | vector of nonnegative decimals

Option strike price value, specified as the comma-separated pair consisting of `'Strike'` and a scalar nonnegative decimal or an `NINST`-by-`1` vector of nonnegative decimals.

Data Types: `double`

`Maturity` — Floor maturity date

datetime array | string array | date character vector

Floor maturity date, specified as the comma-separated pair consisting of `'ExerciseDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `Floor` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

Optional Floor Name-Value Pair Arguments

`Reset` — Reset frequency payments per year

`1` (default) | scalar numeric with value of `0`, `1`, `2`, `3`, `4`, `6`, or `12` | numeric vector with values of `0`, `1`, `2`, `3`, `4`, `6`, or `12`

Reset frequency payments per year, specified as the comma-separated pair consisting of `'Reset'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Basis` — Day count basis

`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer or an `NINST`-by-`1` vector of integers with the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal amount or principal value schedule

`100` (default) | scalar numeric | numeric vector | timetable

Principal amount or principal value schedule, specified as the comma-separated pair consisting of `'Principal'` and a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

`Principal` accepts a `timetable`, where the first column is dates and the second column is its associated principal value. The date indicates the last day that the principal value is valid.

Note If you are creating one or more `Floor` instruments and use a timetable, the timetable specification applies to all of the `Floor` instruments. `Principal` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `double` | `timetable`

ResetOffset — Lag in rate setting
0 (default) | scalar numeric | numeric vector

Lag in rate setting, specified as the comma-separated pair consisting of 'ResetOffset' and a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

DaycountAdjustedCashFlow — Flag to adjust cash flows based on actual period day count
false (default) | scalar logical value of true or false | vector of logical values of true or false

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of 'DaycountAdjustedCashFlow' and a scalar logical or an NINST-by-1 vector of logicals with values of true or false.

Data Types: logical

BusinessDayConvention — Business day conventions
"actual" (default) | string | string array | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of 'BusinessDayConvention' and a scalar string or character vector or an NINST-by-1 cell array of character vector or string array for a business day convention. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- "actual" — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- "follow" — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- "modifiedfollow" — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- "previous" — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- "modifiedprevious" — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: char | cell | string

Holidays — Holidays used in computing business days
NaT (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of 'Holidays' and dates using an NINST-by-1 vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
FloorOpt = fininstrument("floor",'Strike',100,'Maturity',datetime(2025,12,15),'Holidays',H)
```

To support existing code, Floor also accepts serial date numbers as inputs, but they are not recommended.

`ProjectionCurve` — Rate curve used in generating future cash flows
`ratecurve.empty` (default) | `ratecurve` object | vector of `ratecurve` objects

Rate curve used in projecting the future cash flows, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects. This object is created using `ratecurve`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `object`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Strike` — Option strike price value
scalar nonnegative | vector of nonnegative values

Option strike price value, returned as a scalar nonnegative numeric or an `NINST`-by-`1` vector of nonnegative values.

Data Types: `double`

`Maturity` — Floor maturity date
scalar datetime | vector of datetimes

Floor maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Reset` — Reset frequency payments per year
`1` (default) | scalar numeric | numeric vector

Reset frequency payments per year, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Principal` — Principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Principal amount or principal value schedule, returned as a scalar numeric or an `NINST`-by-`1` numeric vector for principal amount or a timetable for a principal value schedule.

Data Types: `double` | `timetable`

ResetOffset — Lag in rate setting
0 (default) | scalar numeric | numeric vector

Lag in rate setting, returned as a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

DaycountAdjustedCashFlow — Flag to adjust cash flows based on actual period day count
false (default) | scalar logical value of true or false | vector of logical values of true or false

Flag to adjust cash flows based on actual period day count, returned as a scalar logical or an NINST-by-1 vector of logical values with values of true or false.

Data Types: logical

BusinessDayConvention — Business day conventions
"actual" (default) | string | string array

Business day conventions, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Holidays — Holidays used in computing business days
NaT (default) | datetimes

Holidays used in computing business days, returned as an NINST-by-1 vector of datetimes.

Data Types: datetime

ProjectionCurve — Rate curve used in generating future cash flows
ratecurve.empty (default) | scalar ratecurve object | vector of ratecurve objects

Rate curve used in projecting the future cash flows, returned as a scalar ratecurve object or an NINST-by-1 vector of ratecurve objects.

Data Types: object

Name — User-defined name for instrument
" " (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Examples

Price Vanilla Floor Instrument Using Black Model and Black Pricer

This example shows the workflow to price a vanilla Floor instrument when you use a Black model and a Black pricing method.

Create ratecurve Object

Create a ratecurve object using ratecurve for the underlying interest-rate curve for the floor instrument.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Floor` Instrument Object

Use `fininstrument` to create a `Floor` instrument object.

```
FloorOpt = fininstrument("Floor",'Maturity',datetime(2022,9,15),'Strike',0.03,'ProjectionCurve',r
```

```
FloorOpt =
  Floor with properties:

                    Strike: 0.0300
                  Maturity: 15-Sep-2022
               ResetOffset: 0
                     Reset: 1
                     Basis: 0
                 Principal: 100
            ProjectionCurve: [1x1 ratecurve]
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                  Holidays: NaT
                      Name: ""
```

Create `Black` Model Object

Use `finmodel` to create a `Black` model object.

```
BlackModel = finmodel("Black",'Volatility',0.2)
```

```
BlackModel =
  Black with properties:

    Volatility: 0.2000
         Shift: 0
```

Create Black Pricer Object

Use `finpricer` to create a `Black` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
 outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackModel)
```

```
outPricer =
  Black with properties:

            Model: [1x1 finmodel.Black]
    DiscountCurve: [1x1 ratecurve]
```

Price Floor Instrument

Use `price` to compute the price for the `Floor` instrument.

```
[Price, outPR] = price(outPricer,FloorOpt)
```

```
Price = 8.0878
```

```
outPR =
  priceresult with properties:

       Results: [1x1 table]
    PricerData: []
```

Price Multiple Vanilla Floor Instruments Using Black Model and Black Pricer

This example shows the workflow to price multiple vanilla `Floor` instruments when you use a `Black` model and a `Black` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `floor` instrument.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 0
              Dates: [10x1 datetime]
              Rates: [10x1 double]
             Settle: 15-Sep-2018
       InterpMethod: "linear"
```

```
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Floor Instrument Object

Use `fininstrument` to create a `Floor` instrument object for three Florr instruments.

```
FloorOpt = fininstrument("Floor",'Maturity',datetime([2022,9,15 ; 2022,9,15 ; 2022,9,15]),'Strike
```

```
FloorOpt=3×1 object
  3x1 Floor array with properties:

    Strike
    Maturity
    ResetOffset
    Reset
    Basis
    Principal
    ProjectionCurve
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    Name
```

Create Black Model Object

Use `finmodel` to create a `Black` model object.

```
BlackModel = finmodel("Black",'Volatility',0.2)
```

```
BlackModel =
  Black with properties:

    Volatility: 0.2000
         Shift: 0
```

Create Black Pricer Object

Use `finpricer` to create a `Black` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
 outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackModel)
```

```
outPricer =
  Black with properties:

              Model: [1x1 finmodel.Black]
    DiscountCurve: [1x1 ratecurve]
```

Price Floor Instruments

Use `price` to compute the prices for the `Floor` instruments.

```
[Price, outPR] = price(outPricer,FloorOpt)
```

```
Price = 3×1

    8.0878
   12.0033
   15.9263


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

Price Vanilla Floor Instrument Using Hull-White Model and Hull-White Pricer

This example shows the workflow to price a vanilla `Floor` instrument when you use a `HullWhite` model and a `HullWhite` pricing method.

Create `Floor` Instrument Object

Use `fininstrument` to create a `Floor` instrument object.

```
FloorOpt = fininstrument("Floor",'Strike',0.039,'Maturity',datetime(2019,1,30),'Reset',4,'Princi
```

```
FloorOpt =
  Floor with properties:

                     Strike: 0.0390
                   Maturity: 30-Jan-2019
                ResetOffset: 0
                      Reset: 4
                      Basis: 12
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "floor_option"
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.032,'Sigma',0.04)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0320
    Sigma: 0.0400
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `HullWhite` Pricer Object

Use `finpricer` to create a `HullWhite` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',HullWhiteModel,'DiscountCurve',myRC)
```

```
outPricer =
  HullWhite with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.HullWhite]
```

Price `Floor` Instrument

Use `price` to compute the price for the `Floor` instrument.

```
Price = price(outPricer,FloorOpt)
```

```
Price = 1.2676
```

Price Amortizing Floor Instrument Using Black Model and Black Pricer

This example shows the workflow to price an amortizing `Floor` instrument when you use a `Black` model and a `Black` pricing method.

Create `Floor` Instrument Object

Use `fininstrument` to create an amortizing `Floor` instrument object.

```
CADates = datetime([2020,9,1 ; 2023,9,1]);
CAPrincipal = [100; 85];
Principal = timetable(CADates,CAPrincipal);
```

```
FloorOpt = fininstrument("Floor",'Maturity',datetime(2023,9,1),'Strike',0.015,'Principal',Princi
```

```
FloorOpt =
  Floor with properties:

                     Strike: 0.0150
                   Maturity: 01-Sep-2023
                ResetOffset: 0
                      Reset: 1
                      Basis: 0
                  Principal: [2x1 timetable]
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "floor_amortizing_option"
```

Create Black Model Object

Use `finmodel` to create a `Black` model object.

```
BlackModel = finmodel("Black",'Volatility',0.2)
```

```
BlackModel =
  Black with properties:

    Volatility: 0.2000
         Shift: 0
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,1);
Type = 'zero';
ZeroTimes = [calyears([1 2 3 4 5 7 10])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates);
```

Create Black Pricer Object

Use `finpricer` to create a `Black` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackModel,'DiscountCurve',myRC)
```

```
outPricer =
  Black with properties:

            Model: [1x1 finmodel.Black]
    DiscountCurve: [1x1 ratecurve]
```

Price Floor Instrument

Use price to compute the price for the Floor instrument.

```
Price = price(outPricer,FloorOpt)
```

```
Price = 3.0030
```

Price Vanilla Floor Instrument Using Hull-White Model and IRTree Pricer

This example shows the workflow to price a vanilla Floor instrument when using a HullWhite model and an IRTree pricing method.

Create Floor Instrument Object

Use fininstrument to create a Floor instrument object.

```
FloorOpt = fininstrument("Floor",'Strike',0.03,'Maturity',datetime(2020,1,30),'Reset',4,'Principa
```

```
FloorOpt =
  Floor with properties:

                    Strike: 0.0300
                  Maturity: 30-Jan-2020
               ResetOffset: 0
                     Reset: 4
                     Basis: 8
                 Principal: 100
            ProjectionCurve: [0x0 ratecurve]
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                  Holidays: NaT
                      Name: "floor_option"
```

Create HullWhite Model Object

Use finmodel to create a HullWhite model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.01,'Sigma',0.10)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0100
    Sigma: 0.1000
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
```

```
myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
CFdates = cfdates(Settle, FloorOpt.Maturity, FloorOpt.Reset, FloorOpt.Basis);
outPricer = finpricer("IRTree",'Model',HullWhiteModel,'DiscountCurve',myRC,'TreeDates',CFdates')
```

```
outPricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [6x1 datetime]
           Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

Price Floor Instrument

Use `price` to compute the price and sensitivities for the `Floor` instrument.

```
[Price, outPR] = price(outPricer,FloorOpt,["all"])
```

```
Price = 5.7821
```

```
outPR =
  priceresult with properties:

      Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma      Vega
    ______    _______    ______    ______

    5.7821    -110.54    141.45    31.821
```

Price Floor Instrument Using BraceGatarekMusiela Model and IRMonteCarlo Pricer

This example shows the workflow to price a `Floor` instrument when using a `BraceGatarekMusiela` model and an `IRMonteCarlo` pricing method.

Create Floor Instrument Object

Use `fininstrument` to create a `Floor` instrument object.

```
FloorOpt = fininstrument("Floor","Maturity",datetime(2022,9,15),'Strike',0.05,'Reset',1,'Name',"
```

```
FloorOpt =
  Floor with properties:

                    Strike: 0.0500
                  Maturity: 15-Sep-2022
               ResetOffset: 0
                     Reset: 1
                     Basis: 0
                 Principal: 100
            ProjectionCurve: [0x0 ratecurve]
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                  Holidays: NaT
                      Name: "floor_option"
```

Create BraceGatarekMusiela Model Object

Use `finmodel` to create a `BraceGatarekMusiela` model object.

```
BGMVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
BGMVolParams = [.3 -.02 .7 .14];
numRates = 20;
VolFunc(1:numRates-1) = {@(t) BGMVolFunc(BGMVolParams,t)};
Beta = .08;
CorrFunc = @(i,j,Beta) exp(-Beta*abs(i-j));
Correlation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),Beta);
BGM = finmodel("BraceGatarekMusiela",'Volatility',VolFunc,'Correlation',Correlation,'Period',1);
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 0
              Dates: [9x1 datetime]
```

```
               Rates: [9x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",'Model',BGM,'DiscountCurve',myRC,'SimulationDates',ZeroDates
```

```
outPricer =
  BGMMonteCarlo with properties:

         NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jan-2021    ...    ]
              Model: [1x1 finmodel.BraceGatarekMusiela]
```

Price Floor Instrument

Use `price` to compute the price and sensitivities for the `Floor` instrument.

```
[Price,outPR] = price(outPricer,FloorOpt,["all"])
```

```
Price = 14.7975

outPR =
  priceresult with properties:

       Results: [1x3 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×3 table
    Price      Delta     Gamma
    ______    _______    ______

    14.797    -398.43    1399.5
```

More About

Floor

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

The payoff for a floor is: max(*FloorRate* − *CurrentRate*, 0)

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Floor` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`Cap` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Work with Negative Interest Rates Using Objects” on page 2-22

FRA

FRA instrument object

Description

Create and price a FRA (forward rate agreement) instrument object for one or more FRA instruments using this workflow:

1. Use `fininstrument` to create a FRA instrument object for one or more FRA instruments.
2. Use `ratecurve` to specify an interest-rate model for the FRA instrument object.
3. Use `finpricer` to specify a `Discount` pricing method for one or more FRA instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a FRA instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
FRAObj = fininstrument(InstrumentType,'StartDate',start_date,'
Maturity',maturity_date,'Rate',rate_value)
FRAObj = fininstrument( ___ ,Name,Value)
```

Description

`FRAObj = fininstrument(InstrumentType,'StartDate',start_date,'Maturity',maturity_date,'Rate',rate_value)` creates a FRA object for one or more FRA instruments by specifying `InstrumentType` and sets the properties on page 11-2837 for the required name-value pair arguments `StartDate`, `Maturity`, and `Rate`. For more information on a FRA instrument, see “More About” on page 11-2842.

`FRAObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2837 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `FRAObj = fininstrument("FRA",'StartDate',datetime(2016,1,30),'Maturity',datetime(2019,1,30),'Rate',0.025,'Principal',100,'Basis',1,'BusinessDayConvention',"follow",'Name',"FRA_instrument")` creates a FRA instrument with a principal of 100 and a maturity of January 30, 2019. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"FRA"` | string array with values of `"FRA"` | character vector with value `'FRA'` | cell array of character vectors with values of `'FRA'`

Instrument type, specified as a string with the value of "FRA", a character vector with the value of 'FRA', an NINST-by-1 string array with values of "FRA", or an NINST-by-1 cell array of character vectors with values of 'FRA'.

Data Types: char | cell | string

FRA Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: FRAObj = fininstrument("FRA",'StartDate',datetime(2016,1,30),'Maturity',datetime(2019,1,30),'Rate',0.025,'Principal',100,'Basis',1,'BusinessDayConvention',"follow",'Name',"FRA_instrument")

Required FRA Name-Value Pair Arguments

StartDate — FRA start date

datetime array | string array | date character vector

FRA start date, specified as the comma-separated pair consisting of 'StartDate' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, FRA also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by datetime because the StartDate property is stored as a datetime.

Maturity — FRA maturity date

datetime array | string array | date character vector

FRA maturity date, specified as the comma-separated pair consisting of 'Maturity' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, FRA also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by datetime because the Maturity property is stored as a datetime.

Rate — FRA coupon rate

scalar decimal | vector of decimals

FRA coupon rate, specified as the comma-separated pair consisting of 'Rate' and a scalar decimal or an NINST-by-1 vector of decimals.

Data Types: double

Optional FRA Name-Value Pair Arguments

Basis — Day count basis

0 (actual/actual) (default) | scalar integer from 0 to 13 | vector of integers from 0 to 13

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and scalar value or an `NINST`-by-`1` vector of integers from the following:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal amount
`100` (default) | scalar numeric | numeric vector

Principal amount, specified as the comma-separated pair consisting of `'Principal'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`BusinessDayConvention` — Business day convention
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day convention, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.

- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and dates using an `NINST`-by-`1` vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
FRAObj = fininstrument("FRA",'StartDate',datetime(2016,1,30),'Maturity',datetime(2025,12,15),'Ra
```

To support existing code, `FRA` also accepts serial date numbers as inputs, but they are not recommended.

`Name` — User-defined name for instrument
`"  "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`StartDate` — FRA start date
scalar datetime | vector of datetimes

FRA start date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Maturity` — FRA maturity date
scalar datetime | vector of datetimes

FRA maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Rate` — FRA coupon rate
scalar decimal | vector of decimals

FRA coupon rate, returned as a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Basis` — Day count basis
`[0 0]` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Principal` — Principal amount
`100` (default) | scalar numeric | numeric vector

Principal amount, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`BusinessDayConvention` — Business day convention
`"actual"` (default) | scalar string | string array

Business day convention, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Name` — User-defined name for instrument
`" "` (default) | scalar string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

cashflows	Compute cash flow for FixedBond, FloatBond, Swap, FRA, STIRFuture, OISFuture, OvernightIndexedSwap, or Deposit instrument

Examples

Price FRA Instrument Using `ratecurve` and Discount Pricer

This example shows the workflow to price a FRA (forward rate agreement) instrument when you use a `ratecurve` and a `Discount` pricing method.

Create FRA Instrument Object

Use `fininstrument` to create a FRA instrument object.

```
FRAObj = fininstrument("FRA",'StartDate',datetime(2020,9,15),'Maturity',datetime(2022,9,15),'Rate
```

```
FRAObj =
  FRA with properties:

                     Rate: 0.0175
                    Basis: 2
                StartDate: 15-Sep-2020
                 Maturity: 15-Sep-2022
                Principal: 100
    BusinessDayConvention: "actual"
                 Holidays: NaT
```

```
                    Name: ""
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 0
              Dates: [10x1 datetime]
              Rates: [10x1 double]
             Settle: 15-Sep-2018
       InterpMethod: "linear"
  ShortExtrapMethod: "next"
   LongExtrapMethod: "previous"
```

Create Discount Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price FRA Instrument

Use `price` to compute the price and sensitivities for the FRA instrument.

```
[Price, outPR] = price(outPricer, FRAObj,["all"])
```

```
Price = 3.4176
```

```
outPR =
  priceresult with properties:

      Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    ________

    3.4176    0.001368
```

Price Multiple FRA Instruments Using ratecurve and Discount Pricer

This example shows the workflow to price multiple FRA (forward rate agreement) instruments when you use a `ratecurve` and a `Discount` pricing method.

Create FRA Instrument Object

Use `fininstrument` to create a FRA instrument object for three FRA instruments.

```
FRAObj = fininstrument("FRA",'StartDate',datetime([2020,9,15 ; 2020,10,15 ; 2020,11,15]),'Maturi
```

```
FRAObj=3×1 object
  3x1 FRA array with properties:

    Rate
    Basis
    StartDate
    Maturity
    Principal
    BusinessDayConvention
    Holidays
    Name
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount",'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price FRA Instruments

Use `price` to compute the prices and sensitivities for the FRA instruments.

```
[Price, outPR] = price(outPricer, FRAObj,["all"])
```

```
Price = 3×1

    3.4176
    3.4121
    3.4063


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    ________

    3.4176    0.001368


ans=1×2 table
    Price       DV01
    ______    _________

    3.4121    0.0013938


ans=1×2 table
    Price       DV01
    ______    _________
```

```
    3.4063    0.0014204
```

More About

FRA Instrument

A FRA (forward rate agreement) instrument is an over-the-counter contract between parties that determines the rate of interest to be paid on an agreed upon date in the future.

The FRA determines the rates to be used along with the termination date and notional value. FRAs are cash-settled with the payment based on the net difference between the interest rate of the contract and the floating rate in the market, called the reference rate. The notional amount is not exchanged, but is rather a cash amount based on the rate differentials and the notional value of the contract.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although FRA supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Lookback

`Lookback` instrument

Description

Create and price a `Lookback` instrument object for one or more Lookback instruments using this workflow:

1. Use `fininstrument` to create a `Lookback` instrument object for one or more Lookback instruments.
2. Use `finmodel` to specify a `BlackScholes`, `Heston`, `Bates`, or `Merton` model for the `Lookback` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify a `ConzeViswanathan`, `AssetTree`, or `GoldmanSosinGatto` pricing method for one or more `Lookback` instruments.
   - When using a `BlackScholes`, `Heston`, `Bates`, or `Merton` model, use `finpricer` to specify an `AssetMonteCarlo` pricing method for one or more `Lookback` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Lookback` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
LookbackObj = fininstrument(InstrumentType,'Strike',strike_value,'
ExerciseDate',exercise_date)
LookbackObj = fininstrument( ___ ,Name,Value)
```

Description

`LookbackObj = fininstrument(InstrumentType,'Strike',strike_value,'ExerciseDate',exercise_date)` creates a `Lookback` object for one or more Lookback instruments by specifying `InstrumentType` and sets the properties on page 11-2845 for the required name-value pair arguments `Strike` and `ExerciseDate`.

The `Lookback` instrument supports fixed-strike and floating-strike lookback options. For more information on a `Lookback` instrument, see “More About” on page 11-2858.

`LookbackObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2845 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `LookbackObj =`

`fininstrument("Lookback",'Strike',100,'ExerciseDate',datetime(2019,1,30),'OptionType',"put",'ExerciseStyle',"European",'Name',"lookback_option")` creates a `Lookback` put option with an European exercise. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Lookback"` | string array with values of `"Lookback"` | character vector with value `'Lookback'` | cell array of character vectors with values of `'Lookback'`

Instrument type, specified as a string with the value of `"Lookback"`, a character vector with the value of `'Lookback'`, an `NINST`-by-`1` string array with values `"Lookback"`, or an `NINST`-by-`1` cell array of character vectors with values of `'Lookback'`.

Data Types: `char` | `cell` | `string`

Lookback Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `LookbackObj = fininstrument("Lookback",'Strike',100,'ExerciseDate',datetime(2019,1,30),'OptionType',"put",'ExerciseStyle',"European",'Name',"lookback_option")`

Required Lookback Name-Value Pair Arguments

`Strike` — Option strike price value

nonnegative numeric | vector of nonnegative values | `NaN`

Option strike price value, specified as the comma-separated pair consisting of `'Strike'` and a scalar nonnegative numeric value or an `NINST`-by-`1` vector of nonnegative values for a fixed-strike `Lookback` option. For a floating-strike `Lookback` option, specify `'Strike'` as a `NaN` or an `NINST`-by-`1` vector of `NaN`s.

Note Use the `ConzeViswanathan` pricer for a fixed strike `Lookback` option and use the `GoldmanSosinGatto` pricer for a floating strike `Lookback` option.

Data Types: `double`

`ExerciseDate` — Option exercise date

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of `'ExerciseDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one `ExerciseDate` on the option expiry date.

To support existing code, `Lookback` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `ExerciseDate` property is stored as a datetime.

Optional Lookback Name-Value Pair Arguments

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values `'call'` or `'put'`

Option type, specified as the comma-separated pair consisting of `'OptionType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` or `"American"` | string array with values `"European"` or `"American"` | character vector with value `'European'` or `'American'` | cell array of character vectors with values `'European'` or `'American'`

Option exercise style, specified as the comma-separated pair consisting of `'ExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `string` | `cell` | `char`

`AssetMinMax` — Maximum or minimum underlying asset price
`NaN` where `SpotPrice` of the underlying asset is used (default) | scalar numeric | numeric vector

Maximum or minimum underlying asset price, specified as the comma-separated pair consisting of `'AssetMinMax'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Name` — User-defined name for instrument
`"  "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Strike` — Option strike price value
nonnegative numeric | vector of nonnegative values

Option strike price value, returned as a scalar nonnegative numeric or an `NINST`-by-`1` vector of nonnegative values.

Data Types: `double`

`ExerciseDate` — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a datetime or an NINST-by-1 vector of datetimes.

Data Types: datetime

OptionType — Option type
"call" (default) | string with value "call" or "put" | string array with values "call" or "put"

Option type, returned as a scalar string or an NINST-by-1 string array with the values of "call" or "put".

Data Types: string

ExerciseStyle — Option exercise style
"European" (default) | string with value "European" or "American" | string array with values "European" or "American"

Option exercise style, returned as a scalar string or an NINST-by-1 string array with values of "European" or "American".

Data Types: string

AssetMinMax — Maximum or minimum underlying asset price
NaN where SpotPrice of the underlying asset is used (default) | scalar numeric | numeric vector

Maximum or minimum underlying asset price, returned as a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

Name — User-defined name for instrument
" " (default) | string | string array

User-defined name for the instrument, returned as a string or an NINST-by-1 string array.

Data Types: string

Examples

Price Lookback Instrument Using a Black-Scholes Model and Conze-Viswanathan Pricer

This example shows the workflow to price a LookBack instrument when you use a BlackScholes model and a ConzeViswanathan pricing method.

Create Lookback Instrument Object

Use fininstrument to create a Lookback instrument object.

```
LookbackOpt = fininstrument("Lookback",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionTy
```

```
LookbackOpt =
  Lookback with properties:

       OptionType: "put"
           Strike: 105
      AssetMinMax: NaN
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
```

```
             Name: "lookback_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create ConzeViswanathan Pricer Object

Use `finpricer` to create a `ConzeViswanathan` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic","Model",BlackScholesModel,"DiscountCurve",myRC,"SpotPrice",100,'
```

```
outPricer =
  ConzeViswanathan with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.2500
     DividendType: "continuous"
```

Price Lookback Instrument

Use `price` to compute the price and sensitivities for the `Lookback` instrument.

```
[Price, outPR] = price(outPricer,LookbackOpt,["all"])

Price = 57.8786

outPR =
  priceresult with properties:

      Results: [1x7 table]
    PricerData: []


outPR.Results

ans=1×7 table
    Price      Delta      Gamma      Lambda      Vega      Theta       Rho
    _____    ________    _____    ________    ______    _______    _______

    57.879    -0.33404      0      -0.57714    32.587    -5.1863    -350.41
```

Price Multiple Lookback Instruments Using a Black-Scholes Model and Conze-Viswanathan Pricer

This example shows the workflow to price multiple `LookBack` instrument when you use a `BlackScholes` model and a `ConzeViswanathan` pricing method.

Create Lookback Instrument Object

Use `fininstrument` to create a `Lookback` instrument object for three Lookback instruments.

```
LookbackOpt = fininstrument("Lookback",'Strike',[105 ; 120; 140],'ExerciseDate',datetime([2022,9

LookbackOpt=3×1 object
  3x1 Lookback array with properties:

    OptionType
    Strike
    AssetMinMax
    ExerciseStyle
    ExerciseDate
    Name
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.2)

BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create ratecurve Object

Create a flat ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create ConzeViswanathan Pricer Object

Use finpricer to create a ConzeViswanathan pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("analytic","Model",BlackScholesModel,"DiscountCurve",myRC,"SpotPrice",100,'
```

```
outPricer =
  ConzeViswanathan with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.2500
     DividendType: "continuous"
```

Price Lookback Instruments

Use price to compute the prices and sensitivities for the Lookback instruments.

```
[Price, outPR] = price(outPricer,LookbackOpt,["all"])
```

```
Price = 3×1

   57.8786
   71.3008
   88.9673


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Lambda      Vega      Theta       Rho
    _____    ________    _____    ________    ______    _______    _______

    57.879    -0.33404      0       -0.57714    32.587    -5.1863    -350.41


ans=1×7 table
    Price      Delta        Gamma         Lambda       Vega      Theta       Rho
    _____    ________    __________    ________    ______    _______    _______

    71.301    -0.32722    2.8422e-06    -0.45894    31.997    -4.5677    -410.15


ans=1×7 table
    Price      Delta        Gamma         Lambda       Vega      Theta       Rho
    _____    ________    __________    ________    ______    _______    _______

    88.967    -0.32033    1.4211e-06    -0.36005    31.395    -3.7989    -489.96
```

Price Lookback Instrument Using a Black-Scholes Model and Asset Tree Pricer for LR Binomial Tree

This example shows the workflow to price a `LookBack` instrument when you use an `BlackScholes` model and an `AssetTree` pricing method using a Leisen-Reimer (LR) binomial tree.

Create Lookback Instrument Object

Use `fininstrument` to create a `Lookback` instrument object.

```
LookbackOpt = fininstrument("Lookback",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionTy
```

```
LookbackOpt =
  Lookback with properties:

       OptionType: "put"
           Strike: 105
      AssetMinMax: NaN
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
             Name: "lookback_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
```

```
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object for a LR equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
NumPeriods = 15;
LRPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',150,
```

```
LRPricer =
  LRTree with properties:

    InversionMethod: PP1
             Strike: 150
               Tree: [1x1 struct]
        NumPeriods: 15
              Model: [1x1 finmodel.BlackScholes]
      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 150
       DividendType: "continuous"
      DividendValue: 0
          TreeDates: [21-Dec-2018 09:36:00    28-Mar-2019 19:12:00    ...    ]
```

```
LRPricer.Tree
```

```
ans = struct with fields:
    Probs: [2x15 double]
    ATree: {1x16 cell}
     dObs: [15-Sep-2018 00:00:00    21-Dec-2018 09:36:00    ...    ]
     tObs: [0 0.2667 0.5333 0.8000 1.0667 1.3333 1.6000 1.8667 2.1333 ... ]
```

Price Lookback Instrument

Use `price` to compute the price and sensitivities for the `Lookback` instrument.

```
[Price, outPR] = price(LRPricer,LookbackOpt,["all"])
```

```
Price = 3.9412
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta        Gamma        Vega     Lambda       Rho        Theta
    _____    ________    _________    _____    _______    _______    _______

    3.9412    -0.13312    -0.011131    67.684    -5.0757    -73.857    -1.0383
```

Price Lookback Instrument Using a Black-Scholes Model and Asset Tree Pricer for Standard Trinomial Tree

This example shows the workflow to price a `LookBack` instrument when you use an `BlackScholes` model and an `AssetTree` pricing method using a Standard Trinomial (STT) tree.

Create Lookback Instrument Object

Use `fininstrument` to create a `Lookback` instrument object.

```
LookbackOpt = fininstrument("Lookback",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionTy
```

```
LookbackOpt =
  Lookback with properties:

      OptionType: "put"
          Strike: 105
     AssetMinMax: NaN
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
             Name: "lookback_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

    Volatility: 0.2000
```

```
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object for a Standard Trinomial (STT) equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
NumPeriods = 15;
STTPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',150
```

```
STTPricer =
  STTree with properties:

            Tree: [1x1 struct]
       NumPeriods: 15
            Model: [1x1 finmodel.BlackScholes]
    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 150
     DividendType: "continuous"
    DividendValue: 0
        TreeDates: [21-Dec-2018 09:36:00    28-Mar-2019 19:12:00    ...    ]
```

```
STTPricer.Tree
```

```
ans = struct with fields:
    ATree: {1x16 cell}
    Probs: {1x15 cell}
     dObs: [15-Sep-2018 00:00:00    21-Dec-2018 09:36:00    ...    ]
     tObs: [0 0.2667 0.5333 0.8000 1.0667 1.3333 1.6000 1.8667 2.1333 ... ]
```

Price Lookback Instrument

Use `price` to compute the price and sensitivities for the `Lookback` instrument.

```
[Price, outPR] = price(STTPricer,LookbackOpt,["all"])

Price = 3.3392

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []


outPR.Results

ans=1×7 table
    Price      Delta         Gamma         Vega      Lambda       Rho        Theta
    _____    _________    ___________    ______    _______    ________    _______

    3.3392    -0.15942    -1.0596e-11    63.886    -7.1613    -68.263    -1.0254
```

Price Lookback Instrument Using a Black-Scholes Model and Asset Monte-Carlo Pricer

This example shows the workflow to price a `LookBack` instrument when you use a `BlackScholes` model and an `AssetMonetCarlo` pricing method.

Create Lookback Instrument Object

Use `fininstrument` to create a `Lookback` instrument object.

```
LookbackOpt = fininstrument("Lookback",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionTy

LookbackOpt =
  Lookback with properties:

       OptionType: "put"
           Strike: 105
      AssetMinMax: NaN
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
             Name: "lookback_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.2)

BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BlackScholesModel,'SpotPrice
```

```
outPricer =
  GBMMonteCarlo with properties:

     DiscountCurve: [1x1 ratecurve]
         SpotPrice: 200
    SimulationDates: 15-Sep-2022
         NumTrials: 1000
     RandomNumbers: []
             Model: [1x1 finmodel.BlackScholes]
      DividendType: "continuous"
     DividendValue: 0
```

Price Lookback Instrument

Use `price` to compute the price and sensitivities for the `Lookback` instrument.

```
[Price, outPR] = price(outPricer,LookbackOpt,["all"])
```

```
Price = 1.8553

outPR =
  priceresult with properties:

      Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price       Delta        Gamma       Lambda       Rho       Theta       Vega
    ______    _________    __________    _______    _______    ________    ______
```

```
    1.8553    -0.040442    0.00062792    -4.3596    -39.426    -0.71345    42.311
```

Price Lookback Instrument Using a Bates Model and Asset Monte-Carlo Pricer

This example shows the workflow to price a `LookBack` instrument when you use a `Bates` model and an `AssetMonetCarlo` pricing method.

Create Lookback Instrument Object

Use `fininstrument` to create a `Lookback` instrument object.

```
LookbackOpt = fininstrument("Lookback",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionTy
```

```
LookbackOpt =
  Lookback with properties:

       OptionType: "put"
           Strike: 105
      AssetMinMax: NaN
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
             Name: "lookback_option"
```

Create Bates Model Object

Use `finmodel` to create a `Bates` model object.

```
BatesModel = finmodel("Bates",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9,'Mea
```

```
BatesModel =
  Bates with properties:

          V0: 0.0320
      ThetaV: 0.1000
       Kappa: 0.0030
      SigmaV: 0.2000
       RhoSV: 0.9000
       MeanJ: 0.1100
     JumpVol: 0.0230
    JumpFreq: 0.0200
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:
```

```
                Type: "zero"
        Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BatesModel,'SpotPrice',100,
```

```
outPricer =
  BatesMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 100
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Bates]
       DividendType: "continuous"
      DividendValue: 0
```

Price Lookback Instrument

Use `price` to compute the price and sensitivities for the `Lookback` instrument.

```
[Price, outPR] = price(outPricer,LookbackOpt,["all"])
```

```
Price = 7.2577
```

```
outPR =
  priceresult with properties:

       Results: [1x8 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×8 table
    Price      Delta      Gamma     Lambda       Rho        Theta       Vega      VegaLT
    ______    _______    _____    _______    _______    _________    ______    _______
```

```
    7.2577    -0.84025      0      -11.577    -29.025    -0.027666    30.748    0.68416
```

More About

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 Instruments 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. For more information, see “Lookback Option” on page 3-39.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Lookback` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`PartialLookback` | `finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

OptionEmbeddedFixedBond

`OptionEmbeddedFixedBond` instrument object

Description

Create and price a `OptionEmbeddedFixedBond` instrument object for one or more Option Embedded Fixed Bond instruments using this workflow:

1. Use `fininstrument` to create an `OptionEmbeddedFixedBond` instrument object for one or more Option Embedded Fixed Bond instruments.
2. use `finmodel` to specify a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model for the `OptionEmbeddedFixedBond` instrument object.
3. Choose a pricing method.
   - When using a `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` model, use `finpricer` to specify an `IRTree` pricing method for one or more `OptionEmbeddedFixedBond` instruments.
   - When using a `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model, use `finpricer` to specify an `IRMonteCarlo` pricing method for one or more `OptionEmbeddedFixedBond` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for an `OptionEmbeddedFixedBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
OptionEmbeddedFixedBondObj = fininstrument(InstrumentType,'
CouponRate',couponrate_value,'Maturity',maturity_date,'
CallSchedule',call_schedule_value)
OptionEmbeddedFixedBondObj = fininstrument(InstrumentType,'
CouponRate',couponrate_value,'Maturity',maturity_date,'
PutSchedule',put_schedule_value)
OptionEmbeddedFixedBondObj = fininstrument( ___ ,Name,Value)
```

Description

`OptionEmbeddedFixedBondObj = fininstrument(InstrumentType,'CouponRate',couponrate_value,'Maturity',maturity_date,'CallSchedule',call_schedule_value)` creates a `OptionEmbeddedFixedBond` object for one

or more Option Embedded Fixed Bond instruments by specifying `InstrumentType` and sets the properties on page 11-2866 for the required name-value pair arguments `CouponRate`, `Maturity`, and `CallSchedule`.

The `OptionEmbeddedFixedBond` instrument supports a vanilla bond with embedded option, stepped coupon bond with embedded option, and an amortizing bond with embedded option. For more information, see "More About" on page 11-2881.

`OptionEmbeddedFixedBondObj = fininstrument(InstrumentType,'CouponRate',couponrate_value,'Maturity',maturity_date,'PutSchedule',put_schedule_value)` creates a `OptionEmbeddedFixedBond` object for one or more Option Embedded Fixed Bond instruments by specifying `InstrumentType` and sets the properties on page 11-2866 for the required name-value pair arguments `CouponRate`, `Maturity`, and `PutSchedule`.

`OptionEmbeddedFixedBondObj = fininstrument(___,Name,Value)` sets optional properties on page 11-2866 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `OptionEmbeddedFixedBondObj = fininstrument("OptionEmbeddedFixedBond",'CouponRate',0.034,'Maturity',datetime(2019,1,30),'Period',2,'Basis',1,'Principal',100,'CallSchedule',schedule,'CallExerciseStyle',"American",'Name',"optionembeddedfixedbond_instrument")` creates an `OptionEmbeddedFixedBond` instrument with an American exercise and a call schedule. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"OptionEmbeddedFixedBond"` | string array with values of `"OptionEmbeddedFixedBond"` | character vector with value `'OptionEmbeddedFixedBond'` | cell array of character vectors with values of `'OptionEmbeddedFixedBond'`

Instrument type, specified as a string with the value of `"OptionEmbeddedFixedBond"`, a character vector with the value of `'OptionEmbeddedFixedBond'`, an `NINST`-by-`1` string array with values of `"OptionEmbeddedFixedBond"`, or an `NINST`-by-`1` cell array of character vectors with values of `'OptionEmbeddedFixedBond'`.

Data Types: `char` | `cell` | `string`

`OptionEmbeddedFixedBond` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `OptionEmbeddedFixedBondObj = fininstrument("OptionEmbeddedFixedBond",'CouponRate',0.034,'Maturity',datetime(2019,1,30),'Period',2,'Basis',1,'Principal',100,'CallSchedule',schedule,'CallExerciseStyle',"American",'Name',"optionembeddedfixedbond_instrument")`

Required `OptionEmbeddedFixedBond` Name-Value Pair Arguments

`CouponRate` — Coupon rate for `OptionEmbeddedFixedBond`

scalar decimal | vector of decimals | timetable

Coupon rate for `OptionEmbeddedFixedBond`, specified as the comma-separated pair consisting of `'CouponRate'` as a scalar decimal or an `NINST`-by-`1` vector of decimals for an annual rate or a timetable where the first column is dates and the second column is associated rates. The date indicates the last day that the coupon rate is valid.

Note If you are creating one or more `OptionEmbeddedFixedBond` instruments and use a timetable, the timetable specification applies to all of the `OptionEmbeddedFixedBond` instruments. `CouponRate` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `double` | `timetable`

`Maturity` — Maturity date for `OptionEmbeddedFixedBond`
datetime array | string array | date character vector

Maturity date for `OptionEmbeddedFixedBond`, specified as the comma-separated pair consisting of `'Maturity'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OptionEmbeddedFixedBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`CallSchedule` — Call schedule
timetable

Call schedule, specified as the comma-separated pair consisting of `'CallSchedule'` and a timetable of call dates and strikes.

If you use a date character vector or date string for the dates in this timetable, the format must be recognizable by `datetime` because the `CallSchedule` property is stored as a datetime.

Note The `OptionEmbeddedFixedBond` instrument supports either `CallSchedule` and `CallExerciseStyle` or `PutSchedule` and `PutExerciseStyle`, but not both.

If you are creating one or more `OptionEmbeddedFixedBond` instruments and use a timetable, the timetable specification applies to all of the `OptionEmbeddedFixedBond` instruments. `CallSchedule` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `timetable`

`PutSchedule` — Call schedule
timetable

Put schedule, specified as the comma-separated pair consisting of `'PutSchedule'` and a timetable of call dates and strikes.

If you use a date character vector or date string for dates in this timetable, the format must be recognizable by `datetime` because the `PutSchedule` property is stored as a datetime.

Note The `OptionEmbeddedFixedBond` instrument supports either `CallSchedule` and `CallExerciseStyle` or `PutSchedule` and `PutExerciseStyle`, but not both.

If you are creating one or more `OptionEmbeddedFixedBond` instruments and use a timetable, the timetable specification applies to all of the `OptionEmbeddedFixedBond` instruments. `PutSchedule` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `timetable`

Optional OptionEmbeddedFixedBond Name-Value Pair Arguments

`Period` — Frequency of payments per year

2 (default) | scalar integer | vector of integers

Frequency of payments per year, specified as the comma-separated pair consisting of `'Period'` and a scalar integer or an `NINST`-by-`1` vector of integers. Values for `Period` are: `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`CallExerciseStyle` — Call option exercise style

`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with value `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'` | cell array of character vectors with values of `'European'`, `'American'`, or `'Bermudan'`

Call option exercise style, specified as the comma-separated pair consisting of `'CallExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Note The `CallSchedule` is a timetable of call dates and strikes. If you do not specify a `CallExerciseStyle`, then based on the `CallSchedule` specification, a default value of `CallExerciseStyle` is assigned as follows:

- If there is one exercise date in the `CallSchedule`, then the `CallExerciseStyle` is an `"European"`.
- If there are two exercise dates in the `CallSchedule`, then the `CallExerciseStyle` is an `"American"` with a start date and maturity.
- If there are more than two exercise dates in the `CallSchedule`, then the `CallExerciseStyle` is an `"Bermudan"`.

If the you define a `CallExerciseStyle` and this is not consistent with what you have specified in the `CallSchedule`, you receive an error message.

Data Types: `string` | `cell` | `char`

`PutExerciseStyle` — Put option exercise style

`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with value `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'` | cell array of character vectors with values of `'European'`, `'American'`, or `'Bermudan'`

Put option exercise style, specified as the comma-separated pair consisting of `'PutExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Note The `PutSchedule` is a timetable of call dates and strikes. If you do not specify a `PutExerciseStyle`, then based on the `PutSchedule` specification, a default value of `PutExerciseStyle` is assigned as follows:

- If there is one exercise date in the `PutSchedule`, then the `PutExerciseStyle` is an `"European"`.
- If there are two exercise dates in the `PutSchedule`, then the `PutExerciseStyle` is an `"American"` with a start date and maturity.
- If there are more than two exercise dates in the `PutSchedule`, then the `PutExerciseStyle` is an `"Bermudan"`.

If the you define a `PutExerciseStyle` and this is not consistent with what you have specified in the `PutSchedule`, you receive an error message.

Data Types: `string` | `cell` | `char`

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and scalar integer or an `NINST`-by-`1` vector of integers using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Notional principal amount or principal value schedule, specified as the comma-separated pair consisting of `'Principal'` and a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

`Principal` accepts a `timetable`, where the first column is dates and the second column is the associated notional principal value. The date indicates the last day that the principal value is valid.

Note If you are creating one or more `OptionEmbeddedFixedBond` instruments and use a timetable, the timetable specification applies to all of the `OptionEmbeddedFixedBond` instruments. `Principal` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `double` | `timetable`

`DaycountAdjustedCashFlow` — Flag indicating whether cash flow adjusts for day count convention
`false` (default) | scalar logical value of `true` or `false` | vector of logicals with values of `true` or `false`

Flag indicating whether cash flow adjusts for day count convention, specified as the comma-separated pair consisting of `'DaycountAdjustedCashFlow'` and a scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and dates using an `NINST`-by-`1` vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
OptionEmbeddedFixedBondObj = fininstrument("OptionEmbeddedFixedBond",'CouponRate',0.34,'Maturity
'CallSchedule',schedule,'CallExerciseStyle',"american",'Holidays',H)
```

To support existing code, `OptionEmbeddedFixedBond` also accepts serial date numbers as inputs, but they are not recommended.

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days

`true` (in effect) (default) | scalar logical value of `true` or `false` | vector of logicals with values of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

- If you set `EndMonthRule` to `false`, the software ignores the rule, meaning that a payment date is always the same numerical day of the month.
- If you set `EndMonthRule` to `true`, the software sets the rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date

`NaT` (default) | datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OptionEmbeddedFixedBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `IssueDate` property is stored as a datetime.

`FirstCouponDate` — Irregular first coupon date

`NaT` (default) | datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OptionEmbeddedFixedBond` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or date strings, the format must be recognizable by `datetime` because the `FirstCouponDate` property is stored as a datetime.

`LastCouponDate` — Irregular last coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OptionEmbeddedFixedBond` also accepts serial date numbers as inputs, but they are not recommended.

If you specify `LastCouponDate` but not `FirstCouponDate`, `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify `LastCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `LastCouponDate` property is stored as a datetime.

`StartDate` — Forward starting date of payments
`NaT` (default) | datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OptionEmbeddedFixedBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `StartDate` property is stored as a datetime.

`Name` — User-defined name for instrument
`"  "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for the instrument, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`CouponRate` — Coupon annual rate
scalar decimal | timetable | vector of decimals

Coupon annual rate, returned as a scalar decimal or an `NINST`-by-`1` vector of decimals or a timetable.

Data Types: `double` | `timetable`

`Maturity` — Maturity date
scalar datetime | vector of datetimes

Maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`CallSchedule` — Call schedule
timetable

Call schedule, returned as a timetable.

Data Types: `datetime`

`PutSchedule` — Call schedule
timetable

Put schedule, returned as a timetable.

Data Types: `datetime`

`Period` — Coupons per year
`2` (default) | scalar integer | vector of integers

Coupons per year, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Principal` — Notional principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Notional principal amount or principal value schedule, returned as a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

Data Types: `timetable` | `double`

`DaycountAdjustedCashFlow` — Flag indicating whether cash flow adjusts for day count convention
`false` (default) | scalar logical value of `true` or `false` | vector of logical values of `true` or `false`

Flag indicating whether cash flow adjusted for day count convention, returned as scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array

Business day conventions, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

EndMonthRule — End-of-month rule flag for generating dates when Maturity is end-of-month date for month with 30 or fewer days
`true` (in effect) (default) | scalar logical value of `true` or `false` | vector of logical values of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, returned as a scalar logical or an `NINST`-by-`1` vector of logicals.

Data Types: `logical`

IssueDate — Bond issue date
`NaT` (default) | scalar datetime | vector of datetimes

Bond issue date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

FirstCouponDate — Irregular first coupon date
`NaT` (default) | datetime | vector of datetimes

Irregular first coupon date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

LastCouponDate — Irregular last coupon date
`NaT` (default) | scalar datetime | vector of datetimes

Irregular last coupon date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

StartDate — Forward starting date of payments
`NaT` (default) | scalar datetime | vector of datetimes

Forward starting date of payments, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

CallExerciseStyle — Call option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermuda"` | string array with values of `"European"`, `"American"`, or `"Bermuda"`

This property is read-only.

Call option exercise style, returned as a scalar string or an `NINST`-by-`1` string array with values of `"European"`, `"American"`, or `"Bermuda"`.

Data Types: `string`

PutExerciseStyle — Put option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermuda"` | string array with values of `"European"`, `"American"`, or `"Bermuda"`

This property is read-only.

Put option exercise style, returned as a scalar string or an `NINST`-by-`1` string array with values of `"European"`, `"American"`, or `"Bermuda"`.

Data Types: `string`

Name — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

setCallExercisePolicy	Set call exercise policy for OptionEmbeddedFixedBond, OptionEmbeddedFloatBond, or ConvertibleBond instrument
setPutExercisePolicy	Set put exercise policy for OptionEmbeddedFixedBond, OptionEmbeddedFloatBond, or ConvertibleBond instrument

Examples

Price Option Embedded Fixed Bond Instruments Using Hull-White Model and IRTree Pricer

This example shows the workflow to price American, European, and Bermudan exercise styles for three callable `OptionEmbeddedFixedBond` instruments when you use a `HullWhite` model and an `IRTree` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding);
```

Create `OptionEmbeddedFixedBond` Instrument Objects

Use `fininstrument` to create three `OptionEmbeddedFixedBond` instrument objects with the different exercise styles.

```
Maturity = datetime(2024,1,1);

% Option embedded bond (Bermudan callable bond)
Strike = [100; 100];
ExerciseDates = [datetime(2020,1,1); datetime(2024,1,1)];
Period = 1;
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});

CallableBondBermudan = fininstrument("OptionEmbeddedFixedBond",'Maturity',Maturity,...
                                'CouponRate',0.025,'Period',Period, ...
                                'CallSchedule',CallSchedule,'CallExerciseStyle', "bermudan")
```

```
CallableBondBermudan =
  OptionEmbeddedFixedBond with properties:

                  CouponRate: 0.0250
```

```
                    Period: 1
                     Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2024
                   CallDates: [2x1 datetime]
                    PutDates: [0x1 datetime]
                CallSchedule: [2x1 timetable]
                 PutSchedule: [0x0 timetable]
           CallExerciseStyle: "bermudan"
            PutExerciseStyle: [0x0 string]
                        Name: ""
```

```
% Option embedded bond (American callable bond)
Strike = 100;
ExerciseDates = datetime(2024,1,1);
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
Period = 1;

CallableBondAmerican = fininstrument("OptionEmbeddedFixedBond",'Maturity',Maturity,...
                              'CouponRate',0.025,'Period', Period, ...
                              'CallSchedule',CallSchedule,'CallExerciseStyle',"american")
```

```
CallableBondAmerican =
  OptionEmbeddedFixedBond with properties:

                CouponRate: 0.0250
                    Period: 1
                     Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2024
                   CallDates: 01-Jan-2024
                    PutDates: [0x1 datetime]
                CallSchedule: [1x1 timetable]
                 PutSchedule: [0x0 timetable]
           CallExerciseStyle: "american"
            PutExerciseStyle: [0x0 string]
                        Name: ""
```

```
% Option embedded bond (European callable bond)
Strike = 100;
ExerciseDates = datetime(2024,1,1);
```

```
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
Period = 1;

CallableBondEuropean = fininstrument("OptionEmbeddedFixedBond",'Maturity',Maturity,...
                                'CouponRate',0.025,'Period',Period, ...
                                'CallSchedule',CallSchedule)
```

```
CallableBondEuropean =
  OptionEmbeddedFixedBond with properties:

                 CouponRate: 0.0250
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2024
                   CallDates: 01-Jan-2024
                    PutDates: [0x1 datetime]
                CallSchedule: [1x1 timetable]
                 PutSchedule: [0x0 timetable]
           CallExerciseStyle: "european"
            PutExerciseStyle: [0x0 string]
                        Name: ""
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;

HWModel = finmodel("HullWhite",'alpha',AlphaCurve,'sigma',VolCurve);
```

Create `IRTree` Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [10x1 datetime]
           Model: [1x1 finmodel.HullWhite]
   DiscountCurve: [1x1 ratecurve]
```

Price OptionEmbeddedFixedBond Instruments

Use `price` to compute the price and sensitivities for the three `OptionEmbeddedFixedBond` instruments.

```
[Price, outPR] = price(HWTreePricer,CallableBondBermudan,["all"])
```

```
Price = 103.2729

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma      Vega
    _____    _______    _____    _______

    103.27    -290.33    1375.9    -148.28
```

```
[Price, outPR] = price(HWTreePricer,CallableBondAmerican,["all"])
```

```
Price = 100

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price    Delta    Gamma    Vega
    _____    _____    _____    ____

     100       0        0       0
```

```
[Price, outPR] = price(HWTreePricer,CallableBondEuropean,["all"])
```

```
Price = 107.7023

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta     Gamma     Vega
```

```
    _____    ________    _______    ____

    107.7    -602.56     4086.4      0
```

Price Multiple Option Embedded Fixed Bond Instruments Using Hull-White Model and IRTree Pricer

This example shows the workflow to price multiple callable `OptionEmbeddedFixedBond` instruments with Bermudan exercise styles when you use a `HullWhite` model and an `IRTree` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding);
```

Create `OptionEmbeddedFixedBond` Instrument Objects

Use `fininstrument` to create an `OptionEmbeddedFixedBond` instrument object for three Option Embedded Fixed Bond instuments.

```
Maturity = datetime([2025,1,1 ; 2026,1,1 ; 2027,1,1]);

% Option embedded bond (Bermudan callable bond)
Strike = [100 ; 200 ; 300];
ExerciseDates = datetime([2022,1,1 ; 2023,1,1 ; 2024,1,1]);
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
Period = 1;

CallableBondBermudan = fininstrument("OptionEmbeddedFixedBond",'Maturity',Maturity,...
                                'CouponRate',0.025,'Period', Period, ...
                                'CallSchedule',CallSchedule,'CallExerciseStyle',"Bermudan")
```

```
CallableBondBermudan=3×1 object
  3x1 OptionEmbeddedFixedBond array with properties:

    CouponRate
    Period
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
```

```
    CallDates
    PutDates
    CallSchedule
    PutSchedule
    CallExerciseStyle
    PutExerciseStyle
    Name
```

When you create multiple `OptionEmbeddedFixedBond` instruments and use a timetable for `CallSchedule`, the timetable specification applies to all of the `OptionEmbeddedFixedBond` instruments. The `CallSchedule` input argument does not accept an `NINST`-by-`1` cell array of timetables as input.

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;

HWModel = finmodel("HullWhite",'alpha',AlphaCurve,'sigma',VolCurve);
```

Create `IRTree` Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

             Tree: [1x1 struct]
        TreeDates: [10x1 datetime]
            Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

Price `OptionEmbeddedFixedBond` Instruments

Use `price` to compute the prices and sensitivities for the `OptionEmbeddedFixedBond` instruments.

```
[Price, outPR] = price(HWTreePricer,CallableBondBermudan,["all"])
```

```
Price = 3×1

  104.5001
  102.0649
   97.6664


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results

ans=1×4 table
    Price     Delta     Gamma      Vega
    _____    _______    _____    _______

    104.5    -584.34    4134.2    -166.73


ans=1×4 table
    Price      Delta     Gamma      Vega
    _____    _______    _____    _______

    102.06    -621.72    4850.3    -201.07


ans=1×4 table
    Price      Delta     Gamma      Vega
    _____    _______    _____    _______

    97.666    -743.76    6857.7    -84.933
```

Price Option Embedded Fixed Bond Option Instrument Using Hull-White Model and IRMonteCarlo Pricer

This example shows the workflow to price an `OptionEmbeddedFixedBondOption` instrument when using a `HullWhite` model and an `IRMonteCarlo` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)

myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create OptionEmbeddedFixedBondOption Instrument Object

Use `fininstrument` to create an `OptionEmbeddedFixedBondOption` instrument object.

```
% Option embedded bond (European callable bond)
Maturity = datetime(2022,9,15);
Strike = 100;
ExerciseDates = datetime(2024,1,1);
CallSchedule = timetable(datetime(2020,3,15), 50);
Period = 1;

CallableBondEuropean = fininstrument("OptionEmbeddedFixedBond",'Maturity',Maturity,...
                                     'CouponRate',0.025,'Period',Period, ...
                                     'CallSchedule',CallSchedule)
```

```
CallableBondEuropean =
  OptionEmbeddedFixedBond with properties:

                  CouponRate: 0.0250
                      Period: 1
                       Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                   CallDates: 15-Mar-2020
                    PutDates: [0x1 datetime]
                CallSchedule: [1x1 timetable]
                 PutSchedule: [0x0 timetable]
           CallExerciseStyle: "european"
            PutExerciseStyle: [0x0 string]
                        Name: ""
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.32,'Sigma',0.49)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.3200
    Sigma: 0.4900
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",'Model',HullWhiteModel,'DiscountCurve',myRC,'SimulationDates
```

```
outPricer =
  HWMonteCarlo with properties:

         NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [15-Mar-2019    15-Sep-2019    15-Mar-2020    ...    ]
              Model: [1x1 finmodel.HullWhite]
```

Price `OptionEmbeddedFixedBondOption` Instrument

Use `price` to compute the price and sensitivities for the `OptionEmbeddedFixedBondOption` instrument.

```
[Price,outPR] = price(outPricer,CallableBondEuropean,["all"])
```

```
Price = 58.1882

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma     Vega
    _____    _______    _____    _____

    58.188    -125.43    356.04    18.24
```

Price OptionEmbeddedFixedBond Instrument and Obtain Exercise Probabilities Using Black-Karasinski Model and IRTree Pricer

This example shows the workflow to price a callable `OptionEmbeddedFixedBond` instrument and obtain the exercise probabilities when you use a `BlackKarasinski` model and an `IRTree` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018, 1, 1);
ZeroTimes = calyears(1:4)';
ZeroRates = [0.035; 0.042147; 0.047345; 0.052707];
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
```

```
          Compounding: 1
                Basis: 0
                Dates: [4x1 datetime]
                Rates: [4x1 double]
               Settle: 01-Jan-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `OptionEmbeddedFixedBond` Instrument Object

Use `fininstrument` to create an `OptionEmbeddedFixedBond` instrument object with an American exercise style.

```
CouponRate = 0.0425;
Strike = [95; 98];
ExerciseDates = [datetime(2021,1,1); datetime(2022,1,1)];
Maturity = datetime(2022,1,1);
Period = 1;
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
CallableBond = fininstrument("OptionEmbeddedFixedBond", 'Maturity',Maturity,...
                             'CouponRate',CouponRate,'Period', Period, ...
                             'CallSchedule',CallSchedule,...
                             'CallExerciseStyle', "American",...
                             'Name',"MyCallableBond")
```

```
CallableBond =
  OptionEmbeddedFixedBond with properties:

                CouponRate: 0.0425
                    Period: 1
                     Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                   CallDates: [2x1 datetime]
                    PutDates: [0x1 datetime]
                CallSchedule: [2x1 timetable]
                 PutSchedule: [0x0 timetable]
           CallExerciseStyle: "american"
            PutExerciseStyle: [0x0 string]
                        Name: "MyCallableBond"
```

Create `BlackKarasinski` Model Object

Use `finmodel` to create a `BlackKarasinski` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;
BKModel = finmodel("BlackKarasinski",'alpha',AlphaCurve,'sigma',VolCurve)
```

```
BKModel =
  BlackKarasinski with properties:

    Alpha: 0.1000
    Sigma: 0.0100
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
BKTreePricer = finpricer("IRTree",'Model',BKModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
BKTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [4x1 datetime]
           Model: [1x1 finmodel.BlackKarasinski]
   DiscountCurve: [1x1 ratecurve]
```

Price OptionEmbeddedFixedBond Instrument

Use `price` to compute the price and sensitivities for the `OptionEmbeddedFixedBond` instrument.

```
[Price,  PriceResults]= price(BKTreePricer, CallableBond)
```

```
Price = 92.5235

PriceResults =
  priceresult with properties:

      Results: [1x1 table]
   PricerData: [1x1 struct]
```

Examine the output `PriceResults.PricerData.PriceTree.ExTree`, which contains the exercise indicator arrays. In the cell array, a `1` indicates an exercised option and a `0` indicates an unexercised option.

```
PriceResults.PricerData.PriceTree.ExTree{5}
```

```
ans = 1x7 logical array

   1   1   1   1   1   1   1
```

No options are exercised.

```
PriceResults.PricerData.PriceTree.ExTree{4}
```

```
ans = 1x7 logical array

   0   0   0   0   0   0   0
```

The instrument is exercised at all nodes.

```
PriceResults.PricerData.PriceTree.ExTree{3}
```

```
ans = 1x5 logical array
```

```
   0   0   0   0   0
```

No options are exercised.

```
PriceResults.PricerData.PriceTree.ExTree{2}
```

```
ans = 1x3 logical array
```

```
   0   0   0
```

No options are exercised.

View the probability of reaching each node from the root node using `PriceResults.PricerData.PriceTree.ProbTree`.

```
PriceResults.PricerData.PriceTree.ProbTree{2}
```

```
ans = 1×3
```

```
    0.1667    0.6667    0.1667
```

```
PriceResults.PricerData.PriceTree.ProbTree{3}
```

```
ans = 1×5
```

```
    0.0203    0.2206    0.5183    0.2206    0.0203
```

```
PriceResults.PricerData.PriceTree.ProbTree{4}
```

```
ans = 1×7
```

```
    0.0018    0.0395    0.2370    0.4433    0.2370    0.0395    0.0018
```

```
PriceResults.PricerData.PriceTree.ProbTree{5}
```

```
ans = 1×7
```

```
    0.0018    0.0395    0.2370    0.4433    0.2370    0.0395    0.0018
```

View the exercise probabilities using `PriceResults.PricerData.PriceTree.ExProbTree`. `PriceResults.PricerData.PriceTree.ExProbTree` contains the exercise probabilities. Each element in the cell array is an array containing `0`'s where there is no exercise, or the probability of reaching that node where exercise happens.

```
PriceResults.PricerData.PriceTree.ExProbTree{5}
```

```
ans = 1×7
```

```
    0.0018    0.0395    0.2370    0.4433    0.2370    0.0395    0.0018
```

```
PriceResults.PricerData.PriceTree.ExProbTree{4}
```

```
ans = 1×7

     0     0     0     0     0     0     0
```

```
PriceResults.PricerData.PriceTree.ExProbTree{3}
```

```
ans = 1×5

     0     0     0     0     0
```

```
PriceResults.PricerData.PriceTree.ExProbTree{2}
```

```
ans = 1×3

     0     0     0
```

View the exercise probabilities at each tree level using `PriceResults.PricerData.PriceTree.ExProbsByTreeLevel`. `PriceResults.PricerData.PriceTree.ExProbsByTreeLevel` is an array in which each row holds the exercise probability for a given option at each tree observation time.

```
PriceResults.PricerData.PriceTree.ExProbsByTreeLevel
```

```
ans = 1×5

         0         0         0         0    1.0000
```

More About

Vanilla Bond with Embedded Option

A vanilla coupon bond is a security representing an obligation to repay a borrowed amount at a designated time and to make periodic interest payments until that time.

The issuer of a bond makes the periodic interest payments until the bond matures. At maturity, the issuer pays to the holder of the bond the principal amount owed (face value) and the last interest payment. A vanilla bond with an embedded option is where an option contract has an underlying asset of a vanilla bond.

Stepped Coupon Bond with Callable and Puttable Features

A step-up bond and step-down bond is a debt security with a predetermined coupon structure over time.

With these instruments, coupons increase (step up) or decrease (step down) at specific times during the life of the bond. Stepped coupon bonds can have options features (call and puts).

Amortizing Callable and Puttable Bond

An amortizing callable bond or amortizing puttable bond work under a scheduled `Principal`.

An amortizing callable bond gives the issuer the right to call back the bond, but instead of paying the `Principal` amount at maturity, it repays part of the principal along with the coupon payments. An amortizing puttable bond, repays part of the principal along with the coupon payments and gives the bondholder the right to sell the bond back to the issuer.

Tips

After creating an `OptionEmbeddedFixedBond` object, you can modify the `CallSchedule` and `CallExerciseStyle` using `setCallExercisePolicy`. Or, you can modify the `PutSchedule` and `PutExerciseStyle` values using `setPutExercisePolicy`.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `OptionEmbeddedFixedBond` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`OptionEmbeddedFloatBond` | `finmodel` | `finpricer` | `timetable`

Topics

“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

OptionEmbeddedFloatBond

`OptionEmbeddedFloatBond` instrument object

Description

Create and price a `OptionEmbeddedFloatBond` instrument object for one or more Option Embedded Float Bond instruments using this workflow:

1. Use `fininstrument` to create an `OptionEmbeddedFloatBond` instrument object for one or more Option Embedded Float Bond instruments.
2. Use `finmodel` to specify a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model for the `OptionEmbeddedFloatBond` instrument object.
3. Choose a pricing method.
   - When using a `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` model, use `finpricer` to specify an `IRTree` pricing method for one or more `OptionEmbeddedFloatBond` instruments.
   - When using a `HullWhite`, `BlackKarasinski`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, or `LinearGaussian2F` model, use `finpricer` to specify an `IRMonteCarlo` pricing method for one or more `OptionEmbeddedFloatBond` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for an `OptionEmbeddedFloatBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
OptionEmbeddedFloatBondObj = fininstrument(InstrumentType,'
Spread',spread_value,'Maturity',maturity_date,'
CallSchedule',call_schedule_value)
OptionEmbeddedFloatBondObj = fininstrument(InstrumentType,'
Spread',spread_value,'Maturity',maturity_date,'
PutSchedule',put_schedule_value)
OptionEmbeddedFloatBondObj = fininstrument( ___ ,Name,Value)
```

Description

`OptionEmbeddedFloatBondObj = fininstrument(InstrumentType,'Spread',spread_value,'Maturity',maturity_date,'CallSchedule',call_schedule_value)` creates a `OptionEmbeddedFloatBond` object for one

or more Option Embedded Float Bond instruments by specifying `InstrumentType` and the required name-value pair arguments `Spread`, `Maturity`, and `CallSchedule` sets the properties on page 11-2890 using required name-value pair arguments.

`OptionEmbeddedFloatBond` supports vanilla bonds with embedded options, stepped coupon bonds with embedded options and amortizing bonds with embedded options.

`OptionEmbeddedFloatBondObj = fininstrument(InstrumentType,'Spread',spread_value,'Maturity',maturity_date,'PutSchedule',put_schedule_value)` creates a `OptionEmbeddedFloatBond` object for one or more Option Embedded Float Bond instruments by specifying `InstrumentType` and the required name-value pair arguments `Spread`, `Maturity`, and `PutSchedule` sets the properties on page 11-2890 using required name-value pair arguments.

`OptionEmbeddedFloatBondObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2890 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `OptionEmbeddedFloatBondObj = fininstrument("OptionEmbeddedFloatBond",'Spread',0.01,'Maturity',datetime(2019,1,30),'Period',4,'Basis',5,'Principal',1000,'FirstCouponDate',datetime(2016,1,30),'EndMonthRule',1,'CallSchedule',schedule,'CallExerciseStyle',"american",'ProjectionCurve',ratecurve_obj,'Name',"optionembeddedfloatbond")`. You can specify multiple name-value pairs.

Input Arguments

`InstrumentType` — Instrument type

string with value `"OptionEmbeddedFloatBond"` | string array with values of `"OptionEmbeddedFloatBond"` | character vector with value `'OptionEmbeddedFloatBond'` | cell array of character vectors with values of `'OptionEmbeddedFloatBond'`

Instrument type, specified as a string with the value of `"OptionEmbeddedFloatBond"`, a character vector with the value of `'OptionEmbeddedFloatBond'`, an `NINST`-by-`1` string array with values of `"OptionEmbeddedFloatBond"`, or an `NINST`-by-`1` cell array of character vectors with values of `'OptionEmbeddedFloatBond'`.

Data Types: `char` | `cell` | `string`

`OptionEmbeddedFloatBond` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `OptionEmbeddedFloatBondObj = fininstrument("OptionEmbeddedFloatBond",'Spread',0.01,'Maturity',datetime(2019,1,30),'Period',4,'Basis',5,'Principal',1000,'FirstCouponDate',datetime(2016,1,30),'EndMonthRule',1,'CallSchedule',schedule,'CallExerciseStyle',"american",'ProjectionCurve',ratecurve_obj,'Name',"optionembeddedfloatbond")`

Required `OptionEmbeddedFloatBond` Name-Value Pair Arguments

`Spread` — Number of basis points over the reference rate

nonnegative numeric | vector of nonnegative numeric

Number of basis points over the reference rate, specified as the comma-separated pair consisting of `'Spread'` and a scalar nonnegative numeric or an `NINST`-by-`1` vector of nonnegative numeric.

Data Types: `double`

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as the comma-separated pair consisting of `'Maturity'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OptionEmbeddedFloatBond` also accepts serial date numbers as inputs, but they are not recommended.

`CallSchedule` — Call schedule
timetable

Call schedule, specified as the comma-separated pair consisting of `'CallSchedule'` and a timetable of call dates and strikes.

If you use a date character vector or date string for the dates in this timetable, the format must be recognizable by `datetime` because the `CallSchedule` property is stored as a datetime.

Note The `OptionEmbeddedFloatBond` instrument supports either `CallSchedule` and `CallExerciseStyle` or `PutSchedule` and `PutExerciseStyle`, but not both.

If you are creating one or more `OptionEmbeddedFloatBond` instruments and use a timetable, the timetable specification applies to all of the `OptionEmbeddedFloatBond` instruments. `CallSchedule` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `timetable`

`PutSchedule` — Call schedule
timetable

Put schedule, specified as the comma-separated pair consisting of `'PutSchedule'` and a timetable of call dates and strikes.

If you use a date character vector or date string for dates in this timetable, the format must be recognizable by `datetime` because the `PutSchedule` property is stored as a datetime.

Note The `OptionEmbeddedFloatBond` instrument supports either `CallSchedule` and `CallExerciseStyle` or `PutSchedule` and `PutExerciseStyle`, but not both.

If you are creating one or more `OptionEmbeddedFloatBond` instruments and use a timetable, the timetable specification applies to all of the `OptionEmbeddedFloatBond` instruments. `PutSchedule` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `timetable`

Optional OptionEmbeddedFloatBond Name-Value Pair Arguments

Reset — Frequency of payments per year
2 (default) | scalar integer | vector of integers

Frequency of payments per year, specified as the comma-separated pair consisting of `'Reset'` and a scalar integer or an `NINST`-by-1 vector of integers. Values for `Reset` are: 1, 2, 3, 4, 6, and 12.

Data Types: `double`

CallExerciseStyle — Call option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with values of `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'` | cell array of character vectors with values of `'European'`, `'American'`, or `'Bermudan'`

Call option exercise style, specified as the comma-separated pair consisting of `'CallExerciseStyle'` and a scalar string or character vector or an `NINST`-by-1 cell array of character vectors or string array.

Note The `CallSchedule` is a timetable of call dates and strikes. If you do not specify a `CallExerciseStyle`, then based on the `CallSchedule` specification, a default value of `CallExerciseStyle` is assigned as follows:

- If there is one exercise date in the `CallSchedule`, then the `CallExerciseStyle` is an `"European"`.
- If there are two exercise dates in the `CallSchedule`, then the `CallExerciseStyle` is an `"American"` with a start date and maturity.
- If there are more than two exercise dates in the `CallSchedule`, then the `CallExerciseStyle` is an `"Bermudan"`.

If the you define a `CallExerciseStyle` and this is not consistent with what you have specified in the `CallSchedule`, you receive an error message.

Data Types: `string` | `cell` | `char`

PutExerciseStyle — Put option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with values of `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'` | cell array of character vectors with values of `'European'`, `'American'`, or `'Bermudan'`

Put option exercise style, specified as the comma-separated pair consisting of `'PutExerciseStyle'` and a scalar string or character vector or an `NINST`-by-1 cell array of character vectors or string array.

Note The `PutSchedule` is a timetable of call dates and strikes. If you do not specify a `PutExerciseStyle`, then based on the `PutSchedule` specification, a default value of `PutExerciseStyle` is assigned as follows:

- If there is one exercise date in the `PutSchedule`, then the `PutExerciseStyle` is an `"European"`.

- If there are two exercise dates in the `PutSchedule`, then the `PutExerciseStyle` is an `"American"` with a start date and maturity.
- If there are more than two exercise dates in the `PutSchedule`, then the `PutExerciseStyle` is an `"Bermudan"`.

If the you define a `PutExerciseStyle` and this is not consistent with what you have specified in the `PutSchedule`, you receive an error message.

Data Types: `string` | `cell` | `char`

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and scalar integer or an `NINST`-by-`1` vector of integers using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Notional principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Notional principal amount or principal value schedule, specified as the comma-separated pair consisting of `'Principal'` and a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

`Principal` accepts a `timetable`, where the first column is dates and the second column is the associated notional principal value. The date indicates the last day that the principal value is valid.

Note If you are creating one or more `OptionEmbeddedFloatBond` instruments and use a timetable, the timetable specification applies to all of the `OptionEmbeddedFloatBond` instruments. `Principal` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `double` | `timetable`

`DaycountAdjustedCashFlow` — Flag indicating whether cash flow adjusts for day count convention
`false` (default) | scalar logical value of `true` or `false` | vector of logicals with values of `true` or `false`

Flag indicating whether cash flow adjusts for day count convention, specified as the comma-separated pair consisting of `'DaycountAdjustedCashFlow'` and a scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and dates using an `NINST`-by-`1` vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
OptionEmbeddedFixedBondObj = fininstrument("OptionEmbeddedFixedBond",'CouponRate',0.34,'Maturity
'CallSchedule',schedule,'CallExerciseStyle',"american",'Holidays',H)
```

To support existing code, `OptionEmbeddedFloatBond` also accepts serial date numbers as inputs, but they are not recommended.

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`true` (in effect) (default) | scalar logical value of `true` or `false` | vector of logicals with values of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar logical or an `NINST`-by-`1` vector of logicals values of `true` or `false`.

- If you set `EndMonthRule` to `false`, the software ignores the rule, meaning that a payment date is always the same numerical day of the month.
- If you set `EndMonthRule` to `true`, the software sets the rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
`NaT` (default) | datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OptionEmbeddedFloatBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `IssueDate` property is stored as a datetime.

`FirstCouponDate` — Irregular first coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OptionEmbeddedFloatBond` also accepts serial date numbers as inputs, but they are not recommended.

When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `FirstCouponDate` property is stored as a datetime.

`LastCouponDate` — Irregular last coupon date
`NaT` (default) | datetime array | string array | date character vector

Irregular last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OptionEmbeddedFloatBond` also accepts serial date numbers as inputs, but they are not recommended.

If you specify `LastCouponDate` but not `FirstCouponDate`, `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at `LastCouponDate`,

regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify `LastCouponDate`, the cash flow payment dates are determined from other inputs.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `LastCouponDate` property is stored as a datetime.

`StartDate` — Forward starting date of payments
`NaT` (default) | datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `OptionEmbeddedFloatBond` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `StartDate` property is stored as a datetime.

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for the instrument, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Spread` — Number of basis points over the reference rate
scalar nonnegative numeric | vector of nonnegative numeric

Number of basis points over the reference rate, returned as a scalar nonnegative numeric or an `NINST`-by-`1` vector of nonnegative numeric values.

Data Types: `double`

`Maturity` — Maturity date
scalar datetime | vector of datetimes

Maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`CallSchedule` — Call schedule
timetable

Call schedule, returned as a timetable.

Data Types: `cell` | `datetime`

`PutSchedule` — Call schedule
timetable

Put schedule, returned as a timetable.

Data Types: `cell` | `datetime`

`Reset` — Frequency of payments per year
`2` (default) | scalar integer | vector of integers

Frequency of payments per year, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Basis` — Day count basis
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Principal` — Notional principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Notional principal amount or principal value schedule, returned as a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

Data Types: `timetable` | `double`

`DaycountAdjustedCashFlow` — Flag indicating whether cash flow adjusts for day count convention
`false` (default) | scalar logical value of `true` or `false` | vector of logicals with values of `true` or `false`

Flag indicating whether cash flow adjusted for day count convention, returned as scalar logical or an `NINST`-by-`1` vector of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array

Business day conventions, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`true` (in effect) (default) | scalar logical value of `true` or `false` | vector of logicals with values of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, returned as a scalar logical or an `NINST`-by-`1` vector of logicals.

Data Types: `logical`

`IssueDate` — Bond issue date
`NaT` (default) | scalar datetime | vector of datetimes

Bond issue date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`FirstCouponDate` — Irregular first coupon date
`NaT` (default) | scalar datetime | vector of datetimes

Irregular first coupon date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`LastCouponDate` — Irregular last coupon date
`NaT` (default) | scalar datetime | vector of datetimes

Irregular last coupon date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`StartDate` — Forward starting date of payments
`NaT` (default) | scalar datetime

Forward starting date of payments, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`CallExerciseStyle` — Call option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with values of `"European"`, `"American"`, or `"Bermudan"`

This property is read-only.

Call option exercise style, returned as a string or an `NINST`-by-`1` string array with values of `"European"`, `"American"`, or `"Bermudan"`.

Data Types: `string`

`PutExerciseStyle` — Put option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with values of `"European"`, `"American"`, or `"Bermudan"`

This property is read-only.

Put option exercise style, returned as a string or an `NINST`-by-`1` string array with values of `"European"`, `"American"`, or `"Bermudan"`.

Data Types: `string`

`Name` — User-defined name for instrument
`"  "` (default) | string | string array

User-defined name for the instrument, returned as a string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

setCallExercisePolicy	Set call exercise policy for OptionEmbeddedFixedBond, OptionEmbeddedFloatBond, or ConvertibleBond instrument
setPutExercisePolicy	Set put exercise policy for OptionEmbeddedFixedBond, OptionEmbeddedFloatBond, or ConvertibleBond instrument

Examples

Price Option Embedded Float Bond Instruments Using Hull-White Model and IRTree Pricer

This example shows the workflow to price American, European, and Bermudan exercise styles for three callable `OptionEmbeddedFloatBond` instruments when you use a `HullWhite` model and an `IRTree` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding);
```

Create `OptionEmbeddedFloatBond` Instrument Objects

Use `fininstrument` to create three `OptionEmbeddedFloatBond` instrument objects with different exercise styles.

```
Maturity = datetime(2024,1,1);

% Option embedded float bond (Bermudan callable bond)
Strike = [100; 100];
ExerciseDates = [datetime(2020,1,1); datetime(2024,1,1)];
Reset = 1;
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});

CallableBondBermudan = fininstrument("OptionEmbeddedFloatBond",'Maturity',Maturity,...
                              'Spread',0.025,'Reset',Reset, ...
                              'CallSchedule',CallSchedule,'CallExerciseStyle', "bermudan")
```

```
CallableBondBermudan =
  OptionEmbeddedFloatBond with properties:

                    Spread: 0.0250
            ProjectionCurve: [0x0 ratecurve]
                ResetOffset: 0
                      Reset: 1
                      Basis: 0
               EndMonthRule: 1
                  Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                   Holidays: NaT
```

```
                IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                StartDate: NaT
                 Maturity: 01-Jan-2024
                CallDates: [2x1 datetime]
                 PutDates: [0x1 datetime]
              CallSchedule: [2x1 timetable]
               PutSchedule: [0x0 timetable]
          CallExerciseStyle: "bermudan"
           PutExerciseStyle: [0x0 string]
                       Name: ""
```

```
% Option embedded float bond (American callable bond)
Strike = 100;
ExerciseDates = datetime(2024,1,1);
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
Reset = 1;

CallableBondAmerican = fininstrument("OptionEmbeddedFloatBond",'Maturity',Maturity,...
                              'Spread',0.025,'Reset', Reset, ...
                              'CallSchedule',CallSchedule,'CallExerciseStyle',"american")
```

```
CallableBondAmerican =
  OptionEmbeddedFloatBond with properties:

                    Spread: 0.0250
            ProjectionCurve: [0x0 ratecurve]
                ResetOffset: 0
                      Reset: 1
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                   Holidays: NaT
                IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                StartDate: NaT
                 Maturity: 01-Jan-2024
                CallDates: 01-Jan-2024
                 PutDates: [0x1 datetime]
              CallSchedule: [1x1 timetable]
               PutSchedule: [0x0 timetable]
          CallExerciseStyle: "american"
           PutExerciseStyle: [0x0 string]
                       Name: ""
```

```
% Option embedded float bond (European callable bond)
Strike = 100;
ExerciseDates = datetime(2024,1,1);
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
Reset = 1;

CallableBondEuropean = fininstrument("OptionEmbeddedFloatBond",'Maturity',Maturity,...
```

```
                            'Spread',0.025,'Reset',Reset, ...
                            'CallSchedule',CallSchedule)

CallableBondEuropean =
  OptionEmbeddedFloatBond with properties:

                     Spread: 0.0250
            ProjectionCurve: [0x0 ratecurve]
                ResetOffset: 0
                      Reset: 1
                      Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
            FirstCouponDate: NaT
             LastCouponDate: NaT
                  StartDate: NaT
                   Maturity: 01-Jan-2024
                  CallDates: 01-Jan-2024
                   PutDates: [0x1 datetime]
               CallSchedule: [1x1 timetable]
                PutSchedule: [0x0 timetable]
          CallExerciseStyle: "european"
           PutExerciseStyle: [0x0 string]
                       Name: ""
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;

HWModel = finmodel("HullWhite",'alpha',AlphaCurve,'sigma',VolCurve);
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [10x1 datetime]
           Model: [1x1 finmodel.HullWhite]
   DiscountCurve: [1x1 ratecurve]
```

Price OptionEmbeddedFixedBond Instruments

Use `price` to compute the price and sensitivities for the three `OptionEmbeddedFixedBond` instruments.

```
[Price, outPR] = price(HWTreePricer,CallableBondBermudan,["all"])

Price = 104.9598

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×4 table
    Price      Delta     Gamma     Vega
    _____    _______    _____    ____

    104.96    -7.3926    19.597     0


[Price, outPR] = price(HWTreePricer,CallableBondAmerican,["all"])

Price = 100

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×4 table
    Price    Delta    Gamma    Vega
    _____    _____    _____    ____

     100       0        0       0


[Price, outPR] = price(HWTreePricer,CallableBondEuropean,["all"])

Price = 114.5571

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×4 table
    Price      Delta     Gamma        Vega
    _____    _______    _____    __________

    114.56    -50.006    262.58    -2.8422e-10
```

Price Option Embedded Float Bond Option Instrument Using Hull-White Model and IRMonteCarlo Pricer

This example shows the workflow to price an `OptionEmbeddedFloatBondOption` instrument when using a `HullWhite` model and an `IRMonteCarlo` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `OptionEmbeddedFloatBondOption` Instrument Object

Use `fininstrument` to create an `OptionEmbeddedFloatBondOption` instrument object.

```
% Option embedded float bond (European callable bond)
Maturity = datetime(2022,9,15);
Strike = 100;
ExerciseDates = datetime(2024,1,1);
CallSchedule = timetable(datetime(2020,3,15), 50);
Reset = 1;

CallableBondEuropean = fininstrument("OptionEmbeddedFloatBond",'Maturity',Maturity,...
                               'Spread',0.025,'Reset',Reset, ...
                               'CallSchedule',CallSchedule)
```

```
CallableBondEuropean =
  OptionEmbeddedFloatBond with properties:

                    Spread: 0.0250
            ProjectionCurve: [0x0 ratecurve]
                ResetOffset: 0
                      Reset: 1
                      Basis: 0
               EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
```

```
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                   CallDates: 15-Mar-2020
                    PutDates: [0x1 datetime]
                CallSchedule: [1x1 timetable]
                 PutSchedule: [0x0 timetable]
           CallExerciseStyle: "european"
            PutExerciseStyle: [0x0 string]
                        Name: ""
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.32,'Sigma',0.49)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.3200
    Sigma: 0.4900
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",'Model',HullWhiteModel,'DiscountCurve',myRC,'SimulationDates
```

```
outPricer =
  HWMonteCarlo with properties:

         NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [15-Mar-2019    15-Sep-2019    15-Mar-2020    ...    ]
              Model: [1x1 finmodel.HullWhite]
```

Price OptionEmbeddedFloatBondOption Instrument

Use `price` to compute the price and sensitivities for the `OptionEmbeddedFloatBondOption` instrument.

```
[Price,outPR] = price(outPricer,CallableBondEuropean,["all"])
```

```
Price = 51.3788
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
```

```
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta      Gamma      Vega
    ______    ______    _______    _______

    51.379    61.634    -81.051    -7.0508
```

Price Multiple Option Embedded Float Bond Instruments Using Hull-White Model and IRTree Pricer

This example shows the workflow to price multiple `OptionEmbeddedFloatBond` instruments with Bermudan exercise styles when you use a `HullWhite` model and an `IRTree` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding);
```

Create `OptionEmbeddedFloatBond` Instrument Objects

Use `fininstrument` to create an `OptionEmbeddedFloatBond` instrument object for three Option Embedded Float Bond instruments with a `Bermudan` exercise style.

```
Maturity = datetime([2025,1,1 ; 2026,1,1 ; 2027,1,1]);

% Option embedded float bond (Bermudan callable bond)
Strike = [101 ; 102 ; 103];
ExerciseDates = datetime([2022,1,1 ; 2023,1,1 ; 2024,1,1]);
CallSchedule =  timetable(ExerciseDates,Strike,'VariableNames',{'Strike Schedule'});
Reset = 1;

CallableBondBermudan = fininstrument("OptionEmbeddedFloatBond",'Maturity',Maturity,...
                              'Spread',[0.001; 0.0015; 0.002],'Reset', Reset, ...
                              'CallSchedule',CallSchedule,'CallExerciseStyle',"bermudan")
```

```
CallableBondBermudan=3×1 object
  3x1 OptionEmbeddedFloatBond array with properties:

    Spread
    ProjectionCurve
    ResetOffset
    Reset
    Basis
    EndMonthRule
    Principal
    DaycountAdjustedCashFlow
```

```
    BusinessDayConvention
    Holidays
    IssueDate
    FirstCouponDate
    LastCouponDate
    StartDate
    Maturity
    CallDates
    PutDates
    CallSchedule
    PutSchedule
    CallExerciseStyle
    PutExerciseStyle
    Name
```

When you create multiple `OptionEmbeddedFloatBond` instruments and use a timetable for `CallSchedule`, the timetable specification applies to all of the `OptionEmbeddedFloatBond` instruments. The `CallSchedule` input argument does not accept an `NINST`-by-`1` cell array of timetables as input.

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;

HWModel = finmodel("HullWhite",'alpha',AlphaCurve,'sigma',VolCurve);
```

Create `IRTree` Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

             Tree: [1x1 struct]
        TreeDates: [10x1 datetime]
            Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

Price `OptionEmbeddedFixedBond` Instruments

Use `price` to compute the prices and sensitivities for the three `OptionEmbeddedFixedBond` instruments.

```
[Price, outPR] = price(HWTreePricer,CallableBondBermudan,"all")
```

```
Price = 3×1

  100.6713
```

```
  101.1327
  101.6643


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData


outPR.Results

ans=1×4 table
    Price      Delta     Gamma       Vega
    _____    ________    _____    ___________

    100.67    -2.6133    15.33    -4.2633e-10


ans=1×4 table
    Price      Delta     Gamma        Vega
    _____    ________    ______    ___________

    101.13    -4.9053    31.676    -5.6843e-10


ans=1×4 table
    Price      Delta     Gamma       Vega
    _____    ________    ______    _________

    101.66    -7.8748    55.171    -0.066246
```

More About

Floating-Rate Note with Embedded Options

A floating-rate note with an embedded option enables floating-rate notes to have early redemption features.

A floating-rate note with an embedded option gives investors or issuers the option to retire the outstanding principal prior to maturity. An embedded call option gives the right to retire the note prior to the maturity date (callable floater), and an embedded put option gives the right to sell the note back at a specific price (puttable floater).

For more information, see “Floating-Rate Note with Embedded Options” on page 2-11.

Tips

After creating an `OptionEmbeddedFixedBond` object, you can modify the `CallSchedule` and `CallExerciseStyle` using `setCallExercisePolicy`. Or, you can modify the `PutSchedule` and `PutExerciseStyle` values using `setPutExercisePolicy`.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `OptionEmbeddedFloatBond` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`OptionEmbeddedFixedBond` | `finmodel` | `finpricer` | `timetable`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22

Spread

`Spread` instrument object

Description

Create and price a `Spread` instrument object for one or more Spread instruments using this workflow:

1. Use `fininstrument` to create a `Spread` instrument object for one or more Spread instruments.
2. Use `finmodel` to specify a `BlackScholes` or `Bachelier` model for the `Spread` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify a `Kirk`, `BjerksundStensland`, or `AssetMonteCarlo` pricing method for one or more `Spread` instruments.
   - When using a `Bachelier` model, use `finpricer` to specify an `AssetMonteCarlo` pricing method for one or more `Spread` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Spread` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
SpreadObj = fininstrument(InstrumentType,'Strike',strike_value,'
ExerciseDate',exercise_date)
SpreadObj = fininstrument( ___ ,Name,Value)
```

Description

`SpreadObj = fininstrument(InstrumentType,'Strike',strike_value,'ExerciseDate',exercise_date)` creates a `Spread` object for one or more Spread instruments by specifying `InstrumentType` and sets the properties on page 11-2905 for the required name-value pair arguments `Strike` and `ExerciseDate`.

`SpreadObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2905 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `SpreadObj = fininstrument("Spread",'Strike',100,'ExerciseDate',datetime(2019,1,30),'OptionType',"put",'ExerciseStyle',"American",'Name',"spread_instrument")` creates a `Spread` put option with an American exercise. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Spread"` | string array with values of `"Spread"` | character vector with value `'Spread'` | cell array of character vectors with values of `'Spread'`

Instrument type, specified as a string with the value of `"Spread"`, a character vector with the value of `'Spread'`, an `NINST`-by-`1` string array with values of `"Spread"`, or an `NINST`-by-`1` cell array of character vectors with values of `'Spread'`.

Data Types: `char` | `cell` | `string`

Spread Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `SpreadObj = fininstrument("Spread",'Strike',100,'ExerciseDate',datetime(2019,1,30),'OptionType',"put",'ExerciseStyle',"American",'Name',"spread_instrument")`

Required Spread Name-Value Pair Arguments

`Strike` — Option strike price value

nonnegative numeric | nonnegative numeric vector

Option strike price value, specified as the comma-separated pair consisting of `'Strike'` and a scalar nonnegative numeric or an `NINST`-by-`1` nonnegative numeric vector.

Data Types: `double`

`ExerciseDate` — Option exercise date

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of `'ExerciseDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one `ExerciseDate` on the option expiry date.

To support existing code, `Spread` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `ExerciseDate` property is stored as a datetime.

Optional Spread Name-Value Pair Arguments

`OptionType` — Option type

`"call"` (default) | string with value `"call"` or `"put"` | string array with values of `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values of `'call'` or `'put'`

Option type, specified as the comma-separated pair consisting of `'OptionType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` | string array with values of `"European"` | character vector with value `'European'` | cell array of character vectors with values of `'European'`

Option exercise style, specified as the comma-separated pair consisting of `'ExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `string` | `cell` | `char`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Strike` — Option strike price value
nonnegative numeric | nonnegative numeric vector

Option strike price value, returned as a scalar nonnegative numeric or an `NINST`-by-`1` nonnegative numeric vector.

Data Types: `double`

`ExerciseDate` — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with value `"call"` or `"put"`

Option type, returned as a scalar string or an `NINST`-by-`1` string array with values of `"call"` or `"put"`.

Data Types: `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` | string array with values of `"European"`

Option exercise style, returned as a string or an `NINST`-by-`1` string array with values of `"European"`.

Data Types: `string`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Examples

Price Spread Instrument with European Option Using Black-Scholes Model and Bjerksund-Stensland Pricer

This example shows the workflow to price a `Spread` instrument with a European option when using a `BlackScholes` model and a `BjerksundStensland` pricing method.

Create Spread Instrument Object

Use `fininstrument` to create a `Spread` instrument object.

```
SpreadOpt = fininstrument("Spread",'Strike',105,'ExerciseDate',datetime(2021,9,15),'OptionType','
```

```
SpreadOpt =
  Spread with properties:

      OptionType: "put"
          Strike: 105
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2021
             Name: "spread_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',[0.2,0.1])
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: [0.2000 0.1000]
    Correlation: [2x2 double]
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 12
             Dates: 15-Sep-2023
```

```
              Rates: 0.0350
             Settle: 15-Sep-2018
       InterpMethod: "linear"
  ShortExtrapMethod: "next"
   LongExtrapMethod: "previous"
```

Create BjerksundStensland Pricer Object

Use finpricer to create a BjerksundStensland pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',[103
```

```
outPricer =
  BjerksundStensland with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: [103 105]
    DividendValue: [0.0250 0.0280]
     DividendType: "continuous"
```

Price Spread Instrument

Use price to compute the price and sensitivities for the Spread instrument.

```
[Price, outPR] = price(outPricer,SpreadOpt,["all"])
```

```
Price = 95.9884
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price           Delta                  Gamma                  Lambda                Vega
    _____    ___________________    ___________________________    ____________________    ____

    95.988    -0.8916    0.90457    0.0021316    0.00048175    -0.95673    0.97064    13.582
```

Price Multiple Spread Instruments with European Option Using Black-Scholes Model and Bjerksund-Stensland Pricer

This example shows the workflow to price multiple Spread instruments with a European option when using a BlackScholes model and a BjerksundStensland pricing method.

Create Spread Instrument Object

Use fininstrument to create a Spread instrument object for three Spread instruments.

```
SpreadOpt = fininstrument("Spread",'Strike',[105 ; 120 ; 150],'ExerciseDate',datetime([2021,9,15
```

```
SpreadOpt=3×1 object
  3x1 Spread array with properties:

    OptionType
    Strike
    ExerciseStyle
    ExerciseDate
    Name
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',[0.2,0.1])
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: [0.2000 0.1000]
    Correlation: [2x2 double]
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
              Dates: 15-Sep-2023
              Rates: 0.0350
             Settle: 15-Sep-2018
       InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BjerksundStensland Pricer Object

Use `finpricer` to create a `BjerksundStensland` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',[103
```

```
outPricer =
  BjerksundStensland with properties:

    DiscountCurve: [1x1 ratecurve]
```

```
           Model: [1x1 finmodel.BlackScholes]
       SpotPrice: [103 160]
   DividendValue: [0.0250 0.0280]
    DividendType: "continuous"
```

Price Spread Instruments

Use `price` to compute the prices and sensitivities for the `Spread` instruments.

```
[Price, outPR] = price(outPricer,SpreadOpt,["all"])
```

```
Price = 3×1

  146.1732
  159.1989
  185.5513


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price            Delta                  Gamma                  Lambda                V
    _____    ____________________    ____________________    ____________________    ______

    146.17    -0.91985     0.91683    0.00057696    7.9581e-05    -0.64817     0.64604    3.6848


ans=1×7 table
    Price           Delta                  Gamma                  Lambda                Ve
    _____    ____________________    ____________________    ____________________    ______

    159.2    -0.92024     0.91557    0.00042064    5.4001e-05    -0.59539     0.59237    2.7723


ans=1×7 table
    Price            Delta                  Gamma                  Lambda                Veg
    _____    ____________________    ____________________    ____________________    ______

    185.55    -0.92123     0.91439    0.000216    1.9895e-05    -0.51138     0.50758    1.4478
```

Price Spread Instrument for a Commodity Using Black-Scholes Model and Analytic Pricers

This example shows the workflow to price a commodity `Spread` instrument when you use a `BlackScholes` model and `Kirk` and `BjerksundStensland` analytic pricing methods.

Understanding Crack Spread Options

In the petroleum industry, refiners are concerned about the difference between their input costs (crude oil) and output prices (refined products — gasoline, heating oil, diesel fuel, and so on). The

differential between these two underlying commodities is referred to as a *crack spread*. It represents the profit margin between crude oil and the refined products.

A *spread option* is an option on the spread where the holder has the right, but not the obligation, to enter into a spot or forward spread contract. Crack spread options are often used to protect against declines in the crack spread or to monetize volatility or price expectations on the spread.

Define the Commodity

Assume that current gasoline prices are strong, and you want to model a crack spread option strategy to protect the gasoline margin. A crack spread option strategy is used to maintain profits for the following season. The WTI crude oil futures are at $93.20 per barrel and RBOB gasoline futures contract are at $2.85 per gallon.

```
Strike = 20;
Rate = 0.05;

Settle = datetime(2020,1,1);
Maturity = datemnth(Settle,3);

% Price and volatility of RBOB gasoline
PriceGallon1 = 2.85;          % Dollars per gallon
Price1 = PriceGallon1 * 42;   % Dollars per barrel
Vol1 = 0.29;

% Price and volatility of WTI crude oil
Price2 = 93.20;         % Dollars per barrel
Vol2 = 0.36;

% Correlation between the prices of the commodities
Corr = 0.42;
```

Create Spread Instrument Object

Use `fininstrument` to create a `Spread` instrument object.

```
SpreadOpt = fininstrument("Spread", 'ExerciseDate', Maturity, 'Strike', Strike,'ExerciseStyle',"e
```

```
SpreadOpt =
  Spread with properties:

       OptionType: "call"
           Strike: 20
    ExerciseStyle: "european"
     ExerciseDate: 01-Apr-2020
             Name: "spread_instrument"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes", 'Volatility', [Vol1,Vol2], 'Correlation', [1 Corr; C
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
ZeroCurve = ratecurve('zero', Settle, Maturity, Rate, 'Basis', 1);
```

Create BjerksundStensland Pricer Object

Use finpricer to create a BjerksundStensland pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
BJSPricer = finpricer("Analytic", 'Model', BlackScholesModel, 'SpotPrice', [Price1 , Price2], 'Di
```

Create Kirk Pricer Object

Use finpricer to create a Kirk pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
KirkPricer = finpricer("Analytic", 'Model', BlackScholesModel,'SpotPrice', [Price1 , Price2], 'Di
```

Price Spread Instrument Using BjerksundStensland and Kirk Analytic Pricing Methods

Use price to compute the price and sensitivities for the commodity Spread instrument.

```
[PriceKirk, outPR_Kirk] = price(KirkPricer, SpreadOpt, "all");
[PriceBJS,  outPR_BJS]  = price(BJSPricer,  SpreadOpt, "all");

[outPR_Kirk.Results; outPR_BJS.Results]
```

```
ans=2×7 table
    Price           Delta                 Gamma                 Lambda                Vega
    _____    ____________________    ____________________    _________________    _____________

    11.19    0.67224    -0.60665    0.019081    0.021662    7.1907    -6.4891    11.299    9.8869
     11.2    0.67371    -0.60816    0.018992    0.021572    7.2003    -6.4997    11.198    9.9878
```

Price Spread Instrument with American Option Using Black-Scholes Model and Asset Monte-Carlo Pricer

This example shows the workflow to price a Spread instrument with an American option when using a BlackScholes model and an AssetMonteCarlo pricing method.

Create Spread Instrument Object

Use fininstrument to create a Spread instrument object.

```
SpreadOpt = fininstrument("Spread",'Strike',100,'ExerciseDate',datetime(2021,9,15),'OptionType','
```

```
SpreadOpt =
  Spread with properties:

       OptionType: "put"
           Strike: 100
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2021
             Name: "spread_option"
```

Create BlackScholes Model Object

Use finmodel to create a BlackScholes model object.

```
Corr = 0.42;
BlackScholesModel = finmodel("BlackScholes","Volatility",[0.3,0.1],"Correlation", [1 Corr;Corr 1]

BlackScholesModel =
  BlackScholes with properties:

     Volatility: [0.3000 0.1000]
    Correlation: [2x2 double]
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)

myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo","DiscountCurve",myRC,"Model",BlackScholesModel,'SpotPrice

outPricer =
  GBMMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: [100 95]
    SimulationDates: 15-Sep-2021
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.BlackScholes]
       DividendType: ["continuous"    "continuous"]
      DividendValue: [0 0.0100]
```

Price Spread Instrument

Use `price` to compute the price and sensitivities for the `Spread` instrument.

```
[Price, outPR] = price(outPricer,SpreadOpt,["all"])
```

```
Price = 95

outPR =
  priceresult with properties:

      Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta           Gamma                Lambda          Rho     Theta      Vega
    _____    ________    _________________    ____________________    ___    _____    _______

     95       -1     1    0    3.1492e-14     -1.0526           1     0       0       0    0
```

Price Spread Instrument with American Option Using Bachelier Model and Asset Monte-Carlo Pricer

This example shows the workflow to price a `Spread` instrument with an American option when using a `Bachelier` model and an `AssetMonteCarlo` pricing method.

Create Spread Instrument Object

Use `fininstrument` to create a `Spread` instrument object.

```
SpreadOpt = fininstrument("Spread",'Strike',100,'ExerciseDate',datetime(2021,9,15),'OptionType','
```

```
SpreadOpt =
  Spread with properties:

       OptionType: "put"
           Strike: 100
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2021
             Name: "spread_option"
```

Create `Bachelier` Model Object

Use `finmodel` to create a `Bachelier` model object.

```
Corr = 0.42;
BachelierModel = finmodel("BlackScholes","Volatility",[0.3,0.1],"Correlation", [1 Corr;Corr 1])
```

```
BachelierModel =
  BlackScholes with properties:

     Volatility: [0.3000 0.1000]
    Correlation: [2x2 double]
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo","DiscountCurve",myRC,"Model",BachelierModel,'SpotPrice',
```

```
outPricer =
  GBMMonteCarlo with properties:

     DiscountCurve: [1x1 ratecurve]
         SpotPrice: [100 95]
    SimulationDates: 15-Sep-2021
         NumTrials: 1000
     RandomNumbers: []
             Model: [1x1 finmodel.BlackScholes]
      DividendType: ["continuous"    "continuous"]
     DividendValue: [0 0.0100]
```

Price Spread Instrument

Use `price` to compute the price and sensitivities for the `Spread` instrument.

```
[Price, outPR] = price(outPricer,SpreadOpt,["all"])
```

```
Price = 95
```

```
outPR =
  priceresult with properties:

      Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta          Gamma              Lambda          Rho     Theta     Vega
```

```
_____    ________    ______________    ___________________    ___    _____    _____

 95       -1     1    0    3.1492e-14    -1.0526           1     0       0       0    0
```

More About

Spread Option

A spread option is written on the difference of two underlying assets.

For example, a European call on the difference of two assets *X1* and *X2* has the following pay off at maturity:

$$\max(X1 - X2 - K, 0)$$

K is the strike price.

For more information, see “Spread Option” on page 3-30.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Spread` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

Swap

Swap instrument object

Description

Create and price a Swap instrument object for one or more Swap instruments using this workflow:

1. Use fininstrument to create a Swap instrument object for one or more Swap instruments.
2. Use ratecurve to specify a curve model for the Swap instrument object or use finmodel to specify a HullWhite, BlackKarasinski, BlackDermanToy, or LinearGaussian2F model.
3. Choose a pricing method.
   - When using a ratecurve, use finpricer to specify a Discount pricing method
   - When using a HullWhite, BlackKarasinski, or BlackDermanToy model , use an IRTree pricing method for one or more Swap instruments.
   - When using a HullWhite, BlackKarasinski, or LinearGaussian2F model, use finpricer to specify an IRMonteCarlo pricing method for one or more Swap instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a Swap instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
SwapInstrument = fininstrument(InstrumentType,'Maturity',maturity_date,'
LegRate',leg_rate)
SwapInstrument = fininstrument( ___ ,Name,Value)
```

Description

`SwapInstrument = fininstrument(InstrumentType,'Maturity',maturity_date,'LegRate',leg_rate)` creates a Swap object for one or more Swap instruments by specifying InstrumentType and sets the properties on page 11-2921 for the required name-value pair arguments Maturity and LegRate.

The Swap instrument supports vanilla swaps, amortizing swaps and forward swaps. You can use the Swap instrument for a single currency swap but not a cross-currency swap. For more information on a Swap instrument, see “More About” on page 11-2933.

`SwapInstrument = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2921 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, SwapInstrument =

```
fininstrument("Swap",'Maturity',datetime(2019,1,30),'LegRate',[0.06
0.12],'LegType',
["fixed","fixed"],'Basis',1,'Notional',100,'StartDate',datetime(2016,1,30),'D
aycountAdjustedCashFlow',true,'BusinessDayConvention',"follow",'ProjectionCur
ve',ratecurve,'Name',"swap_instrument")
```
creates a `Swap` option with a maturity of January 30, 2019. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Swap"` | string array with values of `"Swap"` | character vector with value `'Swap'` | cell array of character vectors with values of `'Swap'`

Instrument type, specified as a string with the value of `"Swap"`, a character vector with the value of `'Swap'`, an `NINST`-by-`1` string array with values of `"Swap"`, or an `NINST`-by-`1` cell array of character vectors with values of `'Swap'`.

Data Types: `char` | `cell` | `string`

`Swap` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `SwapInstrument =`
```
fininstrument("Swap",'Maturity',datetime(2019,1,30),'LegRate',[0.06
0.12],'LegType',
["fixed","fixed"],'Basis',1,'Notional',100,'StartDate',datetime(2016,1,30),'D
aycountAdjustedCashFlow',true,'BusinessDayConvention',"follow",'ProjectionCur
ve',ratecurve,'Name',"swap_instrument")
```

Required `Swap` Name-Value Pair Arguments

`Maturity` — Swap maturity date

datetime array | string array | date character vector

Swap maturity date, specified as the comma-separated pair consisting of `'Maturity'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `Swap` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`LegRate` — Leg rate in decimal values

matrix

Leg rate in decimal values, specified as the comma-separated pair consisting of `'LegRate'` and a `NINST`-by-`2` matrix. Each row can be defined as one of the following:

- `[CouponRate Spread]` (fixed-float)

- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

`CouponRate` is the decimal annual rate. `Spread` is the number of basis points in decimals over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.

Data Types: `double`

Optional Swap Name-Value Pair Arguments

LegType — Leg type
`["fixed","float"]` for each instrument (default) | cell array of character vectors with values `{'fixed','fixed'}`, `{'fixed','float'}`, `{'float','fixed'}`, or `{'float','float'}` | string array with values `["fixed","fixed"]`, `["fixed","float"]`, `["float","fixed"]`, or `["float","float"]`

Leg type, specified as the comma-separated pair consisting of `'LegType'` and a cell array of character vectors or a string array with the supported values. The `LegType` defines the interpretation of the values entered in `LegRate`.

Note When you specify a `Swap` instrument as the underlying asset for a `Swaption` instrument while using a `Normal`, `SABR`, `Black`, or `HullWhite` pricer, the `Swap` `LegType` must be `["fixed","float"]` or `["float","fixed"]`. You must also set the `ExerciseStyle` name-value pair argument of the associated `Swaption` instrument to `'European'`.

Data Types: `cell` | `string`

ProjectionCurve — Rate curve for projecting floating cash flows
`ratecurve.empty` (default) | scalar `ratecurve` object | vector of `ratecurve` objects

Rate curve for projecting floating cash flows, specified as the comma-separated pair consisting of `'ProjectionCurve'` and a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects. You must create this object using `ratecurve`. Use this optional input if the forward curve is different from the discount curve.

Data Types: `object`

Reset — Frequency of payments per year
`[2 2]` (default) | numeric value of `0`, `1`, `2`, `3`, `4`, `6`, or `12` | matrix

Frequency of payments per year, specified as the comma-separated pair consisting of `'Reset'` and scalar or a `NINST`-by-`2` matrix if `Reset` is different for each leg) with one of the following values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

Basis — Day count basis representing the basis for each leg
`[0 0]` (actual/actual) (default) | integer from `0` to `13`

Day count basis representing the basis for each leg, specified as the comma-separated pair consisting of `'Basis'` and a `NINST`-by-`1` matrix (or `NINST`-by-`2` matrix if `Basis` is different for each leg).

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Notional` — Notional principal amount or principal value schedule
`100` (default) | scalar numeric | numeric vector | timetable

Notional principal amount or principal value schedule, specified as the comma-separated pair consisting of `'Notional'` and a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable. Use a scalar or vector for a vanilla `Swap` instrument and a timetable for an amortizing `Swap` instrument.

`Notional` accepts a scalar for a principal amount (or a `1`-by-`2` matrix if `Notional` is different for each leg) or a `timetable` for principal value schedules. For schedules, the first column of the timetable is dates and the second column is the associated notional principal value. The date indicates the last day that the principal value is valid.

Note If you are creating one or more `Swap` instruments and use a timetable, the timetable specification applies to all of the `Swap` instruments. `Notional` does not accept an `NINST`-by-`1` cell array of timetables as input.

Data Types: `timetable` | `double`

`LatestFloatingRate` — Latest floating rate for float legs
if not specified, then `ratecurve` must contain this information (default) | scalar numeric | vector

Latest floating rate for float legs, specified as the comma-separated pair consisting of `'LatestFloatingRate'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

`LatestFloatingRate` is a `NINST`-by-`1` matrix (or `NINST`-by-`2` matrix if `LatestFloatingRate` is different for each leg).

Data Types: `double`

ResetOffset — Lag in rate setting
`[0 0]` (default) | vector

Lag in rate setting, specified as the comma-separated pair consisting of `'ResetOffset'` and a `NINST`-by-2 matrix.

Data Types: `double`

DaycountAdjustedCashFlow — Flag to adjust cash flows based on actual period day count
`false` (default) | logical value of `true` or `false` | vector of logical values of `true` or `false`

Flag to adjust cash flows based on actual period day count, specified as the comma-separated pair consisting of `'DaycountAdjustedCashFlow'` and a `NINST`-by-1 matrix (or `NINST`-by-2 matrix if `AdjustCashFlowsBasis` is different for each leg) of logicals with values of `true` or `false`.

Data Types: `logical`

BusinessDayConvention — Business day conventions
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and string (or `NINST`-by-2 string array if `BusinessDayConvention` is different for each leg) or a character vector (or `NINST`-by-2 cell array of character vectors if `BusinessDayConvention` is different for each leg). The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However, if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However, if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

Holidays — Holidays used in computing business days
`NaT` (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of `'Holidays'` and dates using an `NINST`-by-1 vector of datetimes, cell array of date character vectors, or date string array. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
Swap = fininstrument("Swap",'Maturity',datetime(2025,12,15),'LegRate',[0.06 20],'Holidays',H)
```

To support existing code, `Swap` also accepts serial date numbers as inputs, but they are not recommended.

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`[true true]` (in effect) (default) | logical with value of `true` or `false` | vector of logicals with values of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, specified as the comma-separated pair consisting of `'EndMonthRule'` and a logical value of `true` or `false` using a `NINST`-by-`1` matrix (or `NINST`-by-`2` matrix if `EndMonthRule` is different for each leg).

- If you set `EndMonthRule` to `false`, the software ignores the rule, meaning that a payment date is always the same numerical day of the month.
- If you set `EndMonthRule` to `true`, the software sets the rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

`StartDate` — Date swap starts
`Settle` date (default) | datetime array | string array | date character vector

Date swap starts, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `Swap` also accepts serial date numbers as inputs, but they are not recommended.

Use `StartDate` to price a forward swap, that is, a swap that starts at a future date.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `StartDate` property is stored as a datetime.

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for the instrument, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Maturity` — Maturity date
scalar datetime | vector of datetimes

Maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`LegRate` — Leg rate
matrix

Leg rate, returned as a `NINST`-by-`2` matrix of decimal values, with each row defined as one of the following:

- `[CouponRate Spread]` (fixed-float)
- `[Spread CouponRate]` (float-fixed)
- `[CouponRate CouponRate]` (fixed-fixed)
- `[Spread Spread]` (float-float)

Data Types: `double`

`LegType` — Leg type
`["fixed","float"]` for each instrument (default) | string array with values `["fixed","fixed"]`, `["fixed","float"]`, `["float","fixed"]`, or `["float","float"]`

Leg type, returned as a string array with the values `["fixed","fixed"]`, `["fixed","float"]`, `["float","fixed"]`, or `["float","float"]`.

Data Types: `string`

`ProjectionCurve` — Rate curve used in generating future cash flows
`ratecurve.empty` (default) | scalar `ratecurve` object | vector of `ratecurve` objects

Rate curve used in projecting the future cash flows, returned as a `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects.

Data Types: `object`

`Reset` — Reset frequency per year for each swap
`[2 2]` (default) | vector

Reset frequency per year for each swap, returned as a scalar or an `NINST`-by-`2` matrix.

Data Types: `double`

`Basis` — Day count basis
`[0 0]` (actual/actual) (default) | integer from `0` to `13`

Day count basis, returned as an `NINST`-by-`1` or an `NINST`-by-`2` matrix.

Data Types: `double`

`ResetOffset` — Lag in rate setting
`[0 0]` (default) | matrix

Lag in rate setting, returned as an `NINST`-by-`2` or an `NINST`-by-`2` matrix.

Data Types: `double`

`Notional` — Notional principal amount or principal value schedules
`100` (default) | scalar numeric | numeric vector | timetable

Notional principal amount, returned as a scalar numeric or an `NINST`-by-`1` numeric vector or a timetable.

Data Types: `double` | `timetable`

`LatestFloatingRate` — Rate for the next floating payment
if not specified, then `ratecurve` must contain this information (default) | scalar numeric | numeric vector

Rate for the next floating payment, set at the last reset date, returned as a scalar numeric or an `NINST`-by-1 numeric vector or `NINST`-by-2 if `LatestFloatingRate` is different for each leg.

Data Types: `double`

`DaycountAdjustedCashFlow` — Flag to adjust cash flows based on actual period day count
`false` (default) | logical value of `true` or `false` | vector of logical values of `true` or `false`

Flag to adjust cash flows based on actual period day count, returned as an `NINST`-by-1 matrix (or an `NINST`-by-2 matrix if `AdjustCashFlowsBasis` is different for each leg) of logicals with values of `true` or `false`.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions
`"actual"` (default) | string | string array

Business day conventions, returned as a string or a `NINST`-by-2 string array if `BusinessDayConvention` is different for each leg.

Data Types: `char` | `cell` | `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-1 vector of datetimes.

Data Types: `datetime`

`EndMonthRule` — End-of-month rule flag for generating dates when `Maturity` is end-of-month date for month with 30 or fewer days
`[true true]` (in effect) (default) | logical with value of `true` or `false` | vector of logicals with values of `true` or `false`

End-of-month rule flag for generating dates when `Maturity` is an end-of-month date for a month with 30 or fewer days, returned as an `NINST`-by-1 matrix (or `NINST`-by-2 matrix if `EndMonthRule` is different for each leg.

Data Types: `logical`

`StartDate` — Date swap starts
`Settle` date (default) | scalar datetime | vector of datetimes

Date swap starts, returned as a scalar datetime or an `NINST`-by-1 vector of datetimes.

Data Types: `datetime`

`Name` — User-defined name for instrument
`"  "` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-1 string array.

Data Types: `string`

Object Functions

cashflows	Compute cash flow for FixedBond, FloatBond, Swap, FRA, STIRFuture, OISFuture, OvernightIndexedSwap, or Deposit instrument

parswaprate Compute par swap rate for Swap instrument

Examples

Price Vanilla Swap Instrument Using ratecurve and Discount Pricer

This example shows the workflow to price a vanilla Swap instrument when you use a `ratecurve` and a `Discount` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the Swap instrument.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create a vanilla Swap instrument object.

```
Swap = fininstrument("Swap",'Maturity',datetime(2024,9,15),'LegRate',[0.022 0.019 ],'LegType',["
```

```
Swap =
  Swap with properties:

                   LegRate: [0.0220 0.0190]
                   LegType: ["float"    "fixed"]
                     Reset: [2 2]
                     Basis: [0 0]
                  Notional: 100
         LatestFloatingRate: [NaN NaN]
                ResetOffset: [0 0]
   DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [1x2 ratecurve]
      BusinessDayConvention: ["actual"    "actual"]
                   Holidays: NaT
```

```
    EndMonthRule: [1 1]
       StartDate: NaT
        Maturity: 15-Sep-2024
            Name: "swap_instrument"
```

Create Discount Pricer Object

Use finpricer to create a Discount pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price Swap Instrument

Use price to compute the price and sensitivities for the vanilla Swap instrument.

```
[Price, outPR] = price(outPricer, Swap,["all"])
```

```
Price = 7.2279
```

```
outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    _________

    7.2279    -0.046631
```

Price Multiple Vanilla Swap Instruments Using ratecurve and Discount Pricer

This example shows the workflow to price multiple vanilla Swap instruments when you use a ratecurve and a Discount pricing method.

Create ratecurve Object

Create a ratecurve object using ratecurve for the underlying interest-rate curve for the Swap instrument.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
```

```
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)

myRC =
  ratecurve with properties:

                 Type: "zero"
          Compounding: -1
                Basis: 0
                Dates: [10x1 datetime]
                Rates: [10x1 double]
               Settle: 15-Sep-2019
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create a vanilla `Swap` instrument object for three Swap instruments.

```
Swap = fininstrument("Swap",'Maturity',datetime([2024,9,15 ; 2025,9,15 ; 2026,9,15]),'LegRate',[0

Swap=3×1 object
  3x1 Swap array with properties:

    LegRate
    LegType
    Reset
    Basis
    Notional
    LatestFloatingRate
    ResetOffset
    DaycountAdjustedCashFlow
    ProjectionCurve
    BusinessDayConvention
    Holidays
    EndMonthRule
    StartDate
    Maturity
    Name
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)

outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price Swap Instruments

Use `price` to compute the prices and sensitivities for the vanilla `Swap` instruments.

```
[Price, outPR] = price(outPricer, Swap,["all"])
```

```
Price = 3×1

    7.2279
    9.9725
   13.0798


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×2 table
    Price       DV01
    ______    _________

    7.2279    -0.046631


ans=1×2 table
    Price       DV01
    ______    _________

    9.9725    -0.054393


ans=1×2 table
    Price      DV01
    _____    _________

    13.08    -0.061381
```

Price Amortizing Swap Instrument Using ratecurve and Discount Pricer

This example shows the workflow to price an amortizing `Swap` instrument when you use a `ratecurve` and a `Discount` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `Swap` instrument.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
```

```
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)

myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create an amortizing `Swap` instrument object.

```
Maturity = datetime(2024,1,1);
ADates = datetime([2020,1,1 ; 2024,1,1]);
APrincipal = [100; 85];
Notional = timetable(ADates,APrincipal);
Swap = fininstrument("Swap",'Maturity',Maturity,'LegRate',[0.035,0.01],'Reset',[1 1],'Notional',N

Swap =
  Swap with properties:

                   LegRate: [0.0350 0.0100]
                   LegType: ["fixed"    "float"]
                     Reset: [1 1]
                     Basis: [0 0]
                  Notional: [2x1 timetable]
         LatestFloatingRate: [NaN NaN]
                ResetOffset: [0 0]
   DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [0x0 ratecurve]
      BusinessDayConvention: ["actual"    "actual"]
                   Holidays: NaT
              EndMonthRule: [1 1]
                 StartDate: NaT
                  Maturity: 01-Jan-2024
                      Name: "swap_instrument"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)

outPricer =
  Discount with properties:
```

```
    DiscountCurve: [1x1 ratecurve]
```

Price Swap Instrument

Use `price` to compute the price and sensitivities for the amortizing `Swap` instrument.

```
[Price, outPR] = price(outPricer, Swap,["all"])

Price = 5.7183

outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []


outPR.Results

ans=1×2 table
    Price       DV01
    ______    ________

    5.7183    0.044672
```

Price Vanilla Swap Instrument Using Hull-White Model and IRTree Pricer

This example shows the workflow to price a vanilla `Swap` instrument when you use a `HullWhite` model and an `IRTree` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `Swap` instrument.

```
Settle = datetime(2020,1,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)

myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: [10x1 datetime]
             Rates: [10x1 double]
            Settle: 15-Jan-2020
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
```

```
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create a vanilla `Swap` instrument object.

```
LegType = ["float","fixed"];
Swap = fininstrument("Swap",'Maturity',datetime(2030,9,15),'LegRate',[0.022 0.019],'LegType',Leg
```

```
Swap =
  Swap with properties:

                     LegRate: [0.0220 0.0190]
                     LegType: ["float"    "fixed"]
                       Reset: [2 2]
                       Basis: [0 0]
                    Notional: 100
           LatestFloatingRate: [NaN NaN]
                  ResetOffset: [0 0]
    DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [1x2 ratecurve]
       BusinessDayConvention: ["actual"    "actual"]
                    Holidays: NaT
                EndMonthRule: [1 1]
                   StartDate: NaT
                    Maturity: 15-Sep-2030
                        Name: "swap_instrument"
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.032,'Sigma',0.04)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0320
    Sigma: 0.0400
```

Compute Swap Instrument Cash Flow Dates

Use `cfdates` to compute the cash flows.

```
CFdates = cfdates(Settle, Swap.Maturity, Swap.Reset(1), Swap.Basis(1))
```

```
CFdates = 1x22 datetime
Columns 1 through 5

   15-Mar-2020   15-Sep-2020   15-Mar-2021   15-Sep-2021   15-Mar-2022

Columns 6 through 10

   15-Sep-2022   15-Mar-2023   15-Sep-2023   15-Mar-2024   15-Sep-2024

Columns 11 through 15
```

```
   15-Mar-2025   15-Sep-2025   15-Mar-2026   15-Sep-2026   15-Mar-2027

Columns 16 through 20

   15-Sep-2027   15-Mar-2028   15-Sep-2028   15-Mar-2029   15-Sep-2029

Columns 21 through 22

   15-Mar-2030   15-Sep-2030
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HullWhiteModel,'DiscountCurve',myRC,'TreeDates',CFdates
```

```
HWTreePricer =
  HWBKTree with properties:

             Tree: [1x1 struct]
        TreeDates: [22x1 datetime]
            Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

Price Swap Instrument

Use `price` to compute the price and sensitivities for the vanilla `Swap` instrument.

```
[Price, outPR] = price(HWTreePricer, Swap,"all")
```

```
Price = 24.3727

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta      Gamma        Vega
    ______    ______    _______    __________

    24.373    820.67    -8790.5    8.5265e-10
```

Price Vanilla Swap Instrument Using LinearGaussian2F Model and IRMonteCarlo Pricer

This example shows the workflow to price a vanilla `Swap` instrument when using a `LinearGaussian2F` model and an `IRMonteCarlo` pricing method.

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2020,1,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2020
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use fininstrument to create a Swap instrument object.

```
LegType = ["float","fixed"];
Swap = fininstrument("Swap",'Maturity',datetime(2030,9,15),'LegRate',[0.022 0.019],'LegType',Leg
```

```
Swap =
  Swap with properties:

                   LegRate: [0.0220 0.0190]
                   LegType: ["float"    "fixed"]
                     Reset: [2 2]
                     Basis: [0 0]
                  Notional: 100
         LatestFloatingRate: [NaN NaN]
                ResetOffset: [0 0]
   DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [1x2 ratecurve]
      BusinessDayConvention: ["actual"    "actual"]
                  Holidays: NaT
              EndMonthRule: [1 1]
                 StartDate: NaT
                  Maturity: 15-Sep-2030
                      Name: "swap_instrument"
```

Create LinearGaussian2F Model Object

Use finmodel to create a LinearGaussian2F model object.

```
LinearGaussian2FModel = finmodel("LinearGaussian2F",'Alpha1',0.07,'Sigma1',0.01,'Alpha2',0.5,'Sig
```

```
LinearGaussian2FModel =
  LinearGaussian2F with properties:

        Alpha1: 0.0700
        Sigma1: 0.0100
        Alpha2: 0.5000
        Sigma2: 0.0060
    Correlation: -0.7000
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",'Model',LinearGaussian2FModel,'DiscountCurve',myRC,'Simulati
```

```
outPricer =
  G2PPMonteCarlo with properties:

         NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [15-Jul-2020    15-Jan-2021    15-Jan-2022    ...    ]
              Model: [1x1 finmodel.LinearGaussian2F]
```

Price Swap Instrument

Use `price` to compute the price and sensitivities for the `Swap` instrument.

```
[Price,outPR] = price(outPricer,Swap,["all"])
```

```
Price = 23.6657
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta      Gamma      Vega
    ______    ______    _______    ______

    23.666    819.11    -8748.9    0    0
```

More About

Swap

A swap is contract between two parties obligating the parties to exchange future cash flows.

A vanilla swap is composed of a floating-rate leg and a fixed-rate leg.

Swap with Amortization

A swap with an amortization schedule repays part of the principal (face value) along with the coupon payments.

A swap with an amortization schedule is used to manage interest-rate risk and serve as a cash flow management tool. For this particular type of swap, the notional amount decreases over time. This means that interest payments decrease not only on the floating leg but also on the fixed leg. Use the `Notional` name-value pair argument to support an amortization schedule.

Forward Swap

In a forward interest-rate swap, a fixed interest-rate loan is exchanged for a floating interest-rate loan at a future specified date.

The `StartDate` name-value pair argument supports the future date for the forward swap.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Swap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`Swaption` | `finmodel` | `finpricer` | `timetable`

Topics

“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Calibrate Shifted SABR Model Parameters for Swaption Instrument” on page 2-167
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

VarianceSwap

`VarianceSwap` instrument object

Description

Create and price a `VarianceSwap` instrument object for one or more Variance Swap instruments using this workflow:

1. Use `fininstrument` to create a `VarianceSwap` instrument object for one or more Variance Swap instruments.
2. Use `ratecurve` to specify a curve model or use `finmodel` to specify a `Heston` model.
3. Choose a pricing method.
   - When using a curve model, use `finpricer` to specify a `ReplicatingVarianceSwap` pricing method for one or more `VarianceSwap` instruments.
   - When using a `Heston` model, use `finpricer` to specify a `Heston` pricing method for one or more `VarianceSwap` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `VarianceSwap` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
VarianceSwapInstrument = fininstrument(InstrumentType,'
Maturity',maturity_date,'Notional',notional_value)
VarianceSwapInstrument = fininstrument( ___ ,Name,Value)
```

Description

`VarianceSwapInstrument = fininstrument(InstrumentType,'Maturity',maturity_date,'Notional',notional_value)` creates a `VarianceSwap` object for one or more Variance Swap instruments by specifying `InstrumentType` and sets properties on page 11-2937 using the required name-value pair arguments `Maturity` and `Notional`.

The `VarianceSwap` instrument supports the `ReplicatingVarianceSwap` and `Heston` pricing methods. For more information on the `VarianceSwap` instrument, see “More About” on page 11-2943.

`VarianceSwapInstrument = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2921 using additional name-value pair arguments in addition to the required arguments in the previous syntax. For example, `VarianceSwapInstrument = fininstrument("VarianceSwap",'Maturity',datetime(2019,1,30),'Notional',100,'S`

tartDate',datetime(2016,1,30),'RealizedVariance',0.02,'Strike',110,'Name',"varianceswap_instrument") creates a VarianceSwap option with a maturity date of January 30, 2019. You can specify multiple name-value pair arguments.

Input Arguments

InstrumentType — Instrument type

string with value "VarianceSwap" | string array with values of "VarianceSwap" | character vector with value 'VarianceSwap' | cell array of character vectors with values of 'VarianceSwap'

Instrument type, specified as a string with the value of "VarianceSwap", a character vector with the value of 'VarianceSwap', an NINST-by-1 string array with values of "VarianceSwap", or an NINST-by-1 cell array of character vectors with values of 'VarianceSwap'.

Data Types: char | cell | string

VarianceSwap Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: VarianceSwapInstrument = fininstrument("VarianceSwap",'Maturity',datetime(2019,1,30),'Notional',100,'StartDate',datetime(2016,1,30),'RealizedVariance',0.02,'Strike',110,'Name',"varianceswap_instrument")

Required VarianceSwap Name-Value Pair Arguments

Maturity — Variance swap maturity date

datetime array | string array | date character vector

Variance swap maturity date, specified as the comma-separated pair consisting of 'Maturity' and a scalar or an NINST-by-1 vector.

If you use date character vectors or strings, the format must be recognizable by datetime because the Maturity property is stored as a datetime.

To support existing code, VarianceSwap also accepts serial date numbers as inputs, but they are not recommended.

Notional — Notional amount

scalar numeric | numeric vector

Notional amount, specified as the comma-separated pair consisting of 'Notional' and a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

Optional VarianceSwap Name-Value Pair Arguments

StartDate — Start date

Settle date of the ratecurve (default) | datetime array | string array | date character vector

Start date, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `VarianceSwap` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `ExerciseDate` property is stored as a datetime.

`RealizedVariance` — Realized variance
`0` (default) | scalar decimal | decimal vector

Realized variance, specified as the comma-separated pair consisting of `'RealizedVariance'` and a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Strike` — Strike variance
fair variance computed by pricer (default) | scalar decimal | decimal vector

Strike variance, specified as the comma-separated pair consisting of `'Strike'` and a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Maturity` — Maturity date
datetime | vector of datetimes

Maturity date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Notional` — Notional amount
scalar numeric | numeric vector

Notional amount, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`StartDate` — Start date
`Settle` date of the `ratecurve` (default) | datetime | vector of datetimes

Start date, returned as a scalar datetime or an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`RealizedVariance` — Realized variance
`0` (default) | scalar decimal | decimal vector

Realized variance, returned as a scalar decimal or an `NINST`-by-`1` decimal vector.

Data Types: `double`

`Strike` — Strike variance
fair variance computed by pricer (default) | scalar decimal | decimal vector

Strike variance, returned as a scalar decimal or an `NINST`-by-`1` decimal vector.

Data Types: `double`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

Examples

Price Variance Swap Instrument Using Heston Model and Heston Pricer

This example shows the workflow to price a `VarianceSwap` instrument when you use a `Heston` model and a `Heston` pricing method.

Create `VarianceSwap` Instrument Object

Use `fininstrument` to create a `VarianceSwap` instrument object.

```
VarianceSwapInst = fininstrument("VarianceSwap",'Maturity',datetime(2020,9,15),'Notional',100,'St
```

```
VarianceSwapInst =
  VarianceSwap with properties:

            Notional: 100
    RealizedVariance: 0.0200
              Strike: 0.0500
           StartDate: 15-Jun-2020
            Maturity: 15-Sep-2020
                Name: "variance_swap_instrument"
```

Create `Heston` Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.06,'ThetaV',0.1,'Kappa',0.9,'SigmaV',0.7,'RhoSV',-.3)
```

```
HestonModel =
  Heston with properties:
```

```
        V0: 0.0600
    ThetaV: 0.1000
     Kappa: 0.9000
    SigmaV: 0.7000
     RhoSV: -0.3000
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020, 1, 1);
ZeroTimes = calmonths(3);
ZeroRates = 0.05;
ZeroDates = Settle + ZeroTimes;
Basis = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Basis',Basis)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Apr-2020
               Rates: 0.0500
              Settle: 01-Jan-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Heston` Pricer Object

Use `finpricer` to create a `Heston` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Analytic",'DiscountCurve',ZeroCurve,'Model',HestonModel)
```

```
outPricer =
  Heston with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.Heston]
```

Price `VarianceSwap` Instrument

Use `price` to compute the price and fair variance for the `VarianceSwap` instrument.

```
[Price, outPR] = price(outPricer,VarianceSwapInst,["all"])
```

```
Price = 10.8321

outPR =
  priceresult with properties:

       Results: [1x2 table]
```

```
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
    Price     FairVariance
    ______    ____________

    10.832      0.07039
```

Price Multiple Variance Swap Instruments Using Heston Model and Heston Pricer

This example shows the workflow to price multiple `VarianceSwap` instrument when you use a `Heston` model and a `Heston` pricing method.

Create `VarianceSwap` Instrument Object

Use `fininstrument` to create a `VarianceSwap` instrument object for three Variance Swap instruments.

```
VarianceSwapInst = fininstrument("VarianceSwap",'Maturity',datetime([2020,9,15 ; 2020,9,15 ; 2020
```

```
VarianceSwapInst=3×1 object
  3x1 VarianceSwap array with properties:

    Notional
    RealizedVariance
    Strike
    StartDate
    Maturity
    Name
```

Create `Heston` Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.06,'ThetaV',0.1,'Kappa',0.9,'SigmaV',0.7,'RhoSV',-.3)
```

```
HestonModel =
  Heston with properties:

        V0: 0.0600
    ThetaV: 0.1000
     Kappa: 0.9000
    SigmaV: 0.7000
     RhoSV: -0.3000
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020, 1, 1);
ZeroTimes = calmonths(3);
```

```
ZeroRates = 0.05;
ZeroDates = Settle + ZeroTimes;
Basis = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Basis',Basis)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Apr-2020
               Rates: 0.0500
              Settle: 01-Jan-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Heston` Pricer Object

Use `finpricer` to create a `Heston` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Analytic",'DiscountCurve',ZeroCurve,'Model',HestonModel)
```

```
outPricer =
  Heston with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.Heston]
```

Price `VarianceSwap` Instruments

Use `price` to compute the prices and fair variance for the three `VarianceSwap` instruments.

```
% [Price, outPR] = price(outPricer,VarianceSwapInst,["all"])
% outPR.Results
```

Price Variance Swap Instrument Using `ratecurve` and Replicating Variance Swap Pricer

This example shows the workflow to price a `VarianceSwap` instrument when you use a `ratecurve` object and a `ReplicatingVarianceSwap` pricing method.

Create `VarianceSwap` Instrument Object

Use `fininstrument` to create a `VarianceSwap` instrument object.

```
VarianceSwapInst = fininstrument("VarianceSwap",'Maturity',datetime(2021,5,1),'Notional',150,'Sta
```

```
VarianceSwapInst =
  VarianceSwap with properties:

            Notional: 150
    RealizedVariance: 0.0500
```

```
            Strike: 0.1000
         StartDate: 01-May-2020
          Maturity: 01-May-2021
              Name: "variance_swap_instrument"
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020, 9, 15);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Basis = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Basis',Basis)
```

```
ZeroCurve =
  ratecurve with properties:

               Type: "zero"
       Compounding: -1
             Basis: 1
             Dates: [10x1 datetime]
             Rates: [10x1 double]
            Settle: 15-Sep-2020
      InterpMethod: "linear"
 ShortExtrapMethod: "next"
  LongExtrapMethod: "previous"
```

Create ReplicatingVarianceSwap Pricer Object

Use `finpricer` to create a `ReplicatingVarianceSwap` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
Strike = (50:5:135)';
Volatility = [.49;.45;.42;.38;.34;.31;.28;.25;.23;.21;.2;.21;.21;.22;.23;.24;.25;.26];
VolatilitySmile = table(Strike, Volatility);
SpotPrice = 100;
CallPutBoundary = 100;

outPricer =  finpricer("ReplicatingVarianceSwap",'DiscountCurve', ZeroCurve, 'VolatilitySmile', V
'SpotPrice', SpotPrice, 'CallPutBoundary', CallPutBoundary)
```

```
outPricer =
  ReplicatingVarianceSwap with properties:

    DiscountCurve: [1x1 ratecurve]
     InterpMethod: "linear"
 VolatilitySmile: [18x2 table]
       SpotPrice: 100
 CallPutBoundary: 100
```

Price VarianceSwap Instrument

Use `price` to compute the price and fair variance for the `VarianceSwap` instrument.

```
[Price, outPR] = price(outPricer,VarianceSwapInst,["all"])

Price = 8.1997

outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×2 table
    Price     FairVariance
    ______    ____________

    8.1997      0.21701
```

More About

Variance Swap

A variance swap is a forward contract on the realized variance over time.

At expiration, the payoff of a variance swap is $(\sigma_R^2 - K_{\text{var}}) \times \text{N}$

Here:

σ^2_R is the realized variance of the underlying asset (for example a stock or equity index) over the life of the contract.

K_{var} is the strike (contractual) variance that is set at the beginning of the contract to make the starting value equal to zero.

N is the notional amount.

Version History

Introduced in R2020b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `VarianceSwap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`finmodel` | `finpricer`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Swaption

`Swaption` instrument object

Description

Create and price a `Swaption` instrument object for one or more Swaption instruments using this workflow:

1. Use `fininstrument` to create a `Swaption` instrument object for one or more Swaption instruments.
2. Use `finmodel` to specify a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `Black`, `Normal`, `SABR`, or `LinearGaussian2F` model for the `Swaption` instrument object.
3. Choose a pricing method.
   - When using a `HullWhite`, `BlackKarasinski`, `BlackDermanToy`, `Black`, `Normal`, or `SABR` model, use `finpricer` for pricing one or more `Swaption` instruments and specify:
     - A `Normal` pricer when using a `Normal` model.
     - A `Black` pricer when using a `Black` model.
     - A `HullWhite` pricer when using a `HullWhite` model.
     - A `SABR` pricer when using a `SABR` model.
     - An `IRTree` pricer when using a `BlackKarasinski` or `BlackDermanToy` model.
   - When using a `HullWhite`, `BlackKarasinski`, or `LinearGaussian2F` model, use `finpricer` to specify an `IRMonteCarlo` pricing method for one or more `Swaption` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Swaption` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
SwaptionInstrument = fininstrument(InstrumentType,'Strike',strike_value,'
ExerciseDate',exercice_date)
SwaptionInstrument = fininstrument( ___ ,Name,Value)
```

Description

`SwaptionInstrument = fininstrument(InstrumentType,'Strike',strike_value,'ExerciseDate',exercice_date)` creates a `Swaption` object for one or more Swaption instruments by specifying `InstrumentType` and sets the properties on page 11-2947 for the

required name-value pair arguments `Strike` and `ExerciseDate`. For more information on a `Swaption` instrument, see “More About” on page 11-2963.

`SwaptionInstrument = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2947 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `SwaptionInstrument = fininstrument("Swaption",'Strike',0.67,'ExerciseDate',datetime(2019,1,30),'Swap',Swap_obj,'OptionType',"put",'ExerciseStyle',"European",'Name',"swaption_instrument")` creates a `Swaption` put instrument with a strike of 0.67 and an European exercise. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Swaption"` | string array with values of `"Swaption"` | character vector with value `'Swaption'` | cell array of character vectors with values of `'Swaption'`

Instrument type, specified as a string with the value of `"Swaption"`, a character vector with the value of `'Swaption'`, an `NINST`-by-`1` string array with values of `"Swaption"`, or an `NINST`-by-`1` cell array of character vectors with values of `'Swaption'`.

Data Types: `char` | `cell` | `string`

`Swaption` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `SwaptionInstrument = fininstrument("Swaption",'Strike',.67,'ExerciseDate',datetime(2019,1,30),'Swap',Swap_obj,'OptionType',"put",'ExerciseStyle',"European",'Name',"swaption_instrument")`

Required Swaption Name-Value Pair Arguments

`Strike` — Option strike value

scalar nonnegative decimal | vector of nonnegative decimals

Option strike value, specified as the comma-separated pair consisting of `'Strike'` and a scalar nonnegative decimal or an `NINST`-by-`1` vector of nonnegative decimals.

Data Types: `double`

`ExerciseDate` — Option exercise date

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of `'ExerciseDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array,or date character vectors.

Note For a European option, there is only one `ExerciseDate` on the option expiry date.

To support existing code, `Swaption` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `ExerciseDate` property is stored as a datetime.

`Swap` — Underlying `Swap` object
scalar `Swap` object | vector of `Swap` objects

Underlying `Swap` object, specified as the comma-separated pair consisting of `'Swap'` and a scalar `Swap` object or an `NINST`-by-`1` vector of `Swap` objects.

Data Types: `object`

Optional Swaption Name-Value Pair Arguments

`OptionType` — Option type
`"call"` (default) | string with values `"call"` or `"put"` | string array with values of `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values of `'call'` or `'put'`

Option type, specified as the comma-separated pair consisting of `'OptionType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"` | string array with values of `"European"` | character vector with value `'European'` | cell array of character vectors with values of `"European"`

Option exercise style, specified as the comma-separated pair consisting of `'ExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Note When you specify a `Swap` instrument as the underlying asset for a `Swaption` instrument and use a `Normal`, `SABR`, `Black`, or `HullWhite` pricer, the `Swap` instrument `LegType` must be `["fixed","float"]` or `["float","fixed"]` and the `Swaption` instrument `ExerciseStyle` must be `"European"`.

Data Types: `string` | `cell` | `char`

`Name` — User-defined name for instrument
`" "` (default) | string | string array | character vector | cell array of character vectors

User-defined name for the instrument, specified as the comma-separated pair consisting of `'Name'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Strike` — Option strike value
scalar nonnegative decimal | vector of nonnegative decimals

Option strike value, returned as a scalar nonnegative decimal or an NINST-by-1 vector of nonnegative decimals.

Data Types: double

ExerciseDate — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: datetime

Swap — Underlying Swap object
scalar Swap object | vector of Swap objects

Swap object, returned as a scalar Swap object or an NINST-by-1 vector of Swap objects.

Data Types: object

OptionType — Option type
"call" (default) | string with value "call" or "put" | string array with values of "call" or "put"

Option type, returned as a scalar string or an NINST-by-1 string array with a value of "call" or "put".

Data Types: string

ExerciseStyle — Option exercise style
"European" (default) | string with value "European" | string array with values of "European"

Option exercise style, returned as a scalar string or an NINST-by-1 string array with a value of "European".

Data Types: string

Name — User-defined name for instrument
" " (default) | string | string array

User-defined name for the instrument, returned as a string or an NINST-by-1 string array.

Data Types: string

Examples

Price Swaption Instrument Using SABR Model and SABR Pricer

This example shows the workflow to price a Swaption instrument when you use a SABR model and a SABR pricing method.

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
```

```
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create the underlying `Swap` instrument object.

```
Swap = fininstrument("Swap",'Maturity',datetime(2027,3,15),'LegRate',[0 0],'LegType',...
    ["float","fixed"],'Notional',100,'StartDate',datetime(2022,3,15),'Name',"swap_instrument")
```

```
Swap =
  Swap with properties:

                   LegRate: [0 0]
                   LegType: ["float"    "fixed"]
                     Reset: [2 2]
                     Basis: [0 0]
                  Notional: 100
         LatestFloatingRate: [NaN NaN]
                ResetOffset: [0 0]
   DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [0x0 ratecurve]
      BusinessDayConvention: ["actual"    "actual"]
                   Holidays: NaT
               EndMonthRule: [1 1]
                  StartDate: 15-Mar-2022
                   Maturity: 15-Mar-2027
                       Name: "swap_instrument"
```

Create Swaption Instrument Object

Use `fininstrument` to create a `Swaption` instrument object.

```
Swaption = fininstrument("Swaption",'Strike',0.02,'ExerciseDate',datetime(2022,3,15),'Swap',Swap
```

```
Swaption =
  Swaption with properties:

      OptionType: "call"
   ExerciseStyle: "european"
    ExerciseDate: 15-Mar-2022
          Strike: 0.0200
            Swap: [1x1 fininstrument.Swap]
```

```
              Name: "swaption_option"
```

Create SABR Model Object

Use `finmodel` to create a SABR model object.

```
SabrModel = finmodel("SABR",'Alpha',0.032,'Beta',0.04,'Rho',.08,'Nu',0.49,'Shift',0.002)
```

```
SabrModel =
  SABR with properties:

             Alpha: 0.0320
              Beta: 0.0400
               Rho: 0.0800
                Nu: 0.4900
             Shift: 0.0020
    VolatilityType: "black"
```

Create SABR Pricer Object

Use `finpricer` to create a SABR pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',SabrModel,'DiscountCurve',myRC)
```

```
outPricer =
  SABR with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.SABR]
```

Price `Swaption` Instrument

Use `price` to compute the price for the `Swaption` instrument.

```
Price = price(outPricer,Swaption)
```

```
Price = 10.8558
```

Price Multiple Swaption Instruments Using SABR Model and SABR Pricer

This example shows the workflow to price multiple `Swaption` instrument when you use a SABR model and a SABR pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create the underlying `Swap` instrument object.

```
Swap = fininstrument("Swap",'Maturity',datetime(2027,3,15),'LegRate',[0 0],'LegType',...
    ["float","fixed"],'Notional',100,'StartDate',datetime(2022,3,15),'Name',"swap_instrument")
```

```
Swap =
  Swap with properties:

                    LegRate: [0 0]
                    LegType: ["float"    "fixed"]
                      Reset: [2 2]
                      Basis: [0 0]
                   Notional: 100
          LatestFloatingRate: [NaN NaN]
                 ResetOffset: [0 0]
    DaycountAdjustedCashFlow: [0 0]
              ProjectionCurve: [0x0 ratecurve]
       BusinessDayConvention: ["actual"    "actual"]
                    Holidays: NaT
                EndMonthRule: [1 1]
                   StartDate: 15-Mar-2022
                    Maturity: 15-Mar-2027
                        Name: "swap_instrument"
```

Create Swaption Instrument Object

Use `fininstrument` to create a `Swaption` instrument object for three Swaption instruments.

```
Swaption = fininstrument("Swaption",'Strike',[0.02 ; 0.03 ; 0.04],'ExerciseDate',datetime([2022,3
```

```
Swaption=3×1 object
  3x1 Swaption array with properties:

    OptionType
    ExerciseStyle
    ExerciseDate
    Strike
    Swap
    Name
```

Create SABR Model Object

Use `finmodel` to create a SABR model object.

```
SabrModel = finmodel("SABR",'Alpha',0.032,'Beta',0.04,'Rho',.08,'Nu',0.49,'Shift',0.002)
```

```
SabrModel =
  SABR with properties:

             Alpha: 0.0320
              Beta: 0.0400
               Rho: 0.0800
                Nu: 0.4900
             Shift: 0.0020
    VolatilityType: "black"
```

Create SABR Pricer Object

Use `finpricer` to create a SABR pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',SabrModel,'DiscountCurve',myRC)
```

```
outPricer =
  SABR with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.SABR]
```

Price Swaption Instruments

Use `price` to compute the prices for the `Swaption` instruments.

```
Price = price(outPricer,Swaption)
```

```
Price = 3×1

   10.8558
    9.0442
    7.4883
```

Price Swaption Instrument Using Black Model and Black Pricer

This example shows the workflow to price a `Swaption` instrument when you use a `Black` model and a `Black` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
```

```
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                 Type: "zero"
          Compounding: -1
                Basis: 0
                Dates: [10x1 datetime]
                Rates: [10x1 double]
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create the underlying `Swap` instrument object.

```
Swap = fininstrument("Swap",'Maturity',datetime(2027,3,15),'LegRate',[0 0],'LegType',...
    ["float","fixed"],'Notional',100,'StartDate',datetime(2022,3,15),'Name',"swap_instrument")
```

```
Swap =
  Swap with properties:

                    LegRate: [0 0]
                    LegType: ["float"    "fixed"]
                      Reset: [2 2]
                      Basis: [0 0]
                   Notional: 100
          LatestFloatingRate: [NaN NaN]
                ResetOffset: [0 0]
    DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [0x0 ratecurve]
       BusinessDayConvention: ["actual"    "actual"]
                    Holidays: NaT
                EndMonthRule: [1 1]
                   StartDate: 15-Mar-2022
                    Maturity: 15-Mar-2027
                        Name: "swap_instrument"
```

Create Swaption Instrument Object

Use `fininstrument` to create a `Swaption` instrument object.

```
Swaption = fininstrument("Swaption",'Strike',0.02,'ExerciseDate',datetime(2022,3,15),'Swap',Swap
```

```
Swaption =
  Swaption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Mar-2022
           Strike: 0.0200
             Swap: [1x1 fininstrument.Swap]
```

```
              Name: "swaption_option"
```

Create Black Model Object

Use `finmodel` to create a `Black` model object.

```
BlackModel = finmodel("Black",'Volatility',0.032,'Shift',0.002)
```

```
BlackModel =
  Black with properties:

    Volatility: 0.0320
         Shift: 0.0020
```

Create Black Pricer Object

Use `finpricer` to create a `Black` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackModel,'DiscountCurve',myRC)
```

```
outPricer =
  Black with properties:

            Model: [1x1 finmodel.Black]
    DiscountCurve: [1x1 ratecurve]
```

Price Swaption Instrument

Use `price` to compute the price for the `Swaption` instrument.

```
Price = price(outPricer,Swaption)
```

```
Price = 3.3116
```

Price Swaption Instrument Using Hull-White Model and IRTree Pricer

This example shows the workflow to price a `Swaption` instrument when you use a `HullWhite` model and an `IRTree` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:
```

```
                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create the underlying `Swap` instrument object.

```
Swap = fininstrument("Swap",'Maturity',datetime(2027,3,15),'LegRate',[0 0],'LegType',...
    ["float","fixed"],'Notional',100,'StartDate',datetime(2022,3,15),'Name',"swap_instrument")
```

```
Swap =
  Swap with properties:

                    LegRate: [0 0]
                    LegType: ["float"    "fixed"]
                      Reset: [2 2]
                      Basis: [0 0]
                   Notional: 100
         LatestFloatingRate: [NaN NaN]
                ResetOffset: [0 0]
    DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [0x0 ratecurve]
       BusinessDayConvention: ["actual"    "actual"]
                   Holidays: NaT
               EndMonthRule: [1 1]
                  StartDate: 15-Mar-2022
                   Maturity: 15-Mar-2027
                       Name: "swap_instrument"
```

Create Swaption Instrument Object

Use `fininstrument` to create a `Swaption` instrument object.

```
Swaption = fininstrument("Swaption",'Strike',0.02,'ExerciseDate',datetime(2022,3,15),'Swap',Swap
```

```
Swaption =
  Swaption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Mar-2022
           Strike: 0.0200
             Swap: [1x1 fininstrument.Swap]
             Name: "swaption_option"
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.032,'Sigma',0.04)

HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0320
    Sigma: 0.0400
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRTree",'Model',HullWhiteModel,'DiscountCurve',myRC,'TreeDates',ZeroDates)

outPricer =
  HWBKTree with properties:

             Tree: [1x1 struct]
        TreeDates: [10x1 datetime]
            Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

Price Swaption Instrument

Use `price` to compute the price and sensitivities for the `Swaption` instrument.

```
[Price, outPR] = price(outPricer,Swaption,["all"])

Price = 14.6783

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×4 table
    Price     Delta      Gamma     Vega
    ______    ______    _______    _____

    14.678    142.29    -2263.2    321.8
```

Price Swaption Instrument Using LinearGaussian2F Model and IRMonteCarlo Pricer

This example shows the workflow to price a `Swaption` instrument when using a `LinearGaussian2F` model and an `IRMonteCarlo` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create the underlying `Swap` instrument object.

```
Swap = fininstrument("Swap",'Maturity',datetime(2022,1,1),'LegRate',[0.05,0.04],'Name',"swap_inst
```

```
Swap =
  Swap with properties:

                    LegRate: [0.0500 0.0400]
                    LegType: ["fixed"    "float"]
                      Reset: [2 2]
                      Basis: [0 0]
                   Notional: 100
          LatestFloatingRate: [NaN NaN]
                 ResetOffset: [0 0]
    DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [0x0 ratecurve]
       BusinessDayConvention: ["actual"    "actual"]
                   Holidays: NaT
                EndMonthRule: [1 1]
                   StartDate: NaT
                    Maturity: 01-Jan-2022
                        Name: "swap_instrument"
```

Create Swaption Instrument Object

Use `fininstrument` to create a `Swaption` instrument object.

```
Swaption = fininstrument("Swaption",'Strike',0.02,'ExerciseDate',datetime(2021,7,1),'Swap',Swap,
```

```
Swaption =
  Swaption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
```

```
     ExerciseDate: 01-Jul-2021
           Strike: 0.0200
             Swap: [1x1 fininstrument.Swap]
             Name: "swaption_option"
```

Create LinearGaussian2F Model Object

Use `finmodel` to create a `LinearGaussian2F` model object.

```
LinearGaussian2FModel = finmodel("LinearGaussian2F",'Alpha1',0.07,'Sigma1',0.01,'Alpha2',0.5,'Sig
```

```
LinearGaussian2FModel =
  LinearGaussian2F with properties:

         Alpha1: 0.0700
         Sigma1: 0.0100
         Alpha2: 0.5000
         Sigma2: 0.0060
    Correlation: -0.7000
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
simDates = datetime(2019,7,1)+calmonths(0:6:30);
outPricer = finpricer("IRMonteCarlo",'Model',LinearGaussian2FModel,'DiscountCurve',myRC,'Simulati
```

```
outPricer =
  G2PPMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jul-2020    ...    ]
              Model: [1x1 finmodel.LinearGaussian2F]
```

Price Swaption Instrument

Use `price` to compute the price and sensitivities for the `Swaption` instrument.

```
[Price,outPR] = price(outPricer,Swaption,["all"])
```

```
Price = 1.5065

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta     Gamma           Vega
```

```
    ______    ______    ______    _________________

    1.5065    44.746    -257.2    1.6729    -2.0015
```

Calibrate Shifted SABR Model Parameters for Swaption Instrument

Calibrate model parameters for a `Swaption` instrument when you use a SABR pricing method.

Load Market Data

```
% Zero curve
ValuationDate = datetime("5-Mar-2016", 'Locale', 'en_US');
ZeroDates = datemnth(ValuationDate,[1 2 3 6 9 12*[1 2 3 4 5 6 7 8 9 10 12]])';
ZeroRates = [-0.33 -0.28 -0.24 -0.12 -0.08 -0.03 0.015 0.028 ...
    0.033 0.042 0.056 0.095 0.194 0.299 0.415 0.525]'/100;
Compounding = 1;
ZeroCurve = ratecurve("zero",ValuationDate,ZeroDates,ZeroRates,'Compounding',Compounding)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: 1
               Basis: 0
               Dates: [16x1 datetime]
               Rates: [16x1 double]
              Settle: 05-Mar-2016
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

```
% Define the swaptions
SwaptionSettle = datetime("5-Mar-2016", 'Locale', 'en_US');
SwaptionExerciseDate = datetime("5-Mar-2017", 'Locale', 'en_US');
SwaptionStrikes = (-0.6:0.01:1.6)'/100; % Include negative strikes
SwapMaturity = datetime("5-Mar-2022", 'Locale', 'en_US'); % Maturity of underlying swap
OptSpec = 'call';
```

Compute Forward Swap Rate by Creating Swap Instrument

Use `fininstrument` to create a `Swap` instrument object.

```
LegRate = [0 0];
Swap = fininstrument("Swap", 'Maturity', SwapMaturity, 'LegRate', LegRate, "LegType",["fixed" "f
    "ProjectionCurve", ZeroCurve, "StartDate", SwaptionExerciseDate)
```

```
Swap =
  Swap with properties:

                   LegRate: [0 0]
                   LegType: ["fixed"    "float"]
                     Reset: [2 2]
                     Basis: [0 0]
                  Notional: 100
         LatestFloatingRate: [NaN NaN]
```

```
              ResetOffset: [0 0]
    DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [1x2 ratecurve]
       BusinessDayConvention: ["actual"    "actual"]
                    Holidays: NaT
                EndMonthRule: [1 1]
                   StartDate: 05-Mar-2017
                    Maturity: 05-Mar-2022
                        Name: ""
```

```
ForwardValue = parswaprate(Swap,ZeroCurve)
```

```
ForwardValue = 7.3271e-04
```

Load the Market Implied Volatility Data

The market swaption volatilities are quoted in terms of shifted Black volatilities with a `0.8` percent shift.

```
StrikeGrid = [-0.5; -0.25; -0.125; 0; 0.125; 0.25; 0.5; 1.0; 1.5]/100;
MarketStrikes = ForwardValue + StrikeGrid;
Shift = 0.008;  % 0.8 percent shift
MarketShiftedBlackVolatilities = [21.1; 15.3; 14.0; 14.6; 16.0; 17.7; 19.8; 23.9; 26.2]/100;
ATMShiftedBlackVolatility = MarketShiftedBlackVolatilities(StrikeGrid==0);
```

Calibrate Shifted SABR Model Parameters

The `Beta` parameter is predetermined at `0.5`. Use `volatilities` to compute the implied volatility.

```
Beta = 0.5;

% Calibrate Alpha, Rho, and Nu
objFun = @(X) MarketShiftedBlackVolatilities - volatilities(finpricer("Analytic", 'Model', ...
    finmodel("SABR", 'Alpha', X(1), 'Beta', Beta, 'Rho', X(2), 'Nu', X(3), 'Shift', Shift), ...
    'DiscountCurve', ZeroCurve), SwaptionExerciseDate, ForwardValue, MarketStrikes);

X = lsqnonlin(objFun, [0.5 0 0.5], [0 -1 0], [Inf 1 Inf]);

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.


Alpha = X(1);
Rho = X(2);
Nu = X(3);
```

Create SABR Model Using the Calibrated Parameters

Use `finmodel` to create a SABR model object.

```
SABRModel = finmodel("SABR",'Alpha',Alpha,'Beta',Beta,'Rho',Rho,'Nu',Nu,'Shift',Shift)
```

```
SABRModel =
  SABR with properties:

             Alpha: 0.0135
```

```
             Beta: 0.5000
              Rho: 0.4654
               Nu: 0.4957
            Shift: 0.0080
    VolatilityType: "black"
```

Create SABR Pricer Using Calibrated SABR Model and Compute Volatilities

Use `finpricer` to create a SABR pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

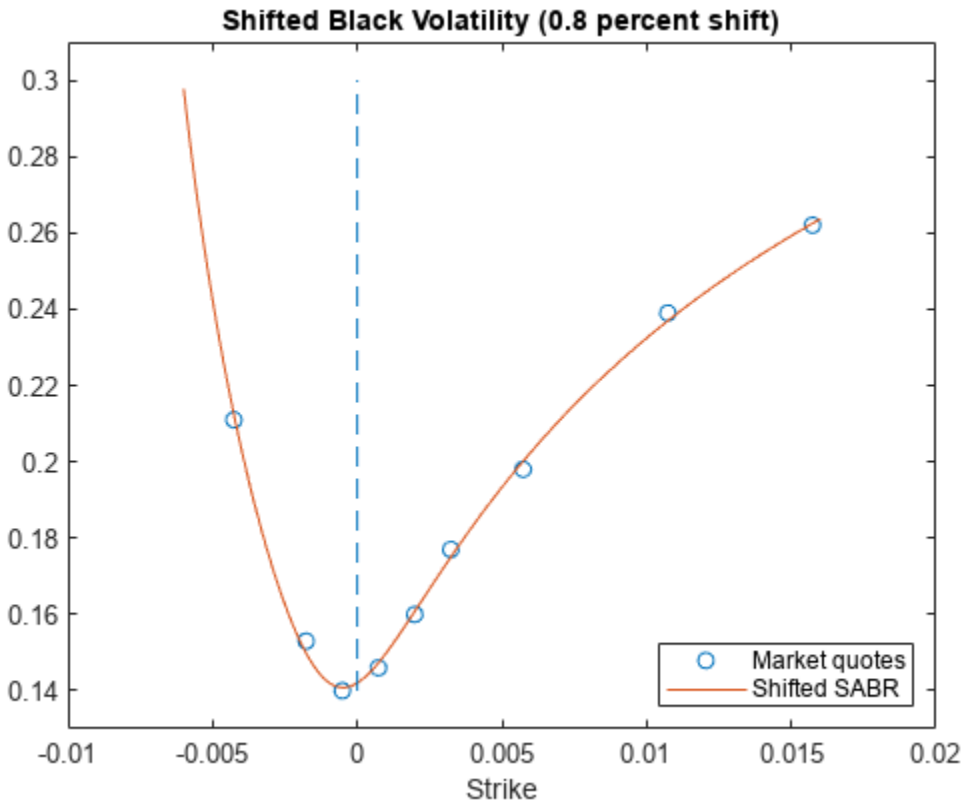
```
SABRPricer = finpricer("Analytic", 'Model', SABRModel, 'DiscountCurve', ZeroCurve)
```

```
SABRPricer =
  SABR with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.SABR]
```

```
SABRShiftedBlackVolatilities = volatilities(SABRPricer, SwaptionExerciseDate, ForwardValue, Swapt
```

```
SABRShiftedBlackVolatilities = 221×1

    0.2978
    0.2911
    0.2848
    0.2787
    0.2729
    0.2673
    0.2620
    0.2568
    0.2518
    0.2470
      ⋮
```

```
figure;
plot(MarketStrikes, MarketShiftedBlackVolatilities, 'o', ...
    SwaptionStrikes, SABRShiftedBlackVolatilities);
h = gca;
line([0,0],[min(h.YLim),max(h.YLim)],'LineStyle','--');
ylim([0.13 0.31])
xlabel('Strike');
legend('Market quotes','Shifted SABR', 'location', 'southeast');
title (['Shifted Black Volatility (',num2str(Shift*100),' percent shift)']);
```



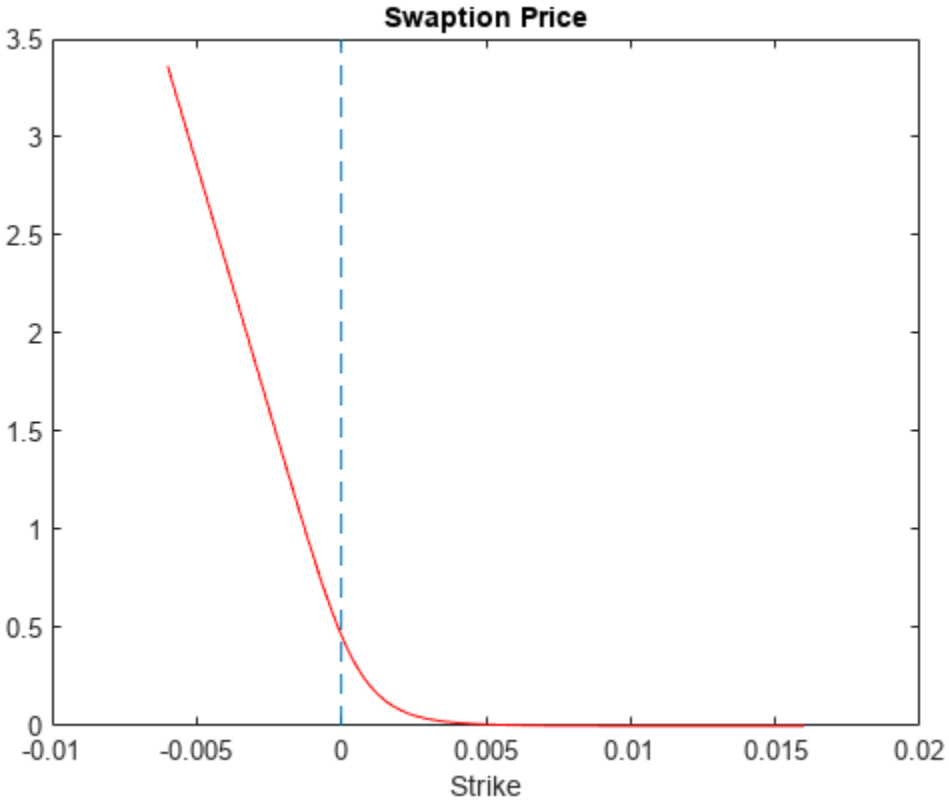
Price Swaption Instruments Using Calibrated SABR Model and SABR Pricer

```
% Create swaption instruments
NumInst = length(SwaptionStrikes);
Swaptions(NumInst, 1) = fininstrument("Swaption", ...
    'Strike', SwaptionStrikes(1), 'ExerciseDate', SwaptionExerciseDate(1), 'Swap', Swap);
for k = 1:NumInst
    Swaptions(k) = fininstrument("Swaption", 'Strike', SwaptionStrikes(k), ...
        'ExerciseDate', SwaptionExerciseDate, 'Swap', Swap, 'OptionType', OptSpec);
end
Swaptions
```

```
Swaptions=221×1 object
  16x1 Swaption array with properties:

    OptionType
    ExerciseStyle
    ExerciseDate
    Strike
    Swap
    Name
      ⋮
```

```
% Price swaptions using the SABR pricer
SwaptionPrices = price(SABRPricer,Swaptions);

figure;
```

```
plot(SwaptionStrikes, SwaptionPrices, 'r');
h = gca;
line([0,0],[min(h.YLim),max(h.YLim)],'LineStyle','--');
xlabel('Strike');
title ('Swaption Price');
```



More About

Call 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.

Put Swaption

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.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Swaption` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`Swap` | `finmodel` | `finpricer` | `volatilities`

Topics
“Calibrate Shifted SABR Model Parameters for Swaption Instrument” on page 2-167
“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62
“Work with Negative Interest Rates Using Objects” on page 2-22

Vanilla

`Vanilla` instrument object

Description

Create and price a `Vanilla` instrument object for one or more Vanilla instruments using this workflow:

1. Use `fininstrument` to create a `Vanilla` instrument object for one or more Vanilla instruments.
2. Use `finmodel` to specify a `BlackScholes`, `Bachelier`, `Heston`, `Bates`, `Merton`, or `Dupire` model for the `Vanilla` instrument object.
3. Choose a pricing method.
   - When using a `BlackScholes` model, use `finpricer` to specify a `FiniteDifference`, `BlackScholes`, `BjerksundStensland`, `RollGeskeWhaley`, `VannaVolga`, `AssetTree`, or `AssetMonteCarlo` pricing method for one or more `Vanilla` instruments.
   - When using a `Heston`, `Bates`, or `Merton` model, use `finpricer` to specify a `FiniteDifference`, `NumericalIntegration`, `FFT`, or `AssetMonteCarlo` pricing method for one or more `Vanilla` instruments.
   - When using a `Dupire` model, use `finpricer` to specify a `FiniteDifference` pricing method for one or more `Vanilla` instruments.
   - When using a `Bachelier` model, use `finpricer` to specify an `AssetMonteCarlo` pricing method for one or more `Vanilla` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Vanilla` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
VanillaObj = fininstrument(InstrumentType,'Strike',strike_value,'
ExerciseDate',exercise_date)
VanillaObj = fininstrument( ___ ,Name,Value)
```

Description

`VanillaObj = fininstrument(InstrumentType,'Strike',strike_value,'ExerciseDate',exercise_date)` creates a `Vanilla` object for one or more Vanilla instruments by specifying `InstrumentType` and sets the properties on page 11-2968 for the required name-value pair arguments `Strike` and `ExerciseDate`. For more information on a `Vanilla` instrument, see “More About” on page 11-2991.

`VanillaObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2968 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `VanillaObj = fininstrument("Vanilla",'Strike',100,'ExerciseDate',datetime(2019,1,30),'OptionType',"put",'ExerciseStyle',"American",'Name',"vanilla_instrument")` creates a `Vanilla` put option with an American exercise. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Vanilla"` | string array with values of `"Vanilla"` | character vector with value `'Vanilla'` | cell array of character vectors with values of `'Vanilla'`

Instrument type, specified as a string with the value of `"Vanilla"`, a character vector with the value of `'Vanilla'`, an `NINST`-by-`1` string array with values of `"Vanilla"`, or an `NINST`-by-`1` cell array of character vectors with values of `'Vanilla'`.

Data Types: `char` | `cell` | `string`

`Vanilla` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `VanillaObj = fininstrument("Vanilla",'Strike',100,'ExerciseDate',datetime(2019,1,30),'OptionType',"put",'ExerciseStyle',"American",'Name',"vanilla_instrument")`

Required Vanilla Name-Value Pair Arguments

`Strike` — Option strike price value

nonnegative numeric | nonnegative numeric vector

Option strike price value, specified as the comma-separated pair consisting of `'Strike'` and a scalar nonnegative numeric value or an `NINST`-by-`1` nonnegative numeric vector.

Note When using a `"Bermudan"` `ExerciseStyle` with a `FiniteDifference` pricer, the `Strike` is a vector.

Data Types: `double`

`ExerciseDate` — Option exercise date

datetime array | string array | date character vector

Option exercise date, specified as the comma-separated pair consisting of `'ExerciseDate'` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note For a European option, there is only one `ExerciseDate` on the option expiry date.

When using a `"Bermudan"` `ExerciseStyle` with a `FiniteDifference` pricer, the `ExerciseDate` is a vector.

To support existing code, `Vanilla` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `ExerciseDate` property is stored as a datetime.

Optional Vanilla Name-Value Pair Arguments

`OptionType` — Option type
`"call"` (default) | string with value `"call"` or `"put"` | string array with values of `"call"` or `"put"` | character vector with value `'call'` or `'put'` | cell array of character vectors with values of `'call'` or `'put'`

Option type, specified as the comma-separated pair consisting of `'OptionType'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Note When you use a `RollGeskeWhaley` pricer with a `Vanilla` option, `OptionType` must be `'call'`.

Data Types: `char` | `cell` | `string`

`ExerciseStyle` — Option exercise style
`"European"` (default) | string with value `"European"`, `"American"`, or `"Bermudan"` | string array with values of `"European"`, `"American"`, or `"Bermudan"` | character vector with value `'European'`, `'American'`, or `'Bermudan'` | cell array of character vectors with values of `'European'`, `'American'`, or `'Bermudan'`

Option exercise style, specified as the comma-separated pair consisting of `'ExerciseStyle'` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Note

- When you use a `BlackScholes` pricer with a `Vanilla` option, the `'American'` option type is not supported.
- When you use a `RollGeskeWhaley` or a `BjerksundStensland` pricer with a `Vanilla` option, you must specify an `'American'` option.
- When you use a `NumericalIntegration` pricer with a `Bates`, `Merton`, or `Heston` model for a `Vanilla` option, the `ExerciseStyle` must be `'European'`.
- When you use a `FFT` pricer with a `Bates`, `Merton`, or `Heston` model for a `Vanilla` option, the `ExerciseStyle` must be `'European'`.
- When you use an `AssetMonteCarlo` pricer with a `BlackScholes`, `Bates`, `Merton`, or `Heston` model for a `Vanilla` option, the `ExerciseStyle` can be `'American'`, `'European'`, or `'Bermudan'`.
- When you use a `FiniteDifference` pricer with a `BlackScholes`, `Bachelier`, `Dupire`, `Bates`, `Merton`, or `Heston` model for a `Vanilla` option, the `ExerciseStyle` can be `'American'`, `'European'`, or `'Bermudan'`.

For more information on `ExerciseStyle`, see “Supported Exercise Styles” on page 1-62.

Data Types: `string` | `cell` | `char`

Name — User-defined name for instrument
" " (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of 'Name' and a scalar string or character vector or an NINST-by-1 cell array of character vectors or string array.

Data Types: char | cell | string

Properties

Strike — Option strike price value
nonnegative numeric | nonnegative numeric vector

Option strike price value, returned as a scalar nonnegative numeric or an NINST-by-1 nonnegative numeric vector.

Data Types: double

ExerciseDate — Option exercise date
datetime | vector of datetimes

Option exercise date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: datetime

OptionType — Option type
"call" (default) | string with value "call" or "put" | string array with values of "call" or "put"

Option type, returned as a scalar string or an NINST-by-1 string array with values of "call" or "put".

Data Types: string

ExerciseStyle — Option type
"European" (default) | string with value "European", "American", or "Bermudan" | string array with values of "European", "American", or "Bermudan"

Option exercise style, returned as a scalar string or an NINST-by-1 string array with values of "European", "American", or "Bermudan".

Data Types: string

Name — User-defined name for instrument
" " (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an NINST-by-1 string array.

Data Types: string

Object Functions

setExercisePolicy Set exercise policy for FixedBondOption, FloatBondOption, or Vanilla instrument

Examples

Price Vanilla Instrument Using Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price a `Vanilla` instrument when you use a `BlackScholes` model and a `BlackScholes` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2018,5,1),'Strike',29,'OptionType','
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 01-May-2018
           Strike: 29
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2500
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2019,1,1);
Rate = 0.05;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 1
             Dates: 01-Jan-2019
             Rates: 0.0500
            Settle: 01-Jan-2018
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackScholes Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',30,'D
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 30
    DividendValue: 0.0450
     DividendType: "continuous"
```

Price Vanilla Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 1.2046
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma      Lambda      Vega       Rho       Theta
    ______    ________    ________    _______    ______    _______    _______

    1.2046    -0.36943    0.086269    -9.3396    6.4702    -4.0959    -2.3107
```

Price Multiple Vanilla Instruments Using Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price multiple `Vanilla` instrument when you use a `BlackScholes` model and a `BlackScholes` pricing method.

Create Vanilla Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object for three Vanilla instruments.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime([2018,5,1 ; 2018,6,1 ; 2018,7,1]),'S
```

```
VanillaOpt=3×1 object
  3x1 Vanilla array with properties:

    OptionType
    ExerciseStyle
```

```
    ExerciseDate
    Strike
    Name
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2500
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2019,1,1);
Rate = 0.05;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2019
               Rates: 0.0500
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackScholes Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',30,'D
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 30
    DividendValue: 0.0450
     DividendType: "continuous"
```

Price Vanilla Instruments

Use `price` to compute the prices and sensitivities for the `Vanilla` instruments.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 3×1

    1.2046
    7.9479
   38.9392


outPR=3×1 object
  3x1 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma      Lambda      Vega       Rho       Theta
    ______    ________    ________    _______    ______    _______    _______

    1.2046    -0.36943    0.086269    -9.3396    6.4702    -4.0959    -2.3107


ans=1×7 table
    Price      Delta       Gamma      Lambda      Vega       Rho       Theta
    ______    ________    ________    _______    ______    _______    _______

    7.9479    -0.89786    0.031587    -3.4532    2.9612    -14.535    -0.3563


ans=1×7 table
    Price      Delta        Gamma         Lambda         Vega          Rho       Theta
    ______    ________    __________    ________    ___________    _______    ______

    38.939    -0.97775    1.2279e-06    -0.77043    0.00013814    -34.136    2.0936
```

Price Vanilla Instrument Using Black-Scholes Model and Asset Tree Pricer for LR Binomial Tree

This example shows the workflow to price a `Vanilla` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method using a Leisen-Reimer (LR) binomial tree.

Create Vanilla Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2018,5,1),'Strike',29,'OptionType','
```

```
VanillaOpt =
  Vanilla with properties:
```

```
       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 01-May-2018
           Strike: 29
             Name: "vanilla_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2500
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2019,1,1);
Rate = 0.05;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2019
               Rates: 0.0500
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object for a LR equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
NumPeriods = 15;
LRPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',50,'
```

```
LRPricer =
  LRTree with properties:

    InversionMethod: PP1
             Strike: 50
               Tree: [1x1 struct]
        NumPeriods: 15
```

```
            Model: [1x1 finmodel.BlackScholes]
      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 50
       DividendType: "continuous"
      DividendValue: 0
          TreeDates: [09-Jan-2018    17-Jan-2018    25-Jan-2018    ...    ]
```

```
LRPricer.Tree
```

```
ans = struct with fields:
    Probs: [2x15 double]
    ATree: {1x16 cell}
     dObs: [01-Jan-2018    09-Jan-2018    17-Jan-2018    ...    ]
     tObs: [0 0.0222 0.0444 0.0667 0.0889 0.1111 0.1333 0.1556 0.1778 ... ]
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(LRPricer,VanillaOpt,["all"])
```

```
Price = 3.5022e-06
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
      Price          Delta         Gamma         Vega       Lambda         Rho            Theta
    _________    ___________    _________    _________    _______    ___________    ___________

    3.5022e-06    -1.9331e-06    1.1068e-06    0.0016243    -30.496    -3.6747e-05    -0.0006010
```

```
outPR.PricerData.PriceTree
```

```
ans = struct with fields:
     PTree: {1x16 cell}
    ExTree: {1x16 cell}
      tObs: [0 0.0222 0.0444 0.0667 0.0889 0.1111 0.1333 0.1556 0.1778 ... ]
      dObs: [01-Jan-2018    09-Jan-2018    17-Jan-2018    ...    ]
     Probs: [2x15 double]
```

```
outPR.PricerData.PriceTree.ExTree
```

```
ans=1×16 cell array
  Columns 1 through 5

    {[0]}    {[0 0]}    {[0 0 0]}    {[0 0 0 0]}    {[0 0 0 0 0]}

  Columns 6 through 8
```

```
    {[0 0 0 0 0 0]}    {[0 0 0 0 0 0 0]}    {[0 0 0 0 0 0 0 0]}

  Columns 9 through 11

    {[0 0 0 0 0 0 0 0 0]}    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}

  Columns 12 through 14

    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}

  Columns 15 through 16

    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}
```

Price Vanilla Instrument Using Black-Scholes Model and Asset Tree Pricer for Standard Trinomial Tree

This example shows the workflow to price a `Vanilla` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method using a Standard Trinomial (STT) tree.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2018,5,1),'Strike',29,'OptionType','
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 01-May-2018
           Strike: 29
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2500
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2019,1,1);
```

```
Rate = 0.05;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2019
               Rates: 0.0500
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object for a Standard Trinomial (STT) equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
NumPeriods = 15;
STTPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',50,
```

```
STTPricer =
  STTree with properties:

            Tree: [1x1 struct]
       NumPeriods: 15
            Model: [1x1 finmodel.BlackScholes]
    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 50
     DividendType: "continuous"
    DividendValue: 0
        TreeDates: [09-Jan-2018    17-Jan-2018    25-Jan-2018    ...    ]
```

```
STTPricer.Tree
```

```
ans = struct with fields:
    ATree: {1x16 cell}
    Probs: {1x15 cell}
     dObs: [01-Jan-2018    09-Jan-2018    17-Jan-2018    ...    ]
     tObs: [0 0.0222 0.0444 0.0667 0.0889 0.1111 0.1333 0.1556 0.1778 ... ]
```

Price Vanilla Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(STTPricer,VanillaOpt,["all"])
```

```
Price = 6.3773e-05
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
```

```
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
      Price          Delta          Gamma          Vega       Lambda         Rho           Theta
    __________    ___________    __________    _________    _______    ___________    _________

    6.3773e-05    -9.1432e-06    1.2388e-06    0.0034421    -21.514    -0.00064994    -0.001188
```

```
outPR.PricerData.PriceTree
```

```
ans = struct with fields:
     PTree: {1x16 cell}
    ExTree: {1x16 cell}
      tObs: [0 0.0222 0.0444 0.0667 0.0889 0.1111 0.1333 0.1556 0.1778 ... ]
      dObs: [01-Jan-2018    09-Jan-2018    17-Jan-2018    ...    ]
     Probs: {1x15 cell}
```

```
outPR.PricerData.PriceTree.ExTree
```

```
ans=1×16 cell array
  Columns 1 through 4

    {[0]}    {[0 0 0]}    {[0 0 0 0 0]}    {[0 0 0 0 0 0 0]}

  Columns 5 through 7

    {[0 0 0 0 0 0 0 0 0]}    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}

  Columns 8 through 10

    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}

  Columns 11 through 13

    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}

  Columns 14 through 16

    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}    {[0 0 0 0 0 0 ... ]}
```

Price American Option for Vanilla Instrument Using Black-Scholes Model and Roll-Geske-Whaley Pricer

This example shows the workflow to price an American option for a `Vanilla` instrument when you use a `BlackScholes` model and a `RollGeskeWhaley` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionType
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2022
           Strike: 105
             Name: "vanilla_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes","Volatility",0.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create RollGeskeWhaley Pricer Object

Use `finpricer` to create a `RollGeskeWhaley` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',100,
```

```
outPricer =
  RollGeskeWhaley with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
```

```
    DividendValue: [1x1 timetable]
     DividendType: "cash"
```

Price Vanilla Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 19.9066

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Lambda     Vega      Theta      Rho
    _____    ________    _________    ______    ______    _______    _____

    19.907    0.66568    0.0090971    3.344     72.804    -3.4537    186.68
```

Price a Vanilla Instrument for Foreign Exchange Using Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price a `Vanilla` instrument for foreign exchange (FX) when you use a `BlackScholes` model and a `BlackScholes` pricing method. Assume that the current exchange rate is $0.52 and has a volatility of 12% per annum. The annualized continuously compounded foreign risk-free rate is 8% per annum.

Create Vanilla Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2022,9,15),'Strike',.50,'OptionType
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
           Strike: 0.5000
             Name: "vanilla_fx_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
Sigma = .12;
BlackScholesModel = finmodel("BlackScholes","Volatility",Sigma)

BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.1200
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)

myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
              Dates: 15-Sep-2023
              Rates: 0.0350
             Settle: 15-Sep-2018
       InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create `BlackScholes` Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument. When pricing currencies using a `Vanilla` instrument, the `DividendType` must be `'continuous'` and `DividendValue` is the annualized risk-free interest rate in the foreign country.

```
ForeignRate = 0.08;
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',.52,

outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 0.5200
    DividendValue: 0.0800
     DividendType: "continuous"
```

Price `Vanilla` FX Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` FX instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 0.0738

outPR =
  priceresult with properties:

      Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
     Price       Delta       Gamma      Lambda       Vega         Rho         Theta
    ________    ________    _______    ________    ________    ________    _________

    0.073778    -0.49349    2.0818     -4.7899     0.27021     -1.3216     -0.013019
```

Price American Option for Vanilla Instrument Using Black-Scholes Model and Asset Monte-Carlo Pricer

This example shows the workflow to price an American option for a `Vanilla` instrument when you use a `BlackScholes` model and an `AssetMonteCarlo` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionType
```

```
VanillaOpt =
  Vanilla with properties:

      OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2022
           Strike: 105
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes","Volatility",0.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BlackScholesModel,'SpotPrice
```

```
outPricer =
  GBMMonteCarlo with properties:

     DiscountCurve: [1x1 ratecurve]
         SpotPrice: 150
    SimulationDates: 15-Sep-2022
         NumTrials: 1000
     RandomNumbers: []
             Model: [1x1 finmodel.BlackScholes]
      DividendType: "continuous"
     DividendValue: 0
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 61.2010

outPR =
  priceresult with properties:

      Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma      Lambda     Rho       Theta      Vega
    ______    _______    _________    ______    ______    _______    ______
```

```
    61.201    0.93074    0.0027813    2.2812    313.95    -3.7909    41.626
```

Price American Option for Vanilla Instrument Using Heston Model and Asset Monte-Carlo Pricer

This example shows the workflow to price an American option for a `Vanilla` instrument when you use a `Heston` model and an `AssetMonteCarlo` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionType
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2022
           Strike: 105
             Name: "vanilla_option"
```

Create `Heston` Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.032,'ThetaV',0.07,'Kappa',0.003,'SigmaV',0.02,'RhoSV',0.09
```

```
HestonModel =
  Heston with properties:

         V0: 0.0320
    ThetaV: 0.0700
     Kappa: 0.0030
    SigmaV: 0.0200
     RhoSV: 0.0900
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
```

```
                Dates: 15-Sep-2023
                Rates: 0.0350
               Settle: 15-Sep-2018
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",HestonModel,'SpotPrice',150,
```

```
outPricer =
  HestonMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 150
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Heston]
       DividendType: "continuous"
      DividendValue: 0
```

Price Vanilla Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 60.5637
```

```
outPR =
  priceresult with properties:

       Results: [1x8 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×8 table
    Price      Delta       Gamma       Lambda     Rho       Theta      Vega     VegaLT
    ______    _______    _________    ______    ______    _______    ______    _______

    60.564    0.94774    0.0011954    2.3473    326.36    -3.7126    35.272    0.31155
```

Price Bermudan Option for Vanilla Instrument Using Black-Scholes Model and Finite Difference Pricer

This example shows the workflow to price a Bermudan option for a `Vanilla` instrument when you use a `BlackScholes` model and a `FiniteDifference` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'Strike',[110,120],'ExerciseDate',[datetime(2022,9,15) , dat
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "bermudan"
     ExerciseDate: [15-Sep-2022    15-Sep-2023]
           Strike: [110 120]
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes","Volatility",0.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 12
             Dates: 15-Sep-2023
             Rates: 0.0350
            Settle: 15-Sep-2018
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `FiniteDifference` Pricer Object

Use `finpricer` to create a `FiniteDifference` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("FiniteDifference",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPric
```

```
outPricer =
  FiniteDifference with properties:

     DiscountCurve: [1x1 ratecurve]
             Model: [1x1 finmodel.BlackScholes]
         SpotPrice: 100
    GridProperties: [1x1 struct]
      DividendType: "continuous"
     DividendValue: 0
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 18.6797
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta       Gamma       Lambda     Theta      Rho       Vega
    _____    _______    _________    ______    _______    ______    ______

    18.68    0.62163    0.0091406    3.3278    -3.3154    184.31    83.162
```

Price Vanilla Instrument Using Heston Model and Multiple Different Pricers

This example shows the workflow to price a `Vanilla` instrument when you use a `Heston` model and various pricing methods.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
Settle = datetime(2017,6,29);
Maturity = datemnth(Settle,6);
Strike = 80;
VanillaOpt = fininstrument('Vanilla','ExerciseDate',Maturity,'Strike',Strike,'Name',"vanilla_opti
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 29-Dec-2017
           Strike: 80
             Name: "vanilla_option"
```

Create Heston Model Object

Use finmodel to create a Heston model object.

```
V0 = 0.04;
ThetaV = 0.05;
Kappa = 1.0;
SigmaV = 0.2;
RhoSV = -0.7;

HestonModel = finmodel("Heston",'V0',V0,'ThetaV',ThetaV,'Kappa',Kappa,'SigmaV',SigmaV,'RhoSV',Rho
```

```
HestonModel =
  Heston with properties:

        V0: 0.0400
    ThetaV: 0.0500
     Kappa: 1
    SigmaV: 0.2000
     RhoSV: -0.7000
```

Create ratecurve object

Create a ratecurve object using ratecurve.

```
Rate = 0.03;
ZeroCurve = ratecurve('zero',Settle,Maturity,Rate);
```

Create NumericalIntegration, FFT, and FiniteDifference Pricer Objects

Use finpricer to create a NumericalIntegration, FFT, and FiniteDifference pricer objects and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
SpotPrice = 80;
Strike = 80;
DividendYield = 0.02;

NIPricer = finpricer("NumericalIntegration",'Model', HestonModel,'SpotPrice',SpotPrice,'DiscountC
```

```
NIPricer =
  NumericalIntegration with properties:

               Model: [1x1 finmodel.Heston]
       DiscountCurve: [1x1 ratecurve]
           SpotPrice: 80
        DividendType: "continuous"
       DividendValue: 0.0200
              AbsTol: 1.0000e-10
              RelTol: 1.0000e-10
    IntegrationRange: [1.0000e-09 Inf]
   CharacteristicFcn: @characteristicFcnHeston
           Framework: "heston1993"
      VolRiskPremium: 0
          LittleTrap: 1
```

```
FFTPricer = finpricer("FFT",'Model',HestonModel, ...
    'SpotPrice',SpotPrice,'DiscountCurve',ZeroCurve, ...
    'DividendValue',DividendYield,'NumFFT',8192)

FFTPricer =
  FFT with properties:

                    Model: [1x1 finmodel.Heston]
            DiscountCurve: [1x1 ratecurve]
                SpotPrice: 80
             DividendType: "continuous"
            DividendValue: 0.0200
                   NumFFT: 8192
    CharacteristicFcnStep: 0.0100
            LogStrikeStep: 0.0767
        CharacteristicFcn: @characteristicFcnHeston
            DampingFactor: 1.5000
               Quadrature: "simpson"
          VolRiskPremium: 0
               LittleTrap: 1


FDPricer = finpricer("FiniteDifference",'Model',HestonModel,'SpotPrice',SpotPrice,'DiscountCurve

FDPricer =
  FiniteDifference with properties:

     DiscountCurve: [1x1 ratecurve]
             Model: [1x1 finmodel.Heston]
         SpotPrice: 80
    GridProperties: [1x1 struct]
      DividendType: "continuous"
     DividendValue: 0.0200
```

Price `Vanilla` Instrument

Use the following sensitivities when pricing the `Vanilla` instrument.

```
InpSensitivity = ["delta", "gamma", "theta", "rho", "vega", "vegalt"];
```

Use `price` to compute the price and sensitivities for the `Vanilla` instrument that uses the `NumericalIntegration` pricer.

```
[PriceNI,  outPR_NI]  = price(NIPricer,VanillaOpt,InpSensitivity)

PriceNI = 4.7007

outPR_NI =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

Use `price` to compute the price and sensitivities for the `Vanilla` instrument that uses the `FFT` pricer.

```
[PriceFFT, outPR_FFT] = price(FFTPricer,VanillaOpt,InpSensitivity)
```

```
PriceFFT = 4.7007

outPR_FFT =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

Use `price` to compute the price and sensitivities for the `Vanilla` instrument that uses the `FiniteDifference` pricer.

```
[PriceFD,  outPR_FD]  = price(FDPricer,VanillaOpt,InpSensitivity)

PriceFD = 4.7003

outPR_FD =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

Aggregate the price results.

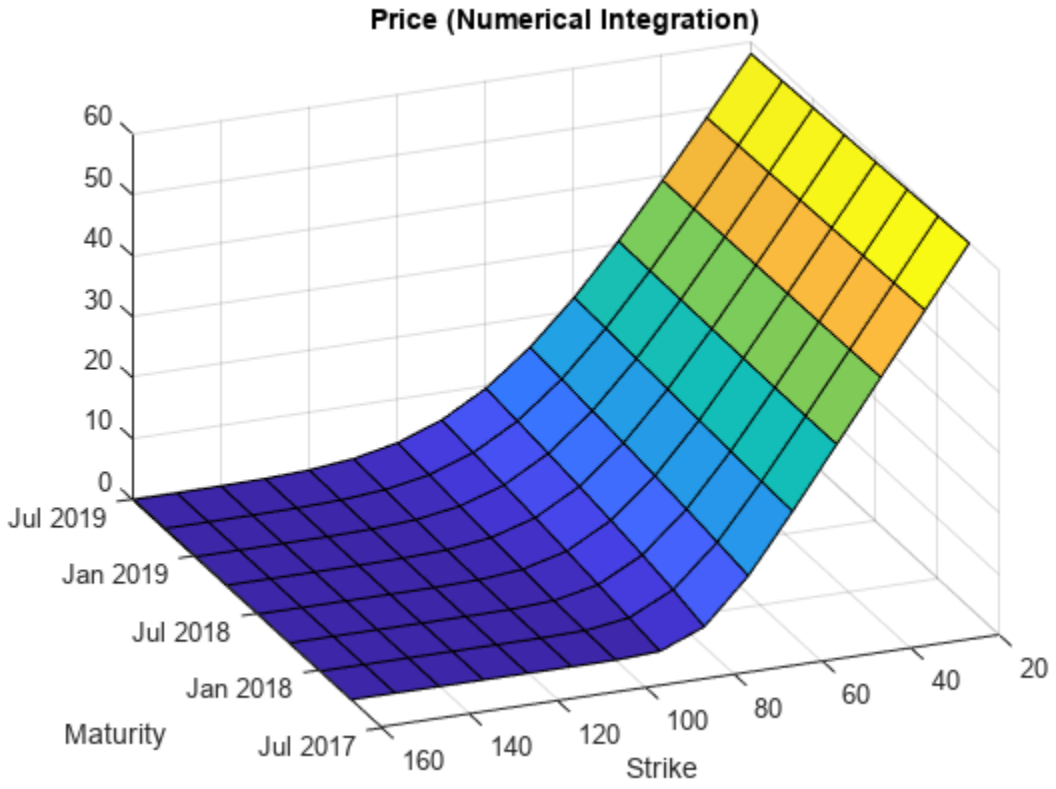
```
[outPR_NI.Results;outPR_FFT.Results;outPR_FD.Results]

ans=3×7 table
    Price      Delta      Gamma       Theta      Rho       Vega      VegaLT
    ______    _______    ________    _______    ______    ______    ______

    4.7007    0.57747     0.03392    -4.8474    20.805    17.028    5.2394
    4.7007    0.57747     0.03392    -4.8474    20.805    17.028    5.2394
    4.7003    0.57722    0.035254    -4.8483    20.801    17.046    5.2422
```

Compute Option Price Surfaces

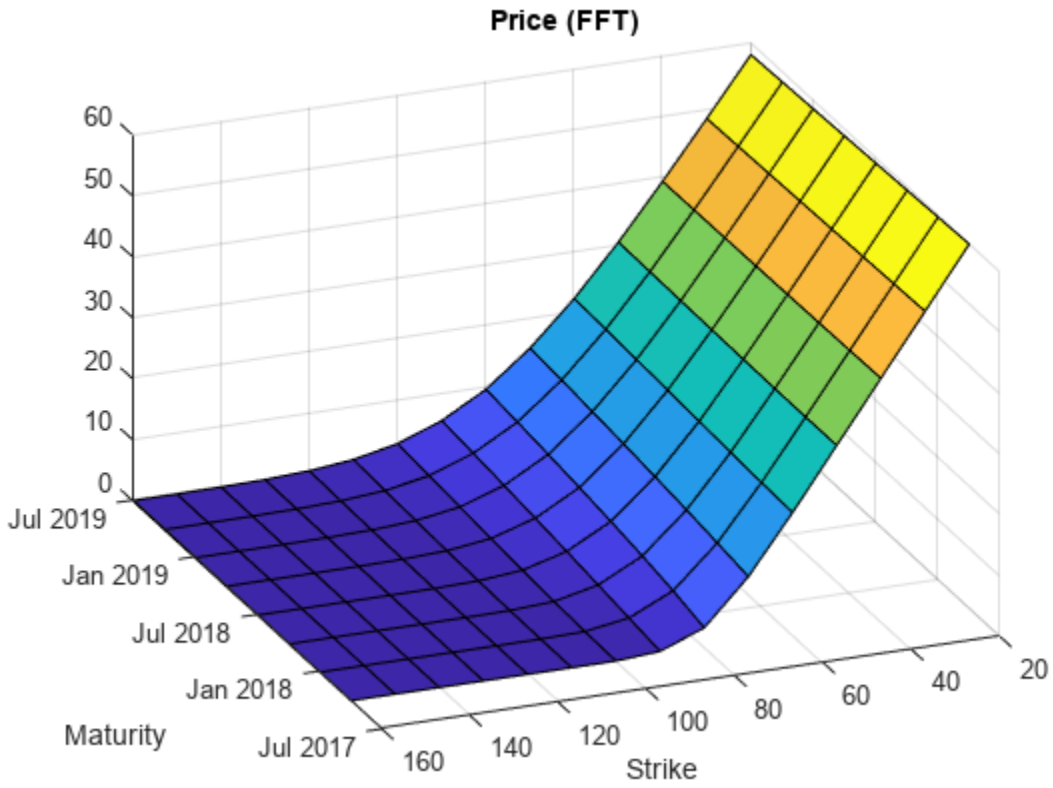
Use the `price` function for the `NumericalIntegration` pricer and the `price` function for the `FFT` pricer to compute the prices for a range of `Vanilla` instruments.

```
Maturities = datemnth(Settle,(3:3:24)');
NumMaturities = length(Maturities);
Strikes = (20:10:160)';
NumStrikes = length(Strikes);

[Maturities_Full,Strikes_Full] = meshgrid(Maturities,Strikes);

NumInst = numel(Strikes_Full);
VanillaOptions(NumInst, 1) = fininstrument("vanilla",...
    "ExerciseDate", Maturities_Full(1), "Strike", Strikes_Full(1));
for instidx=1:NumInst
    VanillaOptions(instidx) = fininstrument("vanilla",...
        "ExerciseDate", Maturities_Full(instidx), "Strike", Strikes_Full(instidx));
end

Prices_NI = price(NIPricer, VanillaOptions);
Prices_FFT = price(FFTPricer, VanillaOptions);
```

```
figure;
surf(Maturities_Full,Strikes_Full,reshape(Prices_NI,[NumStrikes,NumMaturities]));
title('Price (Numerical Integration)');
view(-112,34);
xlabel('Maturity')
ylabel('Strike')
```




```
figure;
surf(Maturities_Full,Strikes_Full,reshape(Prices_FFT,[NumStrikes,NumMaturities]));
title('Price (FFT)');
view(-112,34);
xlabel('Maturity')
ylabel('Strike')
```



More About

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 asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Tips

After creating a `Vanilla` instrument object, you can use `setExercisePolicy` to change the size of the options. For example, if you have the following instrument:

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2021,5,1),'Strike',29,'OptionType','
```

To modify the `Vanilla` instrument's size by changing the `ExerciseStyle` from `"European"` to `"American"`, use `setExercisePolicy`:

```
VanillaOpt = setExercisePolicy(VanillaOpt,[datetime(2021,1,1) datetime(2022,1,1)],100,'American')
```

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Vanilla` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`finmodel` | `finpricer`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Use Black-Scholes Model to Price Asian Options with Several Equity Pricers” on page 3-135
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Supported Exercise Styles” on page 1-62

Deposit

`Deposit` instrument object

Description

Create and price a `Deposit` instrument object for one or more Deposit instruments using this workflow:

1. Use `fininstrument` to create a `Deposit` instrument object for one or more Deposit instruments.
2. Use `ratecurve` to specify an interest-rate model for the `Deposit` instrument object.
3. Use `finpricer` to specify a `Discount` pricing method for one or more `Deposit` instruments.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `Deposit` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
DepositObj = fininstrument(InstrumentType,'Maturity',maturity_date,'
Rate',rate_value)
DepositObj = fininstrument( ___ ,Name,Value)
```

Description

`DepositObj = fininstrument(InstrumentType,'Maturity',maturity_date,'Rate',rate_value)` creates a `Deposit` object for one or more Deposit instruments by specifying `InstrumentType` and sets the properties on page 11-2996 for the required name-value pair arguments `Maturity` and `Rate`.

`DepositObj = fininstrument( ___ ,Name,Value)` sets optional properties on page 11-2996 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `DepositObj = fininstrument("Deposit",'Maturity',datetime(2019,1,30),'Rate',0.027,'Period',2,'Basis',1,'Principal',100,'BusinessDayConvention',"follow",'Name',"deposit_instrument")` creates a `Deposit` instrument with an interest rate of .027 and a maturity of January 30, 2019. You can specify multiple name-value pair arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"Deposit"` | string array with values of `"Deposit"` | character vector with value `'Deposit'` | cell array of character vectors with values of `'Deposit'`

Instrument type, specified as a string with the value of "Deposit", a character vector with the value of 'Deposit', an NINST-by-1 string array with values of "Deposit", or an NINST-by-1 cell array of character vectors with values of 'Deposit'.

Data Types: char | string

Deposit Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: DepositObj = fininstrument("Deposit",'Maturity',datetime(2019,1,30),'Rate',0.027,'Period',2,'Basis',1,'Principal',100,'BusinessDayConvention',"follow",'Name',"deposit_instrument")

Required Deposit Name-Value Pair Arguments

Maturity — Maturity date
datetime array | string array | date character vector

Maturity date, specified as the comma-separated pair consisting of 'Maturity' and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, Deposit also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by datetime because the Maturity property is stored as a datetime.

Rate — Deposit interest rate
scalar decimal | vector of decimals

Deposit interest rate, specified as the comma-separated pair consisting of 'Rate' and a scalar decimal or an NINST-by-1 vector of decimals.

Data Types: double

Optional Deposit Name-Value Pair Arguments

Period — Frequency of payments per year
1 (default) | scalar numeric value of 0, 1, 2, 3, 4, 6, or 12 | vector of numeric values of 0, 1, 2, 3, 4, 6, or 12

Frequency of payments per year, specified as the comma-separated pair consisting of 'Period' and scalar integer or an NINST-by-1 vector of integers. Values for Period are: 0, 1, 2, 3, 4, 6, or 12.

Data Types: double

Basis — Day count basis
0 (actual/actual) (default) | scalar integer from 0 to 13 | vector of integers from 0 to 13

Day count basis, specified as the comma-separated pair consisting of 'Basis' and a scalar integer or an NINST-by-1 vector of integers using the following values:

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

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Principal` — Principal amount
`100` (default) | scalar numeric | numeric vector

Principal amount, specified as the comma-separated pair consisting of `'Principal'` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`BusinessDayConvention` — Business day convention
`"actual"` (default) | string | string array | character vector | cell array of character vectors

Business day convention, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a string or character vector or an `NINST`-by-`1` cell array of character vectors or string array. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `"actual"` — Nonbusiness days are effectively ignored. Cash flows that fall on nonbusiness days are assumed to be distributed on the actual date.
- `"follow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `"modifiedfollow"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `"previous"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `"modifiedprevious"` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: char | cell | string

Holidays — Holidays used in computing business days
NaT (default) | datetime array | string array | date character vector

Holidays used in computing business days, specified as the comma-separated pair consisting of 'Holidays' and dates using an NINST-by-1 vector of a datetime array, string array, or date character vectors. For example:

```
H = holidays(datetime('today'),datetime(2025,12,15));
DepositObj = fininstrument("deposit",'Maturity',datetime(2025,12,15),'Rate',0.027,'Holidays',H)
```

To support existing code, Deposit also accepts serial date numbers as inputs, but they are not recommended.

Name — User-defined name for instrument
" " (default) | string | string array | character vector | cell array of character vectors

User-defined name for one of more instruments, specified as the comma-separated pair consisting of 'Name' and a scalar string or character vector or an NINST-by-1 cell array of character vectors or string array.

Data Types: char | cell | string

Properties

Maturity — Maturity date
datetime | vector of datetimes

Maturity date, returned as a scalar datetime or an NINST-by-1 vector of datetimes.

Data Types: datetime

Rate — Deposit interest rate
scalar decimal | vector of decimals

Deposit interest rate, returned as a scalar decimal or an NINST-by-1 vector of decimals.

Data Types: double

Period — Frequency of payments per year
1 (default) | scalar integer | vector of integers

Frequency of payments per year, returned as a scalar integer or an NINST-by-1 vector of integers.

Data Types: double

Basis — Day count basis
0 (actual/actual) (default) | scalar integer from 0 to 13 | vector of integers from 0 to 13

Day count basis, returned as a scalar integer or an NINST-by-1 vector of integers.

Data Types: double

Principal — Principal amount
100 (default) | scalar numeric | numeric vector

Principal amount, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`BusinessDayConvention` — Business day convention
`"actual"` (default) | scalar string | string array

Business day convention, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

`Holidays` — Holidays used in computing business days
`NaT` (default) | datetimes

Holidays used in computing business days, returned as an `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`Name` — User-defined name for instrument
`" "` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

cashflows	Compute cash flow for FixedBond, FloatBond, Swap, FRA, STIRFuture, OISFuture, OvernightIndexedSwap, or Deposit instrument

Examples

Price Deposit Instrument Using `ratecurve` and Discount Pricer

This example shows the workflow to price a `Deposit` instrument when using a `ratecurve` and a `Discount` pricing method.

Create `Deposit` Instrument Object

Use `fininstrument` to create a `Deposit` instrument object.

```
DepositObj = fininstrument("Deposit",'Maturity',datetime(2019,3,15),'Rate',0.0195,'Period',2,'Bas
```

```
DepositObj =
  Deposit with properties:

                     Rate: 0.0195
                   Period: 2
                    Basis: 1
                 Maturity: 15-Mar-2019
                Principal: 100
    BusinessDayConvention: "actual"
                 Holidays: NaT
                     Name: "deposit_instrument"
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create Discount Pricer Object

Use finpricer to create a Discount pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price Deposit Instrument

Use price to compute the price and sensitivities for the Deposit instrument.

```
[Price, outPR] = price(outPricer, DepositObj,["all"])
```

```
Price = 0.9725

outPR =
  priceresult with properties:

      Results: [1x2 table]
   PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
     Price        DV01
    _______    __________
```

```
    0.97249    4.8225e-05
```

Price Multiple Deposit Instruments Using `ratecurve` and Discount Pricer

This example shows the workflow to price multiple `Deposit` instruments when using a `ratecurve` and a `Discount` pricing method.

Create `Deposit` Instrument Object

Use `fininstrument` to create a `Deposit` instrument object for three Deposit instruments.

```
DepositObj = fininstrument("Deposit",'Maturity',datetime([2019,3,15 ; 2019,4,15 ; 2019,5,15]),'Ra
```

```
DepositObj=3×1 object
  3x1 Deposit array with properties:

    Rate
    Period
    Basis
    Maturity
    Principal
    BusinessDayConvention
    Holidays
    Name
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price `Deposit` Instruments

Use `price` to compute the prices and sensitivities for the `Deposit` instruments.

```
[Price, outPR] = price(outPricer, DepositObj,["all"])
```

```
Price = 3×1

    9.7249
   22.6807
   38.8632


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×2 table
    Price        DV01
    ______    __________

    9.7249    0.00048225


ans=1×2 table
    Price       DV01
    ______    _________

    22.681    0.0013173


ans=1×2 table
    Price       DV01
    ______    _________
```

```
    38.863    0.0025767
```

More About

Deposit Instrument

A deposit instrument is an over-the-counter contract between banks for interbank lending.

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `Deposit` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`finmodel` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

BondFuture

`BondFuture` instrument object

Description

Create and price a `BondFuture` instrument object for one or more Bond Future instruments using this workflow:

1. Use `fininstrument` to create a `BondFuture` instrument object for one or more Bond Future instruments.
2. Use `ratecurve` to specify a curve model for the `BondFuture` instrument object.
3. Use `finpricer` to specify a `Future` pricing method for one or more `BondFuture` instruments.
4. Use `cashsettle` to compute the cash settlement for the `BondFuture` instrument and `fairdelivery` to compute the fair delivery price for the underlying asset for the `BondFuture` instrument.

For more detailed information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `BondFuture` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BondFutureObj = fininstrument(InstrumentType,Maturity=maturity_value,
QuotedPrice=quoted_price,Bond=underlying_bond,
ConversionFactor=conversion_factor)
BondFutureObj = fininstrument( ___ ,Name=Value)
```

Description

`BondFutureObj = fininstrument(InstrumentType,Maturity=maturity_value, QuotedPrice=quoted_price,Bond=underlying_bond, ConversionFactor=conversion_factor)` creates a `BondFuture` object for one or more Bond Future instruments by specifying `InstrumentType` and sets the properties on page 11-3004 for the required name-value pair arguments `Maturity`, `QuotedPrice`, `Bond`, and `ConversionFactor`.

The `BondFuture` instrument supports bond futures such as Treasury bond futures, Eurobond futures, Japanese Government Bond (JGB) futures. For more information, see “Bond Futures” on page 11-3009

You can use bond futures for interest-rate futures and forwards. The main difference between forward and futures contracts:

- Forward contracts are private contracts between two parties, while futures contracts are more standardized and are traded on an exchange.
- Futures contracts are marked-to-market with corresponding settlement cash flows occurring on every trading day, while forward contract cash flows and deliveries do not occur until the maturity of the forward contract.

`BondFutureObj = fininstrument( ___ ,Name=Value)` sets optional properties on page 11-3004 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `BondFutureObj = fininstrument("BondFuture",Maturity=datetime(2022,9,1),QuotedPrice=86,Bond=myFixedBond,ConversionFactor=1.43,Name="bondfuture_instrument")` creates a `BondFuture` instrument. You can specify multiple name-value arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"BondFuture"` | string array with values of `"BondFuture"` | character vector with value `'BondFuture'` | cell array of character vectors with values of `'BondFuture'`

Instrument type, specified as a string with the value of `"BondFuture"`, a character vector with the value of `'BondFuture'`, an `NINST`-by-`1` string array with values of `"BondFuture"`, or an `NINST`-by-`1` cell array of character vectors with values of `'BondFuture'`.

Data Types: `char` | `cell` | `string`

`BondFuture` Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `BondFutureObj = fininstrument("BondFuture",Maturity=datetime(2022,9,1),QuotedPrice=86,SpotPrice=125,Bond=myFixedBond,ConversionFactor=1.43,Name="bondfuture_instrument")`

Required `BondFuture` Name-Value Arguments

`Maturity` — `BondFuture` maturity date

datetime array | string array | date character vector

`BondFuture` maturity date, specified as `Maturity` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array,or date character vectors.

To support existing code, `BondFuture` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`QuotedPrice` — `BondFuture` quoted price

scalar numeric | numeric vector

`BondFuture` quoted price, specified as `QuotedPrice` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

Bond — Underlying FixedBond instrument
FixedBond object | vector of FixedBond objects

Underlying FixedBond instrument, specified as a scalar FixedBond object or an NINST-by-1 vector of FixedBond objects.

Data Types: object

ConversionFactor — Conversion factor for underlying Bond
scalar numeric | numeric vector

Conversion factor for underlying Bond, specified as a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

Optional BondFuture Name-Value Arguments

Notional — Notional value
100000 (default) | scalar numeric | numeric vector

Notional value, specified as a scalar numeric or an NINST-by-1 vector.

Data Types: double

Name — User-defined name for instrument
" " (default) | string | string array | character vector | cell array of character vectors

User-defined name for one or more instruments, specified as Name and a scalar string, character vector, or an NINST-by-1 cell array of character vectors, or a string array.

Data Types: char | cell | string

Properties

Maturity — BondFuture maturity date
scalar datetime | vector of datetimes

BondFuture maturity date, returned as a scalar datetime or NINST-by-1 vector of datetimes.

Data Types: datetime

QuotedPrice — BondFuture quoted price
scalar numeric | numeric vector

BondFuture quoted price, returned as a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

Bond — Underlying FixedBond instrument
FixedBond object

Underlying FixedBond instrument, returned as a scalar FixedBond object or an NINST-by-1 vector of FixedBond objects.

Data Types: object

`ConversionFactor` — Conversion factor for underlying `Bond`

scalar numeric

Conversion factor for underlying `Bond`, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Notional` — Notional value

`100000` (default) | scalar numeric | numeric vector

Notional value, returned as a scalar numeric or an `NINST`-by-`1` vector.

Data Types: `double`

`Name` — User-defined name for instrument

`""` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

cashsettle	Compute cash settlement for BondFuture, CommodityFuture, EquityIndexFuture, or FXFuture instrument
fairdelivery	Compute fair delivery price of underlying asset for BondFuture, CommodityFuture, EquityIndexFuture, or FXFuture instrument

Examples

Price `BondFuture` Instrument Using `ratecurve` and `Future` Pricer

This example shows the workflow to price a `BondFuture` instrument when you use a `ratecurve` object and a `Future` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2022,3,1);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

Create Underlying `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",Maturity=datetime(2032,9,1),CouponRate=0.05,Name="fixed_bond_ins
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
```

```
                    Period: 2
                     Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Sep-2032
                        Name: "fixed_bond_instrument"
```

Create BondFuture Instrument Object

Use `fininstrument` to create a `BondFuture` instrument object.

```
BondFut = fininstrument("BondFuture",Maturity=datetime(2022,9,1),QuotedPrice=86,Bond=FixB,Convers
```

```
BondFut =
  BondFuture with properties:

            Maturity: 01-Sep-2022
         QuotedPrice: 86
                Bond: [1x1 fininstrument.FixedBond]
    ConversionFactor: 1.4300
            Notional: 100000
                Name: "bondfuture_instrument"
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=ZeroCurve,SpotPrice=125)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 125
```

Price BondFuture Instrument

Use `price` to compute the price and price result for the `BondFuture` instrument.

```
[Price,outPR] = price(outPricer,BondFut)
```

```
Price = -151.9270
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×4 table
     Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    _______    _________________    _______________    _______________

    -151.93       1.2283e+05            85.893                0
```

Price Multiple BondFuture Instruments Using ratecurve and Future Pricer

This example shows the workflow to price multiple `BondFuture` instruments when you use a `ratecurve` object and a `Future` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2022,3,1);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

Create Underlying FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",Maturity=datetime(2032,9,1),CouponRate=0.05,Name="fixed_bond_ins
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Sep-2032
                        Name: "fixed_bond_instrument"
```

Create BondFuture Instrument Object

Use `fininstrument` to create a `BondFuture` instrument object for three Bond Future instruments.

```
BondFut = fininstrument("BondFuture",Maturity=datetime([2022,9,1 ; 2022,10,1 ; 2022,11,1]),Quote
```

```
BondFut=3×1 object
  3x1 BondFuture array with properties:
```

```
    Maturity
    QuotedPrice
    Bond
    ConversionFactor
    Notional
    Name
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=ZeroCurve,SpotPrice=125)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 125
```

Price BondFuture Instruments

Use `price` to compute the prices and price results for the `BondFuture` instruments.

```
[Price,outPR] = price(outPricer,BondFut)
```

```
Price = 3×1
10^3 ×

   -0.1519
   -3.3603
   -6.5765


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×4 table
     Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    _______    _________________    _______________    _______________

    -151.93       1.2283e+05            85.893                0


ans=1×4 table
     Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    _______    _________________    _______________    _______________

    -3360.3       1.2288e+05            85.643             0.41436
```

```
ans=1×4 table
     Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    _______    _________________    _______________    _______________

    -6576.5       1.2294e+05            85.385             0.84254
```

More About

Bond Futures

A bond future is a financial derivatives that obligates the contract holder to purchase or sell a bond on a specified date at a predetermined price.

$$B_0 = (S_0 - I)/DiscountFactor$$

- B_0 — Full fair future cash price for $100 face value of deliverable bond
- S_0 — Full spot cash price for $100 face value of deliverable bond
- I — Present value of interest income for $100 face value for the life of the contract
- *DiscountFactor* — Discount factor to maturity of contract

$$FairFuturePrice = F_0 = (B_0 - AccruedInterest)/ConversionFactor$$
$$FairDeliveryPrice = B_0/100 \times Size$$

- F_0 — Clean fair future cash price for $100 face value
- B_0 — Full fair future cash price for $100 face value of the deliverable bond
- *ConversionFactor* — Conversion factor of the deliverable bond
- *AccruedInterest* — Accrued interest at delivery
- *Size* — Futures contract size

There are two workflows:

- The first workflow is to compute the discounted present value of the futures contract using `cashsettle` assuming it is held until maturity like a forward contract.

  $$f = (B_0 - K) \times DiscountFactor/100 \times Size$$

  - B_0 — Full fair future cash price for $100 face value of the delivery bond
  - K_0 — Full contractual delivery bond price for $100 face value of the deliverable bond
  - *DiscountFactor* — Discount factor to the maturity of the contract
  - *Size* — Futures contract size
- Since futures contracts are settled on every trading day, the second workflow is to use `fairdelivery` to compute the current fair delivery bond price B_0 of the futures contract, fair future price F_0, and the accrued interest to support marking to market.

  $$DelieveryBondPrice(B_0) = FairFuturePrice(F_0) \times ConversionFactor + AccruedInterest$$
  $$MarkToMarket = (ChangeinFuturePrice) \times (ContractSize)$$

  - B_0 — Full fair future cash price for $100 face value of the deliverable bond

- F_0 — Clean fair future price for $100 face value
- *ConversionFactor* — Conversion factor of the deliverable bond
- *AccruedInterest* — Accrued interest at delivery for $100 fave value

Version History

Introduced in R2022a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `BondFuture` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`finmodel` | `finpricer`

Topics

“Select Cheapest-to-Deliver Bond Using BondFuture Instrument” on page 2-212
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

CommodityFuture

`CommodityFuture` instrument object

Description

Create and price a `CommodityFuture` instrument object for one or more Commodity Future instruments using this workflow:

1. Use `fininstrument` to create a `CommodityFuture` instrument object for one or more Commodity Future instruments.
2. Use `ratecurve` to specify a curve model for the `CommodityFuture` instrument object.
3. Use `finpricer` to specify a `Future` pricing method for one or more `CommodityFuture` instruments.
4. Use `cashsettle` to compute the cash settlement for the `CommodityFuture` instrument and `fairdelivery` to compute the fair delivery price for the underlying asset for the `CommodityFuture` instrument.

For more detailed information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for a `CommodityFuture` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
CommodityFutureObj = fininstrument(InstrumentType,Maturity=maturity_value,
QuotedPrice=quoted_price)
CommodityFutureObj = fininstrument( ___ ,Name=Value)
```

Description

`CommodityFutureObj = fininstrument(InstrumentType,Maturity=maturity_value, QuotedPrice=quoted_price)` creates a `CommodityFuture` object for one of more Commodity Future instruments by specifying `InstrumentType` and sets the properties on page 11-3015 for the required name-value arguments `Maturity` and `QuotedPrice`.

You can use commodity futures for commodity futures and forwards. The main difference between forward and futures contracts:

- Forward contracts are private contracts between two parties, while futures contracts are more standardized and are traded on an exchange.
- Futures contracts are marked-to-market with corresponding settlement cash flows occurring on every trading day, while forward contract cash flows and deliveries do not occur until the maturity of the forward contract.

The CommodityFuture instrument supports oil, natural gas, gold, silver, and wheat, and other commodities. For more information, see “Commodity Future” on page 11-3019.

CommodityFutureObj = fininstrument(___ ,Name=Value) sets optional properties on page 11-3015 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, CommodityFutureObj = fininstrument("CommodityFuture",Maturity=datetime(2022,9,1),QuotedPrice=68,StorageCost=1000,Income=500,PercentStorageCost=0.03,Name="commodity_future_instrument") creates a CommodityFuture. You can specify multiple name-value arguments.

Input Arguments

InstrumentType — Instrument type

string with value "CommodityFuture" | string array with values of "CommodityFuture" | character vector with value 'CommodityFuture' | cell array of character vectors with values of 'CommodityFuture'

Instrument type, specified as a string with the value of "CommodityFuture", a character vector with the value of 'CommodityFuture', an NINST-by-1 string array with values of "CommodityFuture", or an NINST-by-1 cell array of character vectors with values of 'CommodityFuture'.

Data Types: char | cell | string

CommodityFuture Name-Value Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: CommodityFutureObj = fininstrument("CommodityFuture",Maturity=datetime(2022,9,1), QuotedPrice=68,StorageCost=1000,Income=500,PercentStorageCost=0.03,Name="commodity_future_instrument")

Required CommodityFuture Name-Value Arguments

Maturity — CommodityFuture maturity date

datetime array | string array | date character vector

CommodityFuture maturity date, specified as Maturity and a scalar or an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, CommodityFuture also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by datetime because the Maturity property is stored as a datetime.

QuotedPrice — CommodityFuture quoted price

scalar numeric | numeric vector

CommodityFuture quoted price, specified as QuotedPrice and a scalar numeric or an NINST-by-1 numeric vector.

Data Types: double

Optional CommodityFuture Name-Value Arguments

`StorageCost` — Present value of storage cost for life of contract
`0` (default) | scalar numeric | numeric vector

Present value of storage cost for the life of the contract, specified as `StorageCost` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Income` — Present value of income for life of contract
`0` (default) | scalar numeric | numeric vector

Present value of income for the life of the contract, specified as `Income` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`PercentStorageCost` — Annualized percentage storage cost
`0` (default) | scalar decimal | vector of decimals

Annualized percentage storage cost, specified as `PercentStorageCost` and a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`PercentStorageCostCompounding` — Compounding frequency for `PercentageStorageCost`
`-1` (continuous compounding) (default) | scalar integer with value of `-1`, `0`, `1`, `2`, `4`, `6`, or `12` | vector of integers with values `-1`, `0`, `1`, `2`, `4`, `6`, or `12`

Compounding frequency for `PercentageStorageCost`, specified as `PercentStorageCostCompounding` and a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`PercentageStorageCostBasis` — Day count basis for `PercentStorageCost`
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis for `PercentageStorageCost`, specified as `PercentageStorageCostBasis` and scalar integer or an `NINST`-by-`1` vector of integers using the following values:

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)

- 11 — 30/360E (ICMA)
- 12 — actual/365 (ISDA)
- 13 — BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`ConvenienceYield` — Annualized convenience yield
`0` (default) | scalar decimal | vector of decimals

Annualized convenience yield, specified as `ConvenienceYield` and a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`ConvenienceYieldCompounding` — Compounding frequency for `ConvenienceYield`
`1` (continuous compounding) (default) | scalar integer with value of `-1`, `0`, `1`, `2`, `4`, `6`, or `12` | vector of integers with values `-1`, `0`, `1`, `2`, `4`, `6`, or `12`

Compounding frequency for `ConvenienceYield`, specified as `ConvenienceYieldCompounding` and a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`ConvenienceYieldBasis` — Day count basis for `ConvenienceYield`
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis for `ConvenienceYield`, specified as `ConvenienceYieldBasis` and scalar integer or an `NINST`-by-`1` vector of integers using the following values:

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)
- 11 — 30/360E (ICMA)
- 12 — actual/365 (ISDA)
- 13 — BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Name` — User-defined name for instrument
`""` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one or more instruments, specified as `Name` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Maturity` — `CommodityFuture` maturity date
scalar datetime | vector of datetimes

`CommodityFuture` maturity date, returned as a scalar datetime or `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`QuotedPrice` — `CommodityFuture` quoted price
scalar numeric | numeric vector

`CommodityFuture` quoted price, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`StorageCost` — Present value of storage cost for life of contract
`0` (default) | scalar numeric | numeric vector

Present value of storage cost for the life of the contract, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Income` — Present value of income for life of contract
`0` (default) | scalar numeric | numeric vector

Present value of income for the life of the contract, returned as an `NINST`-by-`1` numeric vector.

Data Types: `double`

`PercentStorageCost` — Annualized percentage storage cost
`0` (default) | scalar decimal | vector of decimals

Annualized percentage storage cost, returned as a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`PercentStorageCostCompounding` — Compounding frequency for `PercentageStorageCost`
`-1` (continuous compounding) (default) | scalar integer with value of `-1`, `0`, `1`, `2`, `4`, `6`, or `12` | vector of integers with values `-1`, `0`, `1`, `2`, `4`, `6`, or `12`

Compounding frequency for `PercentageStorageCost`, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`PercentageStorageCostBasis` — Day count basis for `PercentStorageCost`
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis for `PercentageStorageCost`, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`ConvenienceYield` — Annualized convenience yield
`0` (default) | scalar decimal | vector of decimals

Annualized convenience yield, returned as a scalar decimal or an `NINST`-by-`1` vector of decimals.

Data Types: `double`

`ConvenienceYieldCompounding` — Compounding frequency for `ConvenienceYield`
`1` (continuous compounding) (default) | scalar integer with value of `-1`, `0`, `1`, `2`, `4`, `6`, or `12` | vector of integers with values `-1`, `0`, `1`, `2`, `4`, `6`, or `12`

Compounding frequency for `ConvenienceYield`, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`ConvenienceYieldBasis` — Day count basis for `ConvenienceYield`
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis for `ConvenienceYield`, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Name` — User-defined name for instrument
`""` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

cashsettle	Compute cash settlement for BondFuture, CommodityFuture, EquityIndexFuture, or FXFuture instrument
fairdelivery	Compute fair delivery price of underlying asset for BondFuture, CommodityFuture, EquityIndexFuture, or FXFuture instrument

Examples

Price `CommodityFuture` Instrument Using `ratecurve` and `Future` Pricer

This example shows the workflow to price a `CommodityFuture` instrument when you use a `ratecurve` object and a `Future` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2022,3,1);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
```

```
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

Create `CommodityFuture` Instrument Object

Use `fininstrument` to create a `CommodityFuture` instrument object.

```
CommodityFut = fininstrument("CommodityFuture",Maturity=datetime(2022,9,1),QuotedPrice=68,Storage
```

```
CommodityFut =
  CommodityFuture with properties:

                        Maturity: 01-Sep-2022
                    QuotedPrice: 68
                    StorageCost: 25
                         Income: 0
              PercentStorageCost: 0
    PercentStorageCostCompounding: -1
          PercentStorageCostBasis: 0
                ConvenienceYield: 0
      ConvenienceYieldCompounding: -1
            ConvenienceYieldBasis: 0
                           Name: "commodityfuture_instrument"
```

Create `Future` Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=ZeroCurve,SpotPrice=66)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 66
```

Price `CommodityFuture` Instrument

Use `price` to compute the price and price result for the `CommodityFuture` instrument.

```
[Price,outPR] = price(outPricer,CommodityFut)
```

```
Price = 23.1778

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
```

```
    ______    _________________    _____________    _____________

    23.178         91.239              91.239              0
```

Price Multiple CommodityFuture Instruments Using ratecurve and Future Pricer

This example shows the workflow to price multiple `CommodityFuture` instruments when you use a `ratecurve` object and a `Future` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2022,3,1);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

Create CommodityFuture Instrument Object

Use `fininstrument` to create a `CommodityFuture` instrument object for three Commodity Future instruments.

```
CommodityFut = fininstrument("CommodityFuture",Maturity=datetime([2022,9,1 ; 2022,10,1 ; 2022,11
```

```
CommodityFut=3×1 object
  3x1 CommodityFuture array with properties:

    Maturity
    QuotedPrice
    StorageCost
    Income
    PercentStorageCost
    PercentStorageCostCompounding
    PercentStorageCostBasis
    ConvenienceYield
    ConvenienceYieldCompounding
    ConvenienceYieldBasis
    Name
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=ZeroCurve,SpotPrice=66)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 66
```

Price CommodityFuture Instruments

Use `price` to compute the prices and price results for the `CommodityFuture` instruments.

```
[Price,outPR] = price(outPricer,CommodityFut)
```

```
Price = 3×1

   24.0976
   22.2855
   20.4822


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    24.098         92.161              92.161                0


ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    22.286         92.354              92.354                0


ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    20.482         92.555              92.555                0
```

More About

Commodity Future

A commodity future contract is an agreement to buy or sell a predetermined amount of a commodity at a specific price on a specific date in the future.

In the commodity futures market, the underlying commodity has a standardized size and quality in the contract specification (for example, 1000 barrels of light sweet crude oil for WTI Crude Oil futures). During the life of the contract, the underlying commodity may have unit storage cost or unit income that occurs at specific times. It may also have percentage storage cost or convenience yield that is proportional to the price of the commodity.

$$F_0 = (S_0 + U - I)\exp(r + w - y)T$$

$$FairDeliveryPrice = F_0$$

- F_0 — Fair future price of commodity
- S_0 — Spot price of commodity
- U — Present value of storage cost for the life of the contract
- I — Present value of interest income for the life of the contract
- r — Continuously compounded zero rate
- w — Continuously compounded annualized percent storage cost
- y — Continuously compounded annualized convenience yield
- T — Time to maturity of contract

There are two workflows:

- The first workflow is to compute the discounted present value of the futures contract using `cashsettle` assuming it is held until maturity like a forward contract.

  $$f = (F_0 - K)\exp(-rT)Size = (F_0 - K) \times DiscountFactor \times Size$$

  - F_0 — Fair future price
  - K — Contractual price
  - r — Continuously compounded zero rate
  - T — Time to maturity of contract
- Since futures contracts are settled on every trading day, the second workflow is to use `fairdelivery` to compute the current fair delivery price of the futures contract to support marking to market.

  $$MarkToMarket = (ChangeinFuturePrice) \times (ContractSize)$$

Version History

Introduced in R2022a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `CommodityFuture` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`FXFuture` | `EquityIndexFuture` | `finmodel` | `finpricer`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

EquityIndexFuture

`EquityIndexFuture` instrument object

Description

Create and price an `EquityIndexFuture` instrument object for one or more Equity Index Future instruments using this workflow:

1. Use `fininstrument` to create an `EquityIndexFuture` instrument object for one or more Equity Index Future instruments.
2. Use `ratecurve` to specify a curve model for the `EquityIndexFuture` instrument object.
3. Use `finpricer` to specify a `Future` pricing method for one or more `EquityIndexFuture` instruments.
4. Use `cashsettle` to compute the cash settlement for the `EquityIndexFuture` instrument and `fairdelivery` to compute the fair delivery price for the underlying asset for the `EquityIndexFuture` instrument.

For more detailed information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for an `EquityIndexFuture` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
EquityIndexFutureObj = fininstrument(InstrumentType,Maturity=maturity_value,
QuotedPrice=quoted_price)
EquityIndexFutureObj = fininstrument( ___ ,Name=Value)
```

Description

`EquityIndexFutureObj = fininstrument(InstrumentType,Maturity=maturity_value, QuotedPrice=quoted_price)` creates an `EquityIndexFuture` object for one or more Equity Index Future instruments by specifying `InstrumentType` and sets the properties on page 11-3024 for the required name-value arguments `Maturity` and `QuotedPrice`.

The `EquityIndexFuture` instrument supports stock index futures such as NASDAQ 100 and Dow Jones index futures. For more information, see “Equity Index Futures” on page 11-3028.

`EquityIndexFutureObj = fininstrument( ___ ,Name=Value)` sets optional properties on page 11-3024 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `EquityIndexFutureObj = fininstrument("EquityIndexFuture",Maturity=datetime(2022,9,1),QuotedPrice=4800,Size=500,Name="equityindexfuture_instrument")` creates an `EquityIndexFuture` instrument. You can specify multiple name-value arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"EquityIndexFuture"` | string array with values of `"EquityIndexFuture"` | character vector with value `'EquityIndexFuture'` | cell array of character vectors with values of `'EquityIndexFuture'`

Instrument type, specified as a string with the value of `"EquityIndexFuture"`, a character vector with the value of `'EquityIndexFuture'`, an `NINST`-by-`1` string array with values of `"EquityIndexFuture"`, or an `NINST`-by-`1` cell array of character vectors with values of `'EquityIndexFuture'`.

Data Types: `char` | `cell` | `string`

`EquityIndexFuture` Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `EquityIndexFutureObj = fininstrument("EquityIndexFuture",Maturity=datetime(2022,9,1),QuotedPrice=4800,Size=500,Name="equityindexfuture_instrument")`

Required `EquityIndexFuture` Name-Value Arguments

`Maturity` — `EquityIndexFuture` maturity date

datetime array | string array | date character vector

`EquityIndexFuture` maturity date, specified as `Maturity` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `EquityIndexFuture` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`QuotedPrice` — `EquityIndexFuture` quoted delivery index price

scalar numeric | numeric vector

`EquityIndexFuture` quoted delivery index price, specified as `QuotedPrice` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

Optional `EquityIndexFuture` Name-Value Arguments

`Size` — Contract size multiplier

`250` (default) | scalar numeric | numeric vector

Contract size multiplier, specified as `Size` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`DividendYield` — Annualized dividend yield for life of contract

`0` (default) | scalar numeric | numeric vector

Annualized dividend yield for the life of the contract, specified as `DividendYield` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`DividendYieldCompounding` — Compounding frequency for `DividendYield`
`-1` (continuous compounding) (default) | scalar integer with value of `-1`, `0`, `1`, `2`, `4`, `6`, or `12` | vector of integers with values `-1`, `0`, `1`, `2`, `4`, `6`, or `12`

Compounding frequency for `DividendYield`, specified as `DividendYieldCompounding` and a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`DividendYieldBasis` — Day count basis for `DividendYield`
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis for `DividendYield`, specified as `DividendYieldBasis` and scalar integer or an `NINST`-by-`1` vector of integers using the following values:

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)
- 11 — 30/360E (ICMA)
- 12 — actual/365 (ISDA)
- 13 — BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

`Name` — User-defined name for instrument
`""` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one rf more instruments, specified as `Name` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Maturity` — `EquityIndexFuture` maturity date
scalar datetime | vector of datetimes

`EquityIndexFuture` maturity date, returned as a scalar datetime or `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`QuotedPrice` — `EquityIndexFuture` quoted price
scalar numeric | numeric vector

`EquityIndexFuture` quoted price, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Size` — Contract size multiplier
`250` (default) | scalar numeric | numeric vector

Contract size multiplier, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`DividendYield` — Annualized dividend yield for life of contract
`0` (default) | scalar numeric | numeric vector

Annualized dividend yield for the life of the contract, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`DividendYieldCompounding` — Compounding frequency for `DividendYield`
`-1` (continuous compounding) (default) | scalar integer with value of `-1`, `0`, `1`, `2`, `4`, `6`, or `12` | vector of integers with values `-1`, `0`, `1`, `2`, `4`, `6`, or `12`

Compounding frequency for `DividendYield`, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`DividendYieldBasis` — Day count basis for `DividendYield`
`0` (actual/actual) (default) | scalar integer from `0` to `13` | vector of integers from `0` to `13`

Day count basis for `DividendYield`, returned as a scalar integer or an `NINST`-by-`1` vector of integers.

Data Types: `double`

`Name` — User-defined name for instrument
`""` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

cashsettle	Compute cash settlement for BondFuture, CommodityFuture, EquityIndexFuture, or FXFuture instrument
fairdelivery	Compute fair delivery price of underlying asset for BondFuture, CommodityFuture, EquityIndexFuture, or FXFuture instrument

Examples

Price EquityIndexFuture Instrument Using ratecurve and Future Pricer

This example shows the workflow to price an `EquityIndexFuture` instrument when you use a `ratecurve` object and a `Future` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2022,3,1);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

Create EquityIndexFuture Instrument Object

Use `fininstrument` to create an `EquityIndexFuture` instrument object.

```
EquityIndexFut = fininstrument("EquityIndexFuture",Maturity=datetime(2022,9,1),QuotedPrice=4800,I
```

```
EquityIndexFut =
  EquityIndexFuture with properties:

                  Maturity: 01-Sep-2022
               QuotedPrice: 4800
                      Size: 250
             DividendYield: 0.0300
   DividendYieldCompounding: -1
         DividendYieldBasis: 0
                      Name: "equityindexfuture_instrument"
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=ZeroCurve,SpotPrice=4460)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 4460
```

Price EquityIndexFuture Instrument

Use `price` to compute the price and price result for the `EquityIndexFuture` instrument.

```
[Price,outPR] = price(outPricer,EquityIndexFut)
```

```
Price = -9.8598e+04

outPR =
  priceresult with properties:
```

```
       Results: [1x4 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    -98598       1.1011e+06            4404.6                0
```

Price Multiple EquityIndexFuture Instruments Using ratecurve and Future Pricer

This example shows the workflow to price multiple `EquityIndexFuture` instruments when you use a `ratecurve` object and a `Future` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2022,3,1);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

Create EquityIndexFuture Instrument Object

Use `fininstrument` to create an `EquityIndexFuture` instrument object for three Equity Index Future instruments.

```
EquityIndexFut = fininstrument("EquityIndexFuture",Maturity=datetime([2022,9,1 ; 2022,10,1 ; 202
```

```
EquityIndexFut=3×1 object
  3x1 EquityIndexFuture array with properties:

    Maturity
    QuotedPrice
    Size
    DividendYield
    DividendYieldCompounding
    DividendYieldBasis
    Name
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=ZeroCurve,SpotPrice=4460)
```

```
outPricer =
  Future with properties:
```

```
    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 4460
```

Price EquityIndexFuture Instruments

Use `price` to compute the prices and price results for the `EquityIndexFuture` instruments.

```
[Price,outPR] = price(outPricer,EquityIndexFut)
```

```
Price = 3×1
10^5 ×

   -0.9860
   -1.5060
   -2.0262


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    -98598       1.1011e+06            4404.6                0


ans=1×4 table
      Price       FairDeliveryPrice    FairFuturePrice    AccruedInterest
    __________    _________________    _______________    _______________

    -1.506e+05       1.0989e+06            4395.7                0


ans=1×4 table
       Price       FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ___________    _________________    _______________    _______________

    -2.0262e+05       1.0967e+06            4386.6                0
```

More About

Equity Index Futures

An equity index future is a "futures contract" on an equity index.

Equity index futures are cash settled contracts with the majority having quarterly expiration dates scheduled for the months of March, June, September, and December. In the equity index futures market, the underlying asset is an equity index (for example, S&P 500) multiplied by a size multiplier

that is standardized to each specific type of equity index future (for example, $250 for S&P 500 Future, and $50 for E-Mini S&P 500 Future). If the equity index has dividends, computing the fair future price requires the dividend yield for the life of the contract in addition to the interest rate.

$$F_0 = S_0\exp(r - q)T$$
$$FairDeliveryPrice = F_0 \times Size$$

- F_0 — Fair future price of equity index
- S_0 — Spot price of equity index
- r — Continuously compounded zero rate
- q — Continuously compounded annualized dividend yield
- T — Time to maturity of contract
- $Size$ — Size multiplier for equity index futures (for example, $250 for S&P 500 Future)

There are two workflows.

- The first workflow is to compute the discounted present value of the futures contract using `cashsettle` assuming it is held until maturity like a forward contract.

  $$f = (F_0 - K)\exp(-rT)Size = (F_0 - K) \times DiscountFactor \times Size$$

  - F_0 — Fair future price
  - K — Contractual price
  - r — Continuously compounded zero rate
  - T — Time to maturity of contract
- Since futures contracts are settled on every trading day, the second workflow is to use `fairdelivery` to compute the current fair delivery price of the futures contract to support marking to market.

  $$MarkToMarket = (ChangeinFuturePrice) \times (ContractSize)$$

Version History

Introduced in R2022a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `EquityIndexFuture` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`FXFuture` | `CommodityFuture` | `finmodel` | `finpricer`

Topics
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

FXFuture

`FXFuture` instrument object

Description

Create and price an `FXFuture` instrument object for one or more FX Future instruments using this workflow:

1. Use `fininstrument` to create an `FXFuture` instrument object for one or more FX Future instruments.
2. Use `ratecurve` to specify a curve model for the `FXFuture` instrument object.
3. Use `finpricer` to specify a `Discount` pricing method for one or more `FXFuture` instruments.
4. Use `cashsettle` to compute the cash settlement for the `FXFuture` instrument and `fairdelivery` to compute the fair delivery price for the underlying asset for the `FXFuture` instrument.

For more detailed information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available models and pricing methods for an `FXFuture` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
FXFutureObj = fininstrument(InstrumentType,Maturity=maturity_value,
QuotedPrice=quoted_price,ForeignRateCurve=foreign_rate_curve)
FXFutureObj = fininstrument( ___ ,Name=Value)
```

Description

`FXFutureObj = fininstrument(InstrumentType,Maturity=maturity_value, QuotedPrice=quoted_price,ForeignRateCurve=foreign_rate_curve)` creates an `FXFuture` object for one or more FX Future instruments by specifying `InstrumentType` and sets the properties on page 11-3033 for the required name-value arguments `Maturity`, `QuotedPrice`, and `ForeignRateCurve`.

The `FXFuture` instrument supports currency pairs where the price is quoted in domestic currency for one unit of foreign currency. For more information, see “FXFuture” on page 11-3037.

`FXFutureObj = fininstrument( ___ ,Name=Value)` sets optional properties on page 11-3033 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `FXFutureObj = fininstrument("FXFuture",Maturity=datetime(2022,9,1),QuotedPrice=0.78,ForeignRateCurve=ForeignRC,Notional=200000,Name="fxfuture_instrument")` creates an `FXFuture`. You can specify multiple name-value arguments.

Input Arguments

`InstrumentType` — Instrument type

string with value `"FXFuture"` | string array with values of `"FXFuture"` | character vector with value `'FXFuture'` | cell array of character vectors with values of `'FXFuture'`

Instrument type, specified as a string with the value of `"FXFuture"`, a character vector with the value of `'FXFuture'`, an `NINST`-by-`1` string array with values of `"FXFuture"`, or an `NINST`-by-`1` cell array of character vectors with values of `'FXFuture'`.

Data Types: `char` | `cell` | `string`

`FXFuture` Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `FXFutureObj = fininstrument("FXFuture",Maturity=datetime(2022,9,1),QuotedPrice=0.78,ForeignRateCurve=ForeignRC,Notional=200000,Name="fxfuture_instrument")`

Required `FXFuture` Name-Value Arguments

`Maturity` — `FXFuture` maturity date

datetime array | string array | date character vector

`FXFuture` maturity date, specified as `Maturity` and a scalar or an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `FXFuture` also accepts serial date numbers as inputs, but they are not recommended.

If you use date character vectors or strings, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`QuotedPrice` — `FXFuture` delivery price quoted in domestic currency for one unit of foreign currency

scalar numeric | numeric vector

`FXFuture` delivery price quoted in domestic currency for one unit of foreign currency, specified as `QuotedPrice` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`ForeignRateCurve` — Rate curve object for foreign currency

`ratecurve` object

Rate curve object for foreign currency, specified as a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects.

Data Types: `object`

Optional `FXFuture` Name-Value Arguments

`Notional` — Notional in foreign currency

`100000` (default) | scalar numeric | numeric vector

Notional in foreign currency, specified as `Notional` and a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Name` — User-defined name for instrument
`""` (default) | string | string array | character vector | cell array of character vectors

User-defined name for one or more instruments, specified as `Name` and a scalar string or character vector or an `NINST`-by-`1` cell array of character vectors or string array.

Data Types: `char` | `cell` | `string`

Properties

`Maturity` — `FXFuture` maturity date
scalar datetime | vector of datetimes

`FXFuture` maturity date, returned as a scalar datetime or `NINST`-by-`1` vector of datetimes.

Data Types: `datetime`

`QuotedPrice` — `FXFuture` quoted price
scalar numeric | numeric vector

`FXFuture` quoted price, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`ForeignRateCurve` — Rate curve object for foreign currency
`ratecurve` object

Rate curve object for foreign currency, returned as a scalar `ratecurve` object or an `NINST`-by-`1` vector of `ratecurve` objects.

Data Types: `object`

`Notional` — Notional in foreign currency
`100000` (default) | scalar numeric | numeric vector

Notional in foreign currency, returned as a scalar numeric or an `NINST`-by-`1` numeric vector.

Data Types: `double`

`Name` — User-defined name for instrument
`""` (default) | string | string array

User-defined name for the instrument, returned as a scalar string or an `NINST`-by-`1` string array.

Data Types: `string`

Object Functions

cashsettle	Compute cash settlement for BondFuture, CommodityFuture, EquityIndexFuture, or FXFuture instrument
fairdelivery	Compute fair delivery price of underlying asset for BondFuture, CommodityFuture, EquityIndexFuture, or FXFuture instrument

Examples

Price FXFuture Instrument Using ratecurve and Future Pricer

This example shows the workflow to price an FXFuture instrument when you use a ratecurve object and a Future pricing method.

Create ratecurve Objects

Create ratecurve objects using ratecurve for the foreign and domestic zero curves.

```
% Define Foreign Zero Curve
Settle = datetime(2022, 3, 1);
ForeignZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ForeignZeroRates = [0.0031 0.0035 0.0047 0.0058 0.0062 0.0093 0.0128 0.0182 0.0223 0.0285]';
ForeignZeroDates = Settle + ForeignZeroTimes;
ForeignRC = ratecurve('zero', Settle, ForeignZeroDates, ForeignZeroRates);

% Define Domestic Zero Curve
DomesticZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
DomesticZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
DomesticZeroDates = Settle + DomesticZeroTimes;
DomesticRC = ratecurve('zero', Settle, DomesticZeroDates, DomesticZeroRates);
```

Create FXFuture Instrument Object

Use fininstrument to create an FXFuture instrument object.

```
FXFut = fininstrument("FXFuture",Maturity=datetime(2022,9,1),QuotedPrice=0.78,ForeignRateCurve=Fo
```

```
FXFut =
  FXFuture with properties:

           Maturity: 01-Sep-2022
        QuotedPrice: 0.7800
    ForeignRateCurve: [1x1 ratecurve]
           Notional: 200000
               Name: "FXfuture_instrument"
```

Create Future Pricer Object

Use finpricer to create a Future pricer object and use the ratecurve object with the DiscountCurve name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=DomesticRC,SpotPrice=0.79)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 0.7900
```

Price FXFuture Instrument

Use price to compute the price and price result for the FXFuture instrument.

```
[Price,outPR] = price(outPricer,FXFut)

Price = 2.1617e+03

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: []


outPR.Results

ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    2161.7       1.5817e+05           0.79084                0
```

Price Multiple FXFuture Instruments Using ratecurve and Future Pricer

This example shows the workflow to price multiple FXFuture instruments when you use a ratecurve object and a Future pricing method.

Create ratecurve Objects

Create ratecurve objects using ratecurve for the foreign and domestic zero curves.

```
% Define Foreign Zero Curve
Settle = datetime(2022, 3, 1);
ForeignZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ForeignZeroRates = [0.0031 0.0035 0.0047 0.0058 0.0062 0.0093 0.0128 0.0182 0.0223 0.0285]';
ForeignZeroDates = Settle + ForeignZeroTimes;
ForeignRC = ratecurve('zero', Settle, ForeignZeroDates, ForeignZeroRates);

% Define Domestic Zero Curve
DomesticZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
DomesticZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
DomesticZeroDates = Settle + DomesticZeroTimes;
DomesticRC = ratecurve('zero', Settle, DomesticZeroDates, DomesticZeroRates);
```

Create FXFuture Instrument Object

Use fininstrument to create an FXFuture instrument object for three FX Future instruments.

```
FXFut = fininstrument("FXFuture",Maturity=datetime([2022,9,1 ; 2022,10,1 ; 2022,11,1]),QuotedPric

FXFut=3×1 object
  3x1 FXFuture array with properties:

    Maturity
    QuotedPrice
    ForeignRateCurve
    Notional
    Name
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=DomesticRC,SpotPrice=0.79)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 0.7900
```

Price FXFuture Instruments

Use `price` to compute the prices and price results for the `FXFuture` instrument.

```
[Price,outPR] = price(outPricer,FXFut)
```

```
Price = 3×1
10^4 ×

    0.2162
   -0.5789
   -1.3732


outPR=1×3 object
  1x3 priceresult array with properties:

    Results
    PricerData
```

```
outPR.Results
```

```
ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________

    2161.7       1.5817e+05            0.79084              0


ans=1×4 table
    Price    FairDeliveryPrice    FairFuturePrice    AccruedInterest
    _____    _________________    _______________    _______________

    -5789       1.5819e+05            0.79097              0


ans=1×4 table
    Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    ______    _________________    _______________    _______________
```

```
   -13732       1.5822e+05            0.7911                0
```

More About

FXFuture

An FXFuture is an exchange-traded currency derivative contract obligating the buyer and seller to transact at a set price and predetermined time.

In the FX futures market, the underlying asset is a currency pair (for example, price in domestic currency for one unit of foreign currency). Both the domestic currency interest rate and the foreign currency interest rate are required to compute the fair delivery price.

$$F_0 = S_0\exp(r_d - r_f)T$$
$$FairDeliveryPrice = F_0 \times Notional$$

- F_0 — Fair future price in domestic currency for one unit of foreign currency
- S_0 — Spot price in domestic currency for one unit of foreign currency
- r_d — Domestic currency continuously compounded zero rate
- r_f — Foreign currency continuously compounded zero rate
- T — Time to maturity of contract
- $Notional$ — Notional value of the contract in foreign currency

There are two workflows.

- The first workflow is to compute the discounted present value of the futures contract using `cashsettle` assuming it is held until maturity like a forward contract.

  $$f = (F_0 - K)\exp(-rT)Size = (F_0 - K) \times DiscountFactor \times Size$$

  - F_0 — Fair future price
  - K — Contractual price
  - r — Continuously compounded zero rate
  - T — Time to maturity of contract
- Since futures contracts are settled on every trading day, the second workflow is to use `fairdelivery` to compute the current fair delivery price of the futures contract to support marking to market.

  $$MarkToMarket = (ChangeinFuturePrice) \times (ContractSize)$$

Version History

Introduced in R2022a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `FXFuture` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`finmodel` | `finpricer`

Topics
“Price Portfolio of Bond and Bond Option Instruments” on page 2-172
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Bates

Create `Bates` model object for `Vanilla`, `Asian`, `Barrier`, `DoubleBarrier`, `Lookback`, `PartialLookback`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument

Description

Create and price a `Vanilla`, `Asian`, `Barrier`, `DoubleBarrier`, `Lookback`, `PartialLookback`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument object with a `Bates` model using this workflow:

1. Use `fininstrument` to create a `Vanilla`, `Barrier`, `Lookback`, `PartialLookback`, `Asian`, `DoubleBarrier`, `Cliquet`, `Binary`, `Touch`, or `DoubleTouch` instrument object.
2. Use `finmodel` to specify a `Bates` model object for the `Vanilla`, `Asian`, `Barrier`, `DoubleBarrier`, `Lookback`, `PartialLookback`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument object.
3. Use `finpricer` to specify a `FiniteDifference`, `NumericalIntegration`, or `FFT` pricing method for the `Vanilla` instrument object.

   Use `finpricer` to specify an `AssetMonteCarlo` pricing method for the `Vanilla`, `Asian`, `Barrier`, `DoubleBarrier`, `Lookback`, `PartialLookback`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Vanilla`, `Asian`, `Barrier`, `DoubleBarrier`, `Lookback`, `PartialLookback`, `Touch`, `DoubleTouch`, or `Binary` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BatesObj = finmodel(ModelType,'V0',V0_value,'ThetaV',thetav_value,'
Kappa',kappa_value,'SigmaV',sigmav_value,'RhoSV',rhosv_value, '
MeanJ',meanj_value, 'JumpVol',jumpvol_value,'JumpFreq',jumpfreq_value)
```

Description

`BatesObj = finmodel(ModelType,'V0',V0_value,'ThetaV',thetav_value,'Kappa',kappa_value,'SigmaV',sigmav_value,'RhoSV',rhosv_value, 'MeanJ',meanj_value, 'JumpVol',jumpvol_value,'JumpFreq',jumpfreq_value)` creates an `Bates` object by specifying `ModelType` and the required name-value pair arguments `V0`, `ThetaV`, `Kappa`, `SigmaV`, `RhoSV`, `MeanJ`, `JumpVol`, and `JumpFreq`. The required name-value pair arguments set properties on page 11-3041. For example, `BatesObj = finmodel("Bates",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9,'MeanJ',0.11,'JumpVol',.023,'JumpFreq',0.02)` creates a `Bates` model object.

Input Arguments

ModelType — Model type

string with value "Bates" | character vector with value 'Bates'

Model type, specified as a string with the value of "Bates" or a character vector with the value of 'Bates'.

Data Types: char | string

Bates Name-Value Pair Arguments

Specify required pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: Bates = finmodel("Bates",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9,'MeanJ',0.11,'JumpVol',.023,'JumpFreq',0.02)

V0 — Initial variance of underlying asset

numeric

Initial variance of the underlying asset, specified as the comma-separated pair consisting of 'V0' and a scalar numeric value.

Data Types: double

ThetaV — Long-term variance of underlying asset

numeric

Long-term variance of the underlying asset, specified as the comma-separated pair consisting of 'ThetaV' and a scalar numeric value.

Data Types: double

Kappa — Mean revision speed for the variance of underlying asset

numeric

Mean revision speed for the underlying asset, specified as the comma-separated pair consisting of 'Kappa' and a scalar numeric value.

Data Types: double

SigmaV — Volatility of the variance of underlying asset

numeric

Volatility of the variance of the underlying asset, specified as the comma-separated pair consisting of 'SigmaV' and a scalar numeric value.

Data Types: double

RhoSV — Correlation between Weiner processes for underlying asset and its variance

numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as the comma-separated pair consisting of `'RhoSV'` and a scalar numeric value.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
decimal

Mean of the random percentage jump size (*J*), specified as the comma-separated pair consisting of `'MeanJ'` and a scalar decimal value where `log`(1+*J*) is normally distributed with mean (`log`(1+`MeanJ`)-0.5*`JumpVol`^2) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log`(1+*J*)
decimal

Standard deviation of `log`(1+*J*), where `J` is the random percentage jump size, specified as the comma-separated pair consisting of `'JumpVol'` and a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
numeric

Annual frequency of the Poisson jump process, specified as the comma-separated pair consisting of `'JumpFreq'` and a scalar numeric value.

Data Types: `double`

Properties

`V0` — Initial variance of underlying asset
numeric

Initial variance of the underlying asset, returned as a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset
numeric

Long-term variance of the underlying asset, returned as a scalar numeric value.

Data Types: `double`

`Kappa` — Mean revision speed for the variance of underlying asset
numeric

Mean revision speed for the underlying asset, returned as a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset
numeric

Volatility of the variance of the underlying asset, returned as a scalar numeric value.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance
numeric

Correlation between the Weiner processes for the underlying asset and its variance, returned as a scalar numeric value.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
decimal

Mean of the random percentage jump size (*J*), returned as a scalar decimal value.

Data Types: `double`

`JumpVol` — Standard deviation of `log(1+J)`
decimal

Standard deviation of `log`(1+*J*), where `J` is the random percentage jump size, returned as a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
numeric

Annual frequency of the Poisson jump process, returned as a scalar numeric value.

Data Types: `double`

Examples

Use a Bates Model and Numerical Integration Pricer to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `Bates` model and a `NumericalIntegration` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2022,9,15),'Strike',105,'OptionType
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
           Strike: 105
             Name: "vanilla_option"
```

Create Bates Model Object

Use `finmodel` to create a `Bates` model object.

```
BatesModel = finmodel("Bates",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9,'Mea
```

```
BatesModel =
  Bates with properties:

          V0: 0.0320
      ThetaV: 0.1000
       Kappa: 0.0030
      SigmaV: 0.2000
       RhoSV: 0.9000
       MeanJ: 0.1100
     JumpVol: 0.0230
    JumpFreq: 0.0200
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `NumericalIntegration` Pricer Object

Use `finpricer` to create a `NumericalIntegration` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("numericalintegration",'DiscountCurve',myRC,'Model',BatesModel,'SpotPrice'
```

```
outPricer =
  NumericalIntegration with properties:

               Model: [1x1 finmodel.Bates]
       DiscountCurve: [1x1 ratecurve]
           SpotPrice: 100
        DividendType: "continuous"
       DividendValue: 0
              AbsTol: 1.0000e-10
              RelTol: 1.0000e-10
```

```
     IntegrationRange: [1.0000e-09 Inf]
    CharacteristicFcn: @characteristicFcnBates
            Framework: "heston1993"
       VolRiskPremium: 0
           LittleTrap: 1
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 6.4007

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma      Theta      Rho       Vega      VegaLT
    ______    ________    ________    _____    _______    ______    ______

    6.4007    -0.53541    0.02006     1.106    -239.77    94.257    1.3059
```

Use Bates Model and Asset Monte-Carlo Pricer to Price Asian Instrument

This example shows the workflow to price a fixed-strike `Asian` instrument when you use a `Bates` model and an `AssetMonteCarlo` pricing method.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',100,'OptionType',"pu
```

```
AsianOpt =
  Asian with properties:

          OptionType: "put"
              Strike: 100
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_option"
```

Create `Bates` Model Object

Use `finmodel` to create a `Bates` model object.

```
BatesModel = finmodel("Bates",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.02,'RhoSV',0.9,'Me

BatesModel =
  Bates with properties:

         V0: 0.0320
     ThetaV: 0.1000
      Kappa: 0.0030
     SigmaV: 0.0200
      RhoSV: 0.9000
      MeanJ: 0.1100
    JumpVol: 0.0230
   JumpFreq: 0.0200
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)

myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 12
             Dates: 15-Sep-2023
             Rates: 0.0350
            Settle: 15-Sep-2018
      InterpMethod: "linear"
  ShortExtrapMethod: "next"
   LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BatesModel,'SpotPrice',80,'S

outPricer =
  BatesMonteCarlo with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 80
  SimulationDates: 15-Sep-2022
        NumTrials: 1000
    RandomNumbers: []
            Model: [1x1 finmodel.Bates]
     DividendType: "continuous"
    DividendValue: 0
```

Price Asian Instrument

Use `price` to compute the price and sensitivities for the `Asian` instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 14.5650
```

```
outPR =
  priceresult with properties:

       Results: [1x8 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×8 table
    Price      Delta       Gamma      Lambda       Rho       Theta      Vega     VegaLT
    ______    ________    ________    _______    _______    _______    ______    _______

    14.565    -0.72501    0.015172    -3.9822    -174.38    0.80043    26.545    0.25296
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Black

Create `Black` model object for `Cap`, `Floor`, or `Swaption` instrument

Description

Create and price a `Cap`, `Floor`, or `Swaption` instrument object with a `Black` model using this workflow:

1. Use `fininstrument` to create a `Cap`, `Floor`, or`Swaption` instrument object.
2. Use `finmodel` to specify a `Black` model object for the `Cap`, `Floor`, or `Swaption` instrument object.
3. Use `finpricer` to specify a `Black` pricing method for the `Cap`, `Floor`, or `Swaption` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Cap`, `Floor`, or `Swaption` instrument when using a `Black` model, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BlackModelObj = finmodel(ModelType,'Volatility',volatility_value)
BlackModelObj = finmodel( ___ ,Name,Value)
```

Description

`BlackModelObj = finmodel(ModelType,'Volatility',volatility_value)` creates a `Black` model object by specifying `ModelType` and sets the properties on page 11-3048 for the required name-value pair argument `Volatility`. For more information on a `Black` model, see “More About” on page 11-3050 and “Algorithms” on page 11-3050.

`BlackModelObj = finmodel( ___ ,Name,Value)` sets optional properties on page 11-3048 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `BlackModelObj = finmodel("Black",'Volatility',0.032,'Shift',0.002)` creates a `Black` model object. You can specify multiple name-value pair arguments.

Input Arguments

`ModelType` — Model type
string with value `"Black"` | character vector with value `'Black'`

Model type, specified as a string with the value of `"Black"` or a character vector with the value of `'Black'`.

Data Types: `char` | `string`

Black Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BlackModelObj = finmodel("Black",'Volatility',0.032,'Shift',0.002)`

Required Black Name-Value Pair Arguments

`Volatility` — Volatility value for the underlying asset
nonnegative numeric

Volatility value for the underlying asset, specified as the comma-separated pair consisting of `'Volatility'` and a scalar nonnegative numeric.

Data Types: `double`

Optional Black Name-Value Pair Argument

`Shift` — Shift in decimals for shifted Black model
`0` (no shift) (default) | positive decimal

Shift in decimals for the shifted Black model, specified as the comma-separated pair consisting of `'Shift'` and a scalar rate shift in positive decimals. Set this parameter to a positive rate shift in decimals to add a positive shift to the forward rate and strike, which effectively sets a negative lower bound for the forward rate. For example, a `Shift` value of `0.01` is equal to a 1% shift.

Data Types: `double`

Properties

`Volatility` — Volatility value
nonnegative numeric

Volatility value, returned as a scalar nonnegative numeric.

Data Types: `double`

`Shift` — Shift in decimals for shifted Black model
`0` (no shift) (default) | positive decimal

Shift in decimals for the shifted Black model, returned as a scalar rate shift in a positive decimal.

Data Types: `double`

Examples

Use Black Model and Black Pricer to Price Cap Instrument

This example shows the workflow to price a `Cap` instrument when you use a `Black` model and a `Black` pricing method.

Create Cap Instrument Object

Use `fininstrument` to create a `Cap` instrument object.

```
CapOpt = fininstrument("Cap",'Strike',.001,'Maturity',datetime(2019,1,30),'Reset',4,'Principal',1
```

```
CapOpt =
  Cap with properties:

                     Strike: 1.0000e-03
                   Maturity: 30-Jan-2019
                ResetOffset: 0
                      Reset: 4
                      Basis: 8
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "cap_option"
```

Create Black Model Object

Use `finmodel` to create a `Black` model object.

```
BlackModel = finmodel("Black",'Volatility',0.032,'Shift',0.002)
```

```
BlackModel =
  Black with properties:

    Volatility: 0.0320
         Shift: 0.0020
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 0
              Dates: [10x1 datetime]
              Rates: [10x1 double]
             Settle: 15-Sep-2018
       InterpMethod: "linear"
    ShortExtrapMethod: "next"
```

```
     LongExtrapMethod: "previous"
```

Create Black Pricer Object

Use `finpricer` to create a `Black` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackModel,'DiscountCurve',myRC)
```

```
outPricer =
  Black with properties:

            Model: [1x1 finmodel.Black]
    DiscountCurve: [1x1 ratecurve]
```

Price Cap Instrument

Use `price` to compute the price for the `Cap` instrument.

```
Price = price(outPricer,CapOpt)
```

```
Price = 0.1575
```

More About

Shifted Black Model

The shifted Black model is the same as the Black model, except that it models the movements of (F + *Shift*) as the underlying asset, instead of F (which is the forward rate in the case of caplets).

This model allows negative rates, with a fixed negative lower bound defined by the amount of shift; that is, the zero lower bound of the Black model is shifted.

Algorithms

Black Model

$$dF = \sigma_{Black} F dw$$

$$call = e^{-\gamma T}[FN(d_1) - KN(d_2)]$$

$$put = e^{-\gamma T}[KN(-d_2) - FN(-d_1)]$$

$$d_1 = \frac{\ln\left(\frac{F}{K}\right) + \left(\frac{\sigma_B^2}{2}\right)T}{\sigma_B\sqrt{T}}, \quad d_2 = d_1 - \sigma_B\sqrt{T}$$

$$\sigma_B = \sigma_{Black}$$

Here, F is the forward value and K is the strike.

Shifted Black Model

$$dF = \sigma_{Shifted\_Black}(F + Shift)dw$$

$$call = e^{-\gamma T}[(F + Shift)N(d_{s1}) - (K + Shift)N(d_{s2})]$$

$$put = e^{-\gamma T}[(K + Shift)N(-d_{s2}) - (F + Shift)N(-d_{s1})]$$

$$d_{s1} = \frac{\ln\left(\frac{F + Shift}{K + Shift}\right) + \left(\frac{\sigma_{sB}{}^2}{2}\right)T}{\sigma_{sB}\sqrt{T}}, \quad d_{s2} = d_{s1} - \sigma_{sB}\sqrt{T}$$

$$\sigma_{sB} = \sigma_{Shifted\_Black}$$

Here, $F+Shift$ is the forward value and $K+Shift$ is the strike for the shifted version.

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Work with Negative Interest Rates Using Objects” on page 2-22

CDSBlack

Create CDSBlack model object for CDSOption instrument

Description

Create and price a CDSOption instrument object with a CDSBlack model using this workflow:

1. Use fininstrument to create a CDSOption instrument object.
2. Use finmodel to specify a CDSBlack model object for the CDSOption instrument object.
3. Use finpricer to specify a CDSBlack pricing method for the CDSOption instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a CDSOption instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
CDSBlackModelObj = finmodel(ModelType,'
SpreadVolatility',spreadvolatility_value)
```

Description

`CDSBlackModelObj = finmodel(ModelType,'SpreadVolatility',spreadvolatility_value)` creates a CDSBlack model object by specifying ModelType and the required name-value pair argument SpreadVolatility to set the properties on page 11-3053 using name-value pair arguments. For example, `CDSBlackModelObj = finmodel("CDSBlack",'SpreadVolatility',0.052)` creates a CDSBlack model object.

Input Arguments

ModelType — Model type

string with value `"CDSBlack"` | character vector with value `'CDSBlack'`

Model type, specified as a string with the value of `"CDSBlack"` or a character vector with the value of `'CDSBlack'`.

Data Types: `char` | `string`

CDSBlack Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: `CDSBlackModelObj = finmodel("CDSBlack",'SpreadVolatility',0.052)`

`SpreadVolatility` — Spread volatility value
nonnegative numeric

Spread volatility value, specified as the comma-separated pair consisting of `'SpreadVolatility'` and a scalar nonnegative numeric.

Data Types: `double`

Properties

`SpreadVolatility` — Spread volatility value
nonnegative numeric

Spread volatility value, returned as a scalar nonnegative numeric.

Data Types: `double`

Examples

Use CDS Black Model and CDS Black Pricer to Price CDS Option Instrument

This example shows the workflow to price a `CDSOption` instrument when you use a `CDSBlack` model and a `CDSBlack` pricing method.

Create CDS Instrument Object

Use `fininstrument` to create a `CDS` instrument object as the underlying instrument.

```
CDS = fininstrument("CDS",'Maturity',datetime(2021,9,15),'ContractSpread',150,'Notional',100,'Nar
```

```
CDS =
  CDS with properties:

          ContractSpread: 150
                Maturity: 15-Sep-2021
                  Period: 4
                   Basis: 2
            RecoveryRate: 0.4000
    BusinessDayConvention: "actual"
                Holidays: NaT
        PayAccruedPremium: 1
                Notional: 100
                    Name: "CDS_instrument"
```

Create `defprobcurve` Object

Create a `defprobcurve` object using `defprobcurve`.

```
Settle = datetime(2020,9,20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities,'Basis',5)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                  Settle: 20-Sep-2020
                   Basis: 5
                   Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create CDSOption Instrument Object

Use `fininstrument` to create a `CDSOption` instrument object.

```
CDSOptionInst = fininstrument("CDSOption",'ExerciseDate',datetime(2021,8,15),'Strike',20,'CDS',CD
```

```
CDSOptionInst =
  CDSOption with properties:

              OptionType: "put"
                  Strike: 20
               Knockout: 0
    AdjustedForwardSpread: NaN
            ExerciseDate: 15-Aug-2021
                     CDS: [1x1 fininstrument.CDS]
                    Name: "CDSOption_option"
```

Create CDSBlack Model Object

Use `finmodel` to create a `CDSBlack` model object.

```
CDSBlackModel = finmodel("CDSBlack",'SpreadVolatility',.2)
```

```
CDSBlackModel =
  CDSBlack with properties:
```

```
    SpreadVolatility: 0.2000
```

Create CDSBlack Pricer Object

Use `finpricer` to create a `CDSBlack` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',CDSBlackModel,'DefaultProbabilityCurve',DefaultProbCurve
```

```
outPricer =
  CDSBlack with properties:

                      Model: [1x1 finmodel.CDSBlack]
              DiscountCurve: [1x1 ratecurve]
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price CDSOption Instrument

Use `price` to compute the price for the `CDSOption` instrument.

```
Price = price(outPricer,CDSOptionInst)
```

```
Price = 3.3016e-04
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

BlackScholes

Create `BlackScholes` model object for an `Asian`, `Barrier`, `DoubleBarrier`, `Lookback`, `PartialLookback`, `Spread`, `Vanilla`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument

Description

Create and price a `Vanilla`, `Lookback`, `PartialLookback`, `Barrier`, `DoubleBarrier` `Asian`, `Spread`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument object with a `BlackScholes` model using this workflow:

1. Use `fininstrument` to create a `Vanilla`, `Lookback`, `PartialLookback`, `Barrier`, `Asian`, `Spread`, `DoubleBarrier`, `Cliquet`, `Binary`, `Touch`, or `DoubleTouch` instrument object.
2. Use `finmodel` to specify the `BlackScholes` model object for a `Vanilla`, `Lookback`, `PartialLookback`, `Barrier`, `DoubleBarrier`, `Asian`, `Spread`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument object.
3. Use `finpricer` to specify a supported pricing method. For more information on the available pricing methods for the `Vanilla`, `Lookback`, `PartialLookback`, `Barrier`, `DoubleBarrier`, `Asian`, `Spread`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument object when using a `BlackScholes` model, see “Choose Instruments, Models, and Pricers” on page 1-53.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Vanilla`, `Lookback`, `PartialLookback`, `Barrier`, `DoubleBarrier`, `Asian`, `Spread`, `Touch`, `DoubleTouch`, or `Binary` instrument when using a `BlackScholes` model, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BlackScholesModelObj = finmodel(ModelType,'Volatility',volatility_value)
BlackScholesModelObj = finmodel( ___ ,Name,Value)
```

Description

`BlackScholesModelObj = finmodel(ModelType,'Volatility',volatility_value)` creates a `BlackScholes` model object by specifying `ModelType` and sets the properties on page 11-3057 for the required name-value pair argument `Volatility`.

`BlackScholesModelObj = finmodel( ___ ,Name,Value)` sets optional properties on page 11-3057 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `BlackScholesModelObj = finmodel("BlackScholes",'Volatility',0.032,'Correlation',Corr)` creates a `BlackScholes` model object. You can specify multiple name-value pair arguments.

Input Arguments

`ModelType` — Model type
string with value `"BlackScholes"` | character vector with value `'BlackScholes'`

Model type, specified as a string with the value of `"BlackScholes"` or a character vector with the value of `'BlackScholes'`.

Data Types: `char` | `string`

`BlackScholes` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BlackScholesModelObj = finmodel("BlackScholes",'Volatility',0.032,'Correlation',Corr)`

Required BlackScholes Name-Value Pair Arguments

`Volatility` — Volatility value
nonnegative numeric

Volatility value, specified as the comma-separated pair consisting of `'Volatility'` and a scalar nonnegative numeric.

Data Types: `double`

Optional BlackScholes Name-Value Pair Arguments

`Correlation` — Correlation between underlying asset prices
1 (default) | semidefinite matrix

Correlation between the underlying asset prices, specified as the comma-separated pair consisting of `'Correlation'` and a semidefinite matrix. For more information on creating a positive semidefinite matrix, see `nearcorr`.

Data Types: `double`

Properties

`Volatility` — Volatility value
nonnegative numeric

Volatility value, returned as a scalar nonnegative numeric.

Data Types: `double`

`Corrrelation` — Correlation between underlying assets
1 (default) | semi-definite matrix

Correlation between underlying assets, returned as a semi-definite matrix.

Data Types: `double`

Examples

Use Black-Scholes Model and Turnbull-Wakeman Pricer to Price Asian Instrument

This example shows the workflow to price an `Asian` instrument when you use a `BlackScholes` model and a `TurnbullWakeman` pricing method.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',105,'OptionType',"pu
```

```
AsianOpt =
  Asian with properties:

          OptionType: "put"
              Strike: 105
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.28)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2800
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
              Dates: 15-Sep-2023
              Rates: 0.0350
             Settle: 15-Sep-2018
```

```
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create TurnbullWakeman Pricer Object

Use `finpricer` to create a `TurnbulllWakeman` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',100,
```

```
outPricer =
  TurnbullWakeman with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0
     DividendType: "continuous"
```

Price Asian Instrument

Use `price` to compute the price and sensitivities for the `Asian` instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 11.2249

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma      Lambda      Vega       Theta       Rho
    _____    ________    ________    _______    _____    ________    ______

    11.225    -0.38072    0.01087    -3.3917    44.242    -0.5256    -116.88
```

Use Black-Scholes Model and Asset Monte-Carlo Pricer to Price DoubleBarrier Instrument

This example shows the workflow to price a `DoubleBarrier` instrument when you use a `BlackScholes` model and an `AssetMonteCarlo` pricing method.

Create DoubleBarrier Instrument Object

Use `fininstrument` to create a `DoubleBarrier` instrument object.

```
DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',100,'ExerciseDate',datetime(2020,8,15),
```

```
DoubleBarrierOpt =
  DoubleBarrier with properties:

       OptionType: "call"
           Strike: 100
     BarrierValue: [110 80]
    ExerciseStyle: "american"
     ExerciseDate: 15-Aug-2020
      BarrierType: "dko"
           Rebate: [0 0]
             Name: "doublebarrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes","Volatility",.3)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2017,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 12
             Dates: 15-Sep-2023
             Rates: 0.0350
            Settle: 15-Sep-2017
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonetCarlo` Pricer Object

Use `finpricer` to create an `AssetMonetCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
ExerciseDate = datetime(2020,08,15);
Settle = datetime(2017,9,15);
outPricer = finpricer("AssetMonteCarlo","DiscountCurve",myRC,"Model",BlackScholesModel,'SpotPrice
```

Price DoubleBarrier Instrument

Use `price` to compute the price and sensitivities for the `DoubleBarrier` instrument.

```
[Price, outPR] = price(outPricer,DoubleBarrierOpt,["all"])
```

```
Price = 6.9667

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Lambda      Rho       Theta      Vega
    ______    _______    __________    ______    _______    ______    _______

    6.9667    0.26875    -0.096337    3.8576    0.39855    9.5406    -1.2907
```

Use Black-Scholes Model and Asset Tree Pricer to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method.

Create Vanilla Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2019,5,1),'Strike',29,'OptionType','
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 01-May-2019
           Strike: 29
             Name: "vanilla_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2500
    Correlation: 1
```

Create ratecurve Object

Create a flat ratecurve object using ratecurve.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2020,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2020
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use finpricer to create an AssetTree pricer object for a LR equity tree and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
LRPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',30,'
```

```
LRPricer =
  LRTree with properties:

    InversionMethod: PP1
             Strike: 30
               Tree: [1x1 struct]
        NumPeriods: 15
              Model: [1x1 finmodel.BlackScholes]
      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 30
       DividendType: "continuous"
      DividendValue: 0
          TreeDates: [02-Feb-2018 08:00:00    06-Mar-2018 16:00:00    ...    ]
```

Price Vanilla Instrument

Use price to compute the price and sensitivities for the Vanilla instrument.

```
[Price, outPR] = price(LRPricer,VanillaOpt,"all")
```

```
Price = 2.2542
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Vega     Lambda      Rho       Theta
    ______    ________    _________    ______    ______    _______    ________

    2.2542    -0.33628    0.044039    12.724    -4.469    -16.433    -0.76073
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

BraceGatarekMusiela

Create `BraceGatarekMusiela` model object for `Cap`, `Floor`, `FixedBond`, `FloatBond`, `FloatBondOption`, `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument

Description

Create and price a `Cap`, `Floor`, `FloatBond`, `FloatBondOption`, `FixedBond`, `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object with a `BraceGatarekMusiela` model using this workflow:

1. Use `fininstrument` to create a `Cap`, `Floor`, `FixedBond`, `FloatBond`, `FloatBondOption` `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.
2. Use `finmodel` to specify a `BraceGatarekMusiela` model object for the `Cap`, `Floor`, `FixedBond`, `FloatBond`, `FloatBondOption`, `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.
3. Use `finpricer` to specify an `IRMonteCarlo` pricing method for a `Cap`, `Floor`, `FixedBond`, `FloatBond`, `FloatBondOption`, `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Cap`, `Floor`, `FixedBond`, `FloatBond`, `FloatBondOption`, `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BraceGatarekMusielaModelObj = finmodel(ModelType,Volatility=volatility_value,
Correlation=correlation_value)
BraceGatarekMusielaModelObj = finmodel( ___ ,Name=Value)
```

Description

`BraceGatarekMusielaModelObj = finmodel(ModelType,Volatility=volatility_value, Correlation=correlation_value)` creates a `BraceGatarekMusiela` model object by specifying `ModelType` and the required name-value arguments `Volatility` and `Correlation` to set the properties on page 11-3066.

`BraceGatarekMusielaModelObj = finmodel( ___ ,Name=Value)` sets optional properties on page 11-3066 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `BraceGatarekMusielaModelObj =`

`finmodel("BraceGatarekMusiela",Volatility=VolFunc,Correlation=Correlation,Period=1)` creates a `BraceGatarekMusiela` model object. You can specify multiple name-value arguments.

Input Arguments

`ModelType` — Model type

string with value `"BraceGatarekMusiela"` | character vector with value `'BraceGatarekMusiela'`

Model type, specified as a string with the value of `"BraceGatarekMusiela"` or a character vector with the value of `'BraceGatarekMusiela'`.

Data Types: `char` | `string`

`BraceGatarekMusiela` Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `BraceGatarekMusielaModelObj = finmodel("BraceGatarekMusiela",Volatility=VolFunc,Correlation=Correlation,Period=1)`

`Volatility` — Volatility

function handle

Volatility, specified as `Volatility` and an (`NumRates-1`)-by-`1` cell array of function handles. Each function handle must take time as an input and return a scalar volatility.

Data Types: `double` | `cell`

`Correlation` — Correlation matrix

matrix

Correlation matrix, specified as `Correlation` and a (`NumRates-1`)-by-(`NumRates-1`) correlation matrix.

Data Types: `double`

Optional `BraceGatarekMusiela` Name-Value Arguments

`NumFactors` — Number of Brownian factors

`NaN` (default) | numeric

Number of Brownian factors, specified as `NumFactors` and a scalar numeric. The default is `NaN` which means that `NumFactors` is equal to the number of rates.

Data Types: `double`

`Period` — Period of forward rates

`2` (default) | numeric

Period of forward rates, specified as `Period` and a scalar numeric. The default is `2`, meaning forward rates are spaced at `0`, `.5`, `1`, `1.5`, and so on.

Data Types: `double`

Properties

`Volatility` — Volatility
function handle

Volatility, returned as a (`NumRates-1`)-by-`1` cell array of function handles.

Data Types: `double` | `cell`

`Correlation` — Correlation matrix
matrix

Correlation matrix, returned as a (`NumRates-1`)-by-(`NumRates-1`) correlation matrix.

Data Types: `double`

`NumFactors` — Number of Brownian factors
`NaN` (default) | numeric

Number of Brownian factors, returned as a scalar numeric.

Data Types: `double`

`Period` — Period of forward rates
`2` (default) | numeric

Period of forward rates, returned as a scalar numeric.

Data Types: `double`

Examples

Use BraceGatarekMusiela Model and IRMonteCarlo Pricer to Price Floor Instrument

This example shows the workflow to price a `Floor` instrument when using a `BraceGatarekMusiela` model and an `IRMonteCarlo` pricing method.

Create Floor Instrument Object

Use `fininstrument` to create a `Floor` instrument object.

```
FloorOpt = fininstrument("Floor",Maturity=datetime(2022,9,15),Strike=0.05,Reset=1,Name="floor_op
```

```
FloorOpt =
  Floor with properties:

                     Strike: 0.0500
                   Maturity: 15-Sep-2022
                ResetOffset: 0
                      Reset: 1
                      Basis: 0
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
```

```
                        Name: "floor_option"
```

Create BraceGatarekMusiela Model Object

Use `finmodel` to create a `LinearGaussian2F` model object.

```
BGMVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
BGMVolParams = [.3 -.02 .7 .14];
numRates = 20;
VolFunc(1:numRates-1) = {@(t) BGMVolFunc(BGMVolParams,t)};
Beta = .08;
CorrFunc = @(i,j,Beta) exp(-Beta*abs(i-j));
Correlation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),Beta);
BGM = finmodel("BraceGatarekMusiela",Volatility=VolFunc,Correlation=Correlation,Period=1);
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [9x1 datetime]
               Rates: [9x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create `IRMonteCarlo` Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",Model=BGM,DiscountCurve=myRC,SimulationDates=ZeroDates)
```

```
outPricer =
  BGMMonteCarlo with properties:

         NumTrials: 1000
     RandomNumbers: []
     DiscountCurve: [1x1 ratecurve]
   SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jan-2021    ...    ]
             Model: [1x1 finmodel.BraceGatarekMusiela]
```

Price Floor Instrument

Use `price` to compute the price and sensitivities for the `Floor` instrument.

```
[Price,outPR] = price(outPricer,FloorOpt,["all"])
```

```
Price = 14.7975

outPR =
  priceresult with properties:

       Results: [1x3 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×3 table
    Price      Delta     Gamma
    _____    ________    _____

    14.797    -398.43    1399.5
```

More About

BraceGatarekMusiela Model

The BraceGatarekMusiela (BGM) model, also known as the LIBOR market model, is a financial model of interest rates.

The BGM model is based on the Heath-Jarrow-Morton (HJM) forward rate approach, it builds a process for LIBOR interest rates, assuming a conditional lognormal process for LIBOR. The BGM model is an interest-rate model that differs from short rate models in that it evolves a set of discrete forward rates.

Specifically, the lognormal BGM model specifies the following diffusion equation for each forward rate

$$\frac{dF_i(t)}{F_i} = -\mu_i dt + \sigma_i(t) dW_i$$

where:

W is an N-dimensional geometric Brownian motion with

$$dW_i(t) dW_j(t) = \rho_{ij}$$

The BGM model relates drifts of the forward rates based on no-arbitrage arguments. Specifically, under the Spot LIBOR measure, drifts are expressed as

$$\mu_i(t) = -\sigma_i(t) \sum_{j=q(t)}^{i} \frac{\tau_j \rho_{i,j} \sigma_j(t) F_j(t)}{1 + \tau_j F_j(t)}$$

where:

$\rho_{i,j}$ represents the input argument `Correlation`.

$\sigma_j(t)$ represents the input argument `Volatility`.

$F_j(t)$ represents the computation of the `ZeroCurve`.

τ_i is the time fraction associated with the i th forward rate.

q(t) is an index defined by the relation

$$T_{q(t)-1} < t < T_{q(t)}$$

and the Spot LIBOR numeraire is defined as

$$B(t) = P(t, T_{q(t)}) \prod_{n=0}^{q(t)-1} (1 + \tau_n F_n(T_n))$$

Version History

Introduced in R2021b

See Also

Functions

`SABRBraceGatarekMusiela` | `fininstrument` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

SABRBraceGatarekMusiela

Create SABRBraceGatarekMusiela model object for Cap, Floor, FixedBond, FloatBond, FloatBondOption, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument

Description

Create and price a Cap, Floor, FloatBond, FloatBondOption, FixedBond, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object with a SABRBraceGatarekMusiela model using this workflow:

1. Use fininstrument to create a Cap, Floor, FixedBond, FloatBond, FloatBondOption FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object.
2. Use finmodel to specify a SABRBraceGatarekMusiela model object for the Cap, Floor, FixedBond, FloatBond, FloatBondOption, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object.
3. Use finpricer to specify an IRMonteCarlo pricing method for a Cap, Floor, FixedBond, FloatBond, FloatBondOption, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a Cap, Floor, FixedBond, FloatBond, FloatBondOption, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
SABRBraceGatarekMusielaModelObj = finmodel(ModelType,Alpha=alpha_value,
Beta=beta_value,VolatilityofVolatility=volatilityofvolatility_value,
FwdFwdCorrelation=fwdfwdcorrelation_value,
VolVolCorrelation=volvolcorrelation_value)
```

Description

`SABRBraceGatarekMusielaModelObj = finmodel(ModelType,Alpha=alpha_value, Beta=beta_value,VolatilityofVolatility=volatilityofvolatility_value, FwdFwdCorrelation=fwdfwdcorrelation_value, VolVolCorrelation=volvolcorrelation_value)` creates a classic SABRBraceGatarekMusiela model object with null forward to volatility correlation by specifying ModelType and the required name-value arguments Alpha, Beta, VolatilityofVolatility, FwdFwdCorrelation, and VolVolCorrelation to set properties on page 11-3073 using name-

value arguments. For example, `SABRBraceGatarekMusielaModelObj = finmodel("SABRBraceGatarekMusiela",Alpha=Alpha,Beta=Beta,VolatilityofVolatility=VolVolFunc,FwdFwdCorrelation=FwdFwdCorrelation, VolVolCorrelation=VolVolCorrelation)` creates a classic `SABRBraceGatarekMusiela` model object with null forward to volatility correlation.

`SABRBraceGatarekMusielaModelObj = finmodel( ___ ,Name=Value)` sets optional properties on page 11-3073 using additional name-value arguments in addition to the required arguments in the previous syntax. You can specify multiple name-value arguments. For example, you can use name-value arguments to create the following variations of the `SABRBraceGatarekMusiela` model:

- To create a classic `SABRBraceGatarekMusiela` model object, use the `FwdVolCorrelation` name-value pair argument: `SABRBraceGatarekMusielaModelObj = finmodel("SABRBraceGatarekMusiela",Alpha=Alpha,Beta=Beta,VolatilityofVolatility=VolVolFunc,FwdFwdCorrelation=FwdFwdCorrelation,VolVolCorrelation=VolVolCorrelation,FwdVolCorrelation=FwdVolCorrelation)`
- To create a classic `SABRBraceGatarekMusiela` model object in Rebonato parametric form with null forward-to-volatility correlation, use the `Volatility` name-value pair argument: `SABRBraceGatarekMusielaModelObj = finmodel("SABRBraceGatarekMusiela",Alpha=Alpha,Beta=Beta,VolatilityofVolatility=VolVolFunc,Volatility=VolFunc,FwdFwdCorrelation=FwdFwdCorrelation,VolVolCorrelation=VolVolCorrelation)`
- To create a classic `SABRBraceGatarekMusiela` model object in Rebonato parametric form with `FwdVolCorrelation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),.02)`, use the `Volatility` and `FwdVolCorrelation` name-value arguments: `SABRBraceGatarekMusielaModelObj = finmodel("SABRBraceGatarekMusiela",Alpha=Alpha,Beta=Beta,VolatilityofVolatility=VolVolFunc,Volatility=VolFunc,FwdFwdCorrelation=FwdFwdCorrelation,VolVolCorrelation=VolVolCorrelation)`

Input Arguments

`ModelType` — Model type

string with value `"SABRBraceGatarekMusiela"` | character vector with value `'SABRBraceGatarekMusiela'`

Model type, specified as a string with the value of `"SABRBraceGatarekMusiela"` or a character vector with the value of `'SABRBraceGatarekMusiela'`.

Data Types: `char` | `string`

`SABRBraceGatarekMusiela` Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `SABRBraceGatarekMusielaModelObj = finmodel("SABRBraceGatarekMusiela",Alpha=Alpha,Beta=Beta,VolatilityofVolatility=VolVolFunc,FwdFwdCorrelation=FwdFwdCorrelation, VolVolCorrelation=VolVolCorrelation)`

`Alpha` — Initial SABR volatilities for each forward rate maturity

positive numeric

Initial SABR volatilities for each forward rate maturity, specified as `Alpha` and an (`NumRates-1`)-by-`1` vector of positive numeric values.

Data Types: `double`

`Beta` — SABR exponent parameters for each forward rate maturity
numeric between `0` and `1`

SABR exponent parameters for each forward rate maturity, specified as `Beta` and an (`NumRates-1`)-by-`1` vector of numeric values between `0` and `1`.

Data Types: `double`

`VolatilityofVolatility` — Variation in volatility
function handle

Variation in volatility, specified as `VolatilityofVolatility` and an (`NumRates-1`)-by-`1` cell array of function handles. Each function handle must take time as an input and return a scalar volatility of volatility that must be positive numeric.

Data Types: `double` | `cell`

`FwdFwdCorrelation` — Correlation matrix for forward rates
numeric between `-1` and `1`

Correlation matrix for forward rates, specified as `FwdFwdCorrelation` and a (`NumRates-1`)-by-(`NumRates-1`) correlation matrix.

Data Types: `double`

`VolVolCorrelation` — Correlation matrix for volatilities
numeric between `-1` and `1`

Correlation matrix for volatilities, specified as `VolVolFwdCorrelation` and an (`NumRates-1`)-by-(`NumRates-1`) correlation matrix.

Data Types: `double`

Optional SABRBraceGatarekMusiela Name-Value Arguments

`Volatility` — Volatility
simulate stochastic volatilities without volatility functions (default) | function handle

Volatility, specified as `Volatility` and an (`NumRates-1`)-by-`1` cell array of function handles. Specify these optional volatility function handles to use the Rebonato (2009) parametric form, which simulates stochastic volatilities with deterministic volatility functions and stochastic correction terms. Each function handle must take time as an input and return a scalar volatility that must be positive numeric.

Data Types: `double`

`FwdVolCorrelation` — Correlation matrix between forward rates and volatilities
(`NumRates-1`)-by-(`NumRates-1`) matrix of zeros (default) | numeric between `-1` and `1`

Correlation matrix between forward rates and volatilities, specified as `FwdVolCorrelation` and an (`NumRates-1`)-by-(`NumRates-1`) correlation matrix. The diagonal elements of the matrix are the SABR Rho parameters.

Data Types: `double`

`Period` — Period of forward rates
2 (default) | numeric

Period of forward rates, specified as `Period` and a scalar numeric. The default is `2`, meaning forward rates are spaced at `0`, `.5`, `1`, `1.5`, and so on.

Data Types: `double`

Properties

`Alpha` — Initial SABR volatilities for each forward rate maturity
positive numeric

Initial SABR volatilities for each forward rate maturity, returned as an `NumRates-1`-by-`1` vector of positive numeric values.

Data Types: `double`

`Beta` — SABR exponent parameters for each forward rate maturity
numeric between `0` and `1`

SABR exponent parameters for each forward rate maturity, returned as an `NumRates-1`-by-`1` vector of numeric values between `0` and `1`.

Data Types: `double`

`VolatilityofVolatility` — Variation in volatility
function handle

Variation in volatility, returned as an `NumRates-1`-by-`1` cell array of function handles.

Data Types: `double` | `cell`

`FwdFwdCorrelation` — Correlation matrix for forward rates
numeric between `-1` and `1`

Correlation matrix for forward rates, returned as an (`NumRates-1`)-by-(`NumRates-1`) correlation matrix.

Data Types: `double`

`VolVolCorrelation` — Correlation matrix for volatilities
numeric between `-1` and `1`

Correlation matrix for volatilities, returned as an (`NumRates-1`)-by-(`NumRates-1`) correlation matrix.

Data Types: `double`

`Volatility` — Volatility
simulate stochastic volatilities without volatility functions (default) | function handle

Volatility, returned as an `NumRates-1`-by-`1` cell array of function handles.

Data Types: `double`

FwdVolCorrelation — Correlation matrix between forward rates and volatilities
(NumRates-1)-by-(NumRates-1) matrix of zeros (default) | numeric between -1 and 1

Correlation matrix between forward rates and volatilities, returned as an (NumRates-1)-by-(NumRates-1) correlation matrix.

Data Types: double

Period — Period of forward rates
2 (default) | numeric

Period of forward rates, returned as a scalar numeric.

Data Types: double

Examples

Use SABR-BraceGatarekMusiela Model with Null Forward-Volatility Correlation and IRMonteCarlo Pricer to Price Cap Instrument

This example shows the workflow to price a Cap instrument when you use a SABRBraceGatarekMusiela model with null Forward-Volatility correlation and an IRMonteCarlo pricing method.

Create Cap Instrument Object

Use fininstrument to create a Cap instrument object.

```
CapOpt = fininstrument("Cap",Maturity=datetime(2021,7,1),Strike=0.035,Name="cap_option")
```

```
CapOpt =
  Cap with properties:

                    Strike: 0.0350
                  Maturity: 01-Jul-2021
               ResetOffset: 0
                     Reset: 1
                     Basis: 0
                 Principal: 100
            ProjectionCurve: [0x0 ratecurve]
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                  Holidays: NaT
                      Name: "cap_option"
```

Create Classic SABRBraceGatarekMusiela Model Object with Null Forward-Volatility Correlation

Use finmodel to create a SABRBraceGatarekMusiela model object that is a classic SABR-BGM model with null Forward-Volatility correlation.

```
Alpha = [0.4;0.34;0.31;0.28];
Beta = [0.5;0.5;0.5;0.5];

SABRBGMVolVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
```

```
SABRBGMVolVolParams = [.3 -.02 .7 .14];

numRates = 5;
VolVolFunc(1:numRates-1,1) = {@(t) SABRBGMVolVolFunc(SABRBGMVolVolParams,t)};

CorrFunc = @(i,j,B) exp(-B*abs(i-j));
FwdFwdCorrelation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),.08);
VolVolCorrelation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),.04);

SABRBGM_NF = finmodel("SABRBraceGatarekMusiela",Alpha=Alpha,Beta=Beta,VolatilityofVolatility=Vol
```

```
SABRBGM_NF =
  SABRBraceGatarekMusiela with properties:

                   Period: 2
                    Alpha: [4x1 double]
                     Beta: [4x1 double]
               Volatility: {4x1 cell}
    VolatilityofVolatility: {4x1 cell}
         FwdFwdCorrelation: [4x4 double]
         VolVolCorrelation: [4x4 double]
         FwdVolCorrelation: [4x4 double]
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `IRMonteCarlo` Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
simDates = datetime(2019,7,1)+calmonths(0:6:24);
outPricer = finpricer("IRMonteCarlo",Model=SABRBGM_NF,DiscountCurve=myRC,SimulationDates=simDates
```

```
outPricer =
  SABRBGMMonteCarlo with properties:
```

```
          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jul-2020    ...    ]
              Model: [1x1 finmodel.SABRBraceGatarekMusiela]
```

Price Cap Instrument

Use `price` to compute the price and sensitivities for the `Cap` instrument.

```
[Price,outPR] = price(outPricer,CapOpt,["all"])
```

```
Price = 0.8867

outPR =
  priceresult with properties:

       Results: [1x3 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×3 table
    Price     Delta      Gamma
    ______    ______    _______

    0.8867    118.82    -9486.5
```

You can access the simulated interest-rate `Paths` in the `PricerData` output.

```
outPR.PricerData
```

```
ans = struct with fields:
    SimulationTimes: [6x1 timetable]
              Paths: [6x8x1000 double]
      RandomNumbers: [1x1 struct]
```

Use Classic SABR-BraceGatarekMusiela Model and IRMonteCarlo Pricer to Price Floor Instrument

This example shows the workflow to price a `Floor` instrument when you use a classic `SABRBraceGatarekMusiela` model and an `IRMonteCarlo` pricing method.

Create `Floor` Instrument Object

Use `fininstrument` to create a `Floor` instrument object.

```
FloorOpt = fininstrument("Floor",Maturity=datetime(2021,7,1),Strike=0.05,Reset=1,Name="floor_opti
```

```
FloorOpt =
  Floor with properties:
```

```
                    Strike: 0.0500
                  Maturity: 01-Jul-2021
               ResetOffset: 0
                     Reset: 1
                     Basis: 0
                 Principal: 100
           ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                  Holidays: NaT
                      Name: "floor_option"
```

Create Classic SABRBraceGatarekMusiela Model Object

Use `finmodel` to create a classic `SABRBraceGatarekMusiela` model object.

```
Alpha = [0.4;0.34;0.31;0.28];
Beta = [0.5;0.5;0.5;0.5];

SABRBGMVolVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
SABRBGMVolVolParams = [.3 -.02 .7 .14];

numRates = 5;
VolVolFunc(1:numRates-1,1) = {@(t) SABRBGMVolVolFunc(SABRBGMVolVolParams,t)};

CorrFunc = @(i,j,B) exp(-B*abs(i-j));
FwdFwdCorrelation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),.08);
VolVolCorrelation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),.04);
SABRRho = [0.0005;0.0006;0.0060;0.0055];
FwdVolCorrelation = diag(SABRRho);

SABRBGM_Classic = finmodel("SABRBraceGatarekMusiela",Alpha=Alpha,Beta=Beta,VolatilityofVolatility
```

```
SABRBGM_Classic =
  SABRBraceGatarekMusiela with properties:

                    Period: 2
                     Alpha: [4x1 double]
                      Beta: [4x1 double]
                Volatility: {4x1 cell}
    VolatilityofVolatility: {4x1 cell}
         FwdFwdCorrelation: [4x4 double]
         VolVolCorrelation: [4x4 double]
         FwdVolCorrelation: [4x4 double]
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
simDates = datetime(2019,7,1)+calmonths(0:6:24);
outPricer = finpricer("IRMonteCarlo",Model=SABRBGM_Classic,DiscountCurve=myRC,SimulationDates=sim
```

```
outPricer =
  SABRBGMMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jul-2020    ...    ]
              Model: [1x1 finmodel.SABRBraceGatarekMusiela]
```

Price Floor Instrument

Use `price` to compute the price and sensitivities for the `Floor` instrument.

```
[Price,outPR] = price(outPricer,FloorOpt,["all"])
```

```
Price = 11.6086
```

```
outPR =
  priceresult with properties:

       Results: [1x3 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×3 table
    Price      Delta     Gamma
    ______    _______    _____

    11.609    -171.15    13201
```

You can access the simulated interest-rate `Paths` in the `PricerData` output.

```
outPR.PricerData
```

```
ans = struct with fields:
    SimulationTimes: [6x1 timetable]
              Paths: [6x8x1000 double]
      RandomNumbers: [1x1 struct]
```

Use SABR-BraceGatarekMusiela Model (Rebonato Parametric Form with Null Forward-Volatility Correlation) and IRMonteCarlo Pricer to Price Fixed Bond Instrument

This example shows the workflow to price a `FixedBond` instrument when you use a `SABRBraceGatarekMusiela` model in Rebonato parametric form with null Forward-Volatility correlation and an `IRMonteCarlo` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",Maturity=datetime(2021,7,1),CouponRate=0.021,Period=2,Basis=1,P
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0210
                     Period: 2
                      Basis: 1
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
              FirstCouponDate: NaT
               LastCouponDate: NaT
                    StartDate: NaT
                     Maturity: 01-Jul-2021
                         Name: "fixed_bond_instrument"
```

Create Rebonato Form with Null Forward-Volatility Correlation SABRBraceGatarekMusiela Model Object

Use `finmodel` to create a `SABRBraceGatarekMusiela` model object that is a SABR-BGM model in Rebonato parametric form with null Forward-Volatility correlation.

```
Alpha = [0.4;0.34;0.31;0.28];
Beta = [0.5;0.5;0.5;0.5];
numRates = 5;

SABRBGMVolVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
SABRBGMVolVolParams = [.3 -.02 .7 .14];

SABRBGMVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
SABRBGMVolParams = [.2 -.01 .8 .16];
VolFunc(1:numRates-1,1) = {@(t) SABRBGMVolFunc(SABRBGMVolParams,t)};

VolVolFunc(1:numRates-1,1) = {@(t) SABRBGMVolVolFunc(SABRBGMVolVolParams,t)};
```

```
CorrFunc = @(i,j,B) exp(-B*abs(i-j));
FwdFwdCorrelation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),.08);
VolVolCorrelation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),.04);

SABRBGM_Rebonato = finmodel("SABRBraceGatarekMusiela",Alpha=Alpha,Beta=Beta,VolatilityofVolatili
```

```
SABRBGM_Rebonato =
  SABRBraceGatarekMusiela with properties:

                   Period: 2
                    Alpha: [4x1 double]
                     Beta: [4x1 double]
               Volatility: {4x1 cell}
    VolatilityofVolatility: {4x1 cell}
         FwdFwdCorrelation: [4x4 double]
         VolVolCorrelation: [4x4 double]
         FwdVolCorrelation: [4x4 double]
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `IRMonteCarlo` Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
simDates = datetime(2019,7,1)+calmonths(0:6:24);
outPricer = finpricer("IRMonteCarlo",Model=SABRBGM_Rebonato,DiscountCurve=myRC,SimulationDates=si
```

```
outPricer =
  SABRBGMMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
```

```
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jul-2020    ...    ]
              Model: [1x1 finmodel.SABRBraceGatarekMusiela]
```

Price FixedBond Instrument

Use `price` to compute the price and sensitivities for the `FixedBond` instrument.

```
[Price,outPR] = price(outPricer,FixB,["all"])
```

```
Price = 103.5433

outPR =
  priceresult with properties:

       Results: [1x3 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×3 table
    Price     Delta     Gamma
    ______    ______    _____

    103.54    -253.5    628.2
```

You can access the simulated interest-rate `Paths` in the `PricerData` output.

```
outPR.PricerData
```

```
ans = struct with fields:
    SimulationTimes: [6x1 timetable]
              Paths: [6x8x1000 double]
      RandomNumbers: [1x1 struct]
```

Use SABR-BraceGatarekMusiela Model (Rebonato Parameteric Form) and IRMonteCarlo Pricer to Price Float Bond Instrument

This example shows the workflow to price a `FloatBond` instrument when you use a `SABRBraceGatarekMusiela` model in Rebonato parametric form and an `IRMonteCarlo` pricing method.

Create FloatBond Instrument Object

Use `fininstrument` to create a `FloatBond` instrument object.

```
FloatB = fininstrument("FloatBond",Maturity=datetime(2021,7,1),Spread=0.025,Reset=2,Basis=1,Princ
```

```
FloatB =
  FloatBond with properties:

                    Spread: 0.0250
            ProjectionCurve: [0x0 ratecurve]
```

```
                ResetOffset: 0
                      Reset: 2
                      Basis: 1
                EndMonthRule: 0
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
          LatestFloatingRate: NaN
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jul-2021
                        Name: "float_bond_instrument"
```

Create Rebonato Form SABRBraceGatarekMusiela Model Object

Use `finmodel` to create a `SABRBraceGatarekMusiela` model object that is a SABR-BGM model in Rebonato parametric form.

```
Alpha = [0.4;0.34;0.31;0.28];
Beta = [0.5;0.5;0.5;0.5];
numRates = 5;

SABRBGMVolVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
SABRBGMVolVolParams = [.3 -.02 .7 .14];

SABRBGMVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
SABRBGMVolParams = [.2 -.01 .8 .16];
VolFunc(1:numRates-1,1) = {@(t) SABRBGMVolFunc(SABRBGMVolParams,t)};

VolVolFunc(1:numRates-1,1) = {@(t) SABRBGMVolVolFunc(SABRBGMVolVolParams,t)};

CorrFunc = @(i,j,B) exp(-B*abs(i-j));
FwdFwdCorrelation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),.08);
VolVolCorrelation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),.04);
SABRRho = [0.0005;0.0006;0.0060;0.0055];
FwdVolCorrelation = diag(SABRRho);

SABRBGM_Rebonato_param = finmodel("SABRBraceGatarekMusiela",Alpha=Alpha,Beta=Beta,VolatilityofVo
```

```
SABRBGM_Rebonato_param =
  SABRBraceGatarekMusiela with properties:

                   Period: 2
                    Alpha: [4x1 double]
                     Beta: [4x1 double]
               Volatility: {4x1 cell}
    VolatilityofVolatility: {4x1 cell}
         FwdFwdCorrelation: [4x4 double]
         VolVolCorrelation: [4x4 double]
         FwdVolCorrelation: [4x4 double]
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
simDates = datetime(2019,7,1)+calmonths(0:6:24);
outPricer = finpricer("IRMonteCarlo",Model=SABRBGM_Rebonato_param,DiscountCurve=myRC,SimulationDa
```

```
outPricer =
  SABRBGMMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jul-2020    ...    ]
              Model: [1x1 finmodel.SABRBraceGatarekMusiela]
```

Price FloatBond Instrument

Use `price` to compute the price and sensitivities for the `FloatBond` instrument.

```
[Price,outPR] = price(outPricer,FloatB,["all"])
```

```
Price = 106.1830
```

```
outPR =
  priceresult with properties:

       Results: [1x3 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×3 table
    Price      Delta     Gamma
```

```
    ______    _______    ______

    106.18    -9.2496    16.927
```

You can access the simulated interest-rate `Paths` in the `PricerData` output.

```
outPR.PricerData
```

```
ans = struct with fields:
    SimulationTimes: [6x1 timetable]
              Paths: [6x8x1000 double]
      RandomNumbers: [1x1 struct]
```

More About

SABR-BGM Model

The SABR-BGM model combines the BGM model and the SABR model by introducing the SABR β_k exponent and the SABR volatility $\alpha_k(t)$ to the BGM forward rate SDE.

$$dF_k(t) = \mu_k(t)dt + \alpha_k {F_k}^{\beta_k}(t)dW_{F_k}(t)$$

$$d\alpha_k(t) = \eta_k(t)dt + \nu_k(t)\alpha_k(t)dW_{\alpha_k}(t)$$

$$\mu_k(t) = \alpha_k(t){F_k}^{\beta_k}(t) \sum_{\gamma(t)\le l\le k} \frac{\rho_{l,k}\tau_l\alpha_l(t){F_l}^{\beta_l}(t)}{1+\tau_l F_l(t)}dt$$

$$\eta_k(t) = \phi_{k,k}\nu_k(t)\alpha_k(t) \sum_{\gamma(t)\le l\le k} \frac{\rho_{l,k}\tau_l\alpha_l(t){F_l}^{\beta_l}(t)}{1+\tau_l F_l(t)}dt$$

$$T_{\gamma(t)-1} \le t < T_{\gamma(t)}$$

SABR-BGM Model

- The volatility-of-volatility function $\nu_k(t)$ is a deterministic function of time.
- The correlation matrix is $2N$-by-$2N$:

$$\mathbf{E}\left[dW_{F_k}(t)dW_{F_l}(t)\right] = \rho_{k,l}dt$$
$$\mathbf{E}\left[dW_{\alpha_k}(t)dW_{\alpha_l}(t)\right] = \omega_{k,l}dt$$
$$\mathbf{E}\left[dW_{F_k}(t)dW_{\alpha_l}(t)\right] = \phi_{k,l}dt$$
$$\phi_{k,k}dt = \rho_k dt$$

$$\Pi = \begin{bmatrix} \rho & \phi \\ \phi^T & \omega \end{bmatrix} = \begin{bmatrix} \rho_{1,1} & \cdots & \rho_{1,N} & \phi_{1,1} & \cdots & \phi_{1,N} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \rho_{N,1} & \cdots & \rho_{N,N} & \phi_{N,1} & \cdots & \phi_{N,N} \\ \phi_{1,1} & \cdots & \phi_{N,1} & \omega_{1,1} & \cdots & \omega_{1,N} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \phi_{1,N} & \cdots & \phi_{N,N} & \omega_{N,1} & \cdots & \omega_{N,N} \end{bmatrix}$$

$\rho_{k,l}$: Forward-Forward correlations (BGM)
$\omega_{k,l}$: Volatility-Volatility correlations
$\phi_{k,l}$: Forward-Volatility correlations
ρ_k: SABR Rho parameters

SABR-BGM Correlation Matrix

A special case of null (zero) Forward-Volatility correlations has some advantages:

- Sets the Forward-Volatility correlations $\varphi_{k,1}$ and the SABR Rho ρ_k to zero, which means fewer parameters to calibrate
- Still fits market data
- Gives lower variance and faster convergence in Monte Carlo simulation

SABR-BGM Model in Rebonato Parametric Form

To facilitate the calibration of the SABR-BGM model using market data, Rebonato et. al. (2009) introduced the parametric form.

In the parametric form, the SABR volatility $\alpha_k(t)$ is decomposed into a product of two components: the deterministic volatility function $g(T_{k-1} - t)$ and the stochastic correction term $s_k(t)$.

$$dF_k(t) = \mu_k(t)dt + \alpha_k {F_k}^{\beta_k}(t)dW_{F_k}(t)$$

$$d\alpha_k(t) = g(T_{k-1} - t)ds_k(t)$$

$$ds_k(t) = \eta_k(t)dt + h(T_{k-1} - t)s_k(t)dW_{S_k}(t)$$

$$\mu_k(t) = g(T_{k-1} - t)s_k(t){F_k}^{\beta_k}(t) \sum_{\gamma(t) \le l \le k} \frac{\rho_{l,k}\tau_l g(T_{l-1} - t)s_l(t){F_l}^{\beta_l}(t)}{1 + \tau_l F_l(t)} dt$$

$$\eta_k(t) = \phi_{k,k} h(T_{k-1} - t)s_k(t) \sum_{\gamma(t) \le l \le k} \frac{\rho_{l,k}\tau_l g(T_{l-1} - t)s_l(t){F_l}^{\beta_l}(t)}{1 + \tau_l F_l(t)} dt$$

$$T_{\gamma(t)-1} \le t < T_{\gamma(t)}$$

SABR-BGM Model in Rebonato Parametric Form

- The volatility function $g(T_{k-1} - t)$ is a deterministic function of time.
- The volatility-of-volatility function $h(T_{k-1} - t)$ is a deterministic function of time.
- The correlation matrix is $2N$-by-$2N$:

$$\mathbf{E}\left[dW_{F_k}(t)dW_{F_l}(t)\right] = \rho_{k,l}dt$$
$$\mathbf{E}\left[dW_{\alpha_k}(t)dW_{\alpha_l}(t)\right] = \omega_{k,l}dt$$
$$\mathbf{E}\left[dW_{F_k}(t)dW_{\alpha_l}(t)\right] = \phi_{k,l}dt$$
$$\phi_{k,k}dt = \rho_k dt$$

$$\Pi = \begin{bmatrix} \rho & \phi \\ \phi^T & \omega \end{bmatrix} = \begin{bmatrix} \rho_{1,1} & \cdots & \rho_{1,N} & \phi_{1,1} & \cdots & \phi_{1,N} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \rho_{N,1} & \cdots & \rho_{N,N} & \phi_{N,1} & \cdots & \phi_{N,N} \\ \phi_{1,1} & \cdots & \phi_{N,1} & \omega_{1,1} & \cdots & \omega_{1,N} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \phi_{1,N} & \cdots & \phi_{N,N} & \omega_{N,1} & \cdots & \omega_{N,N} \end{bmatrix}$$

$\rho_{k,l}$: Forward-Forward correlations (BGM)
$\omega_{k,l}$: Volatility-Volatility correlations
$\phi_{k,l}$: Forward-Volatility correlations
ρ_k: SABR Rho parameters

SABR-BGM Model in Rebonato Parametric Form Correlation Matrix

A special case of null (zero) Forward-Volatility correlations has some advantages:

- Sets the Forward-Volatility correlations $\varphi_{k,1}$ and the SABR Rho ρ_k to zero, which means fewer parameters to calibrate
- Still fits market data
- Gives lower variance and faster convergence in Monte Carlo simulation

Version History

Introduced in R2021b

References

[1] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer Finance, 2006.

[2] Crispoldi, C., Wigger, G., and P. Larkin. *SABR and SABR LIBOR Market Models in Practice*. Palgrave MacMillan, 2015.

[3] Hagan, P. and A. Lesniewski. *LIBOR market Model with SABR Style Volatility.* Working paper JPMorgan Chase, 2008.

[4] Rebonato, R., McKay, K., and R. White. *The SABR/LIBOR Market Model: pricing, Calibration, and Hedging for Complex Interest-Rate Derivatives.* Wiley, 2009.

See Also

Functions
`BraceGatarekMusiela` | `fininstrument` | `finpricer`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

LinearGaussian2F

Create `LinearGaussian2F` model object for `Cap`, `Floor`, `Swaption`, `Swap`, `FixedBond`, `FloatBond`, `FloatBondOption`, `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument

Description

Create and price a `Cap`, `Floor`, `Swaption`, `Swap`, `FloatBond`, `FloatBondOption`, `FixedBond`, `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object with a `LinearGaussian2F` model using this workflow:

1. Use `fininstrument` to create a `Cap`, `Floor`, `Swaption`, `Swap`, `FixedBond`, `FloatBond`, `FloatBondOption` `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.
2. Use `finmodel` to specify a `LinearGaussian2F` model object for the `Cap`, `Floor`, `Swaption`, `Swap`, `FixedBond`, `FloatBond`, `FloatBondOption`, `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.
3. Use `finpricer` to specify an `IRMonteCarlo` pricing method for a `Cap`, `Floor`, `Swaption`, `Swap`, `FixedBond`, `FloatBond`, `FloatBondOption`, `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Cap`, `Floor`, `Swaption`, `Swap`, `FixedBond`, `FloatBond`, `FloatBondOption`, `FixedBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
LinearGaussian2FModelObj = finmodel(ModelType,Alpha1=alpha1_value,
Sigma1=sigma1_value,Alpha2=alpha2_value,Sigma2=sigma2_value,
Correlation=correlation_value)
```

Description

`LinearGaussian2FModelObj = finmodel(ModelType,Alpha1=alpha1_value, Sigma1=sigma1_value,Alpha2=alpha2_value,Sigma2=sigma2_value, Correlation=correlation_value)` creates a `LinearGaussian2F` model object by specifying `ModelType` and the required name-value arguments for `Alpha1`, `Sigma1`, `Alpha2`, `Sigma2` and `Correlation` to set properties on page 11-3091 using name-value pair arguments. For example, `LinearGaussian2FModelObj = finmodel("LinearGaussian2F",Alpha1=0.07,Sigma1=0.01,Alpha2=0.5,Sigma2=0.006,Correlation=-0.7)` creates a `LinearGaussian2F` model object.

Input Arguments

`ModelType` — Model type

string with value `"LinearGaussian2F"` | character vector with value `'LinearGaussian2F'`

Model type, specified as a string with the value of `"LinearGaussian2F"` or a character vector with the value of `'LinearGaussian2F'`.

Data Types: `char` | `string`

`LinearGaussian2F` Name-Value Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `LinearGaussian2FModelObj = finmodel("LinearGaussian2F",Alpha1=0.07,Sigma1=0.01,Alpha2=0.5,Sigma2=0.006,Correlation=-0.7)`

`Alpha1` — Positive mean reversion value for first factor

scalar numeric | timetable

Positive mean reversion value for first factor, specified as `Alpha1` and a scalar numeric or timetable.

Data Types: `double` | `timetable`

`Sigma1` — Positive volatility for first factor

scalar numeric | timetable

Positive volatility for first factor, specified as `Sigma1` and a scalar numeric or timetable.

Data Types: `double` | `timetable`

`Alpha2` — Positive mean reversion value for second factor

numeric | timetable

Positive mean reversion value for the second factor, specified as `Alpha2` and a scalar numeric or timetable.

Data Types: `double` | `timetable`

`Sigma2` — Positive volatility for second factor

numeric | timetable

Positive volatility for second factor, specified as `Sigma2` and a scalar numeric or timetable.

Data Types: `double` | `timetable`

`Correlation` — Scalar correlation of factors

numeric

Scalar correlation of factors, specified as `Correlation` and a scalar numeric.

Data Types: `double`

Properties

`Alpha1` — Positive mean reversion for first factor
numeric | timetable

Positive mean reversion for first factor, returned as a scalar numeric or timetable.

Data Types: `double`

`Sigma1` — Positive volatility for first factor
numeric | timetable

Positive volatility for first factor, returned as a scalar numeric value or timetable.

Data Types: `double`

`Alpha2` — Positive mean reversion value for second factor
numeric | timetable

Positive mean reversion value for second factor, returned as a scalar numeric or timetable.

Data Types: `double`

`Sigma2` — Positive volatility for second factor
numeric | timetable

Positive volatility for second factor, returned as a scalar numeric value or timetable.

Data Types: `double` | `timetable`

`Correlation` — Scalar correlation of factors
numeric

Scalar correlation of factors, returned as a scalar numeric value.

Data Types: `double`

Examples

Use LinearGaussian2F Model and IRMonteCarlo Pricer to Price Cap Instrument

This example shows the workflow to price a `Cap` instrument when using a `LinearGaussian2F` model and an `IRMonteCarlo` pricing method.

Create Cap Instrument Object

Use `fininstrument` to create a `Cap` instrument object.

```
CapOpt = fininstrument("Cap",Maturity=datetime(2022,9,15),Strike=0.01,Reset=2,Name="cap_option")
```

```
CapOpt =
  Cap with properties:

                    Strike: 0.0100
                  Maturity: 15-Sep-2022
               ResetOffset: 0
```

```
                    Reset: 2
                    Basis: 0
                Principal: 100
          ProjectionCurve: [0x0 ratecurve]
  DaycountAdjustedCashFlow: 0
     BusinessDayConvention: "actual"
                 Holidays: NaT
                     Name: "cap_option"
```

Create LinearGaussian2F Model Object

Use `finmodel` to create a `LinearGaussian2F` model object.

```
LinearGaussian2FModel = finmodel("LinearGaussian2F",Alpha1=0.07,Sigma1=0.01,Alpha2=0.5,Sigma2=0.0
```

```
LinearGaussian2FModel =
  LinearGaussian2F with properties:

        Alpha1: 0.0700
        Sigma1: 0.0100
        Alpha2: 0.5000
        Sigma2: 0.0060
   Correlation: -0.7000
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

               Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: [10x1 datetime]
             Rates: [10x1 double]
            Settle: 01-Jan-2019
      InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",Model=LinearGaussian2FModel,DiscountCurve=myRC,SimulationDat
```

```
outPricer =
  G2PPMonteCarlo with properties:

         NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jan-2021    ...    ]
              Model: [1x1 finmodel.LinearGaussian2F]
```

Price Cap Instrument

Use `price` to compute the price and sensitivities for the `Cap` instrument.

```
[Price,outPR] = price(outPricer,CapOpt,["all"])
```

```
Price = 1.2156

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta     Gamma          Vega
    ______    ______    _____    _______________

    1.2156    131.37    11048    126.5    -157.38
```

More About

LinearGaussian2F Model

The LinearGaussian2F two-factor additive Gaussian interest-rate model.

Specifically, the LinearGaussian2F model is defined using the following equations:

$$r(t) = x(t) + y(t) + \phi(t)$$

$$dx(t) = -a(t)x(t)dt + \sigma(t)dW_1(t), x(0) = 0$$

$$dy(t) = -b(t)y(t)dt + \eta(t)dW_2(t), y(0) = 0$$

where $dW_1(t)dW_2(t) = \rho dt$ is a two-dimensional Brownian motion with correlation ρ, and ϕ is a function chosen to match the initial zero curve.

Version History

Introduced in R2021b

See Also

Functions
`fininstrument` | `finpricer`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

BlackKarasinski

Create `BlackKarasinski` model object for a `Cap`, `FloorSwaption`, `Swap`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument

Description

Create and price a `Cap`, `Floor`, `Swaption`, `Swap`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object with a `BlackKarasinski` model using this workflow:

1. Use `fininstrument` to create a `Cap`, `Floor`, `Swaption`, `Swap`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.
2. Use `finmodel` to specify a `BlackKarasinski` model object for the `Cap`, `Floor`, `Swaption`, `Swap`, `FixedBond`, `FloatBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.
3. Use `finpricer` to specify a `IRTree` or `IRMonteCarlo` pricing method for the `Cap`, `Floor`, `Swaption`, `SwapFixedBond`, `FloatBond`, `FixedBondOption` or `OptionEmbeddedFixedBond` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Cap`, `FloorSwaption`, `Swap`, `FixedBond`, `FloatBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BlackKarasinskiModelObj = finmodel(ModelType,'Alpha',alpha_value,'
Sigma',sigma_value)
```

Description

`BlackKarasinskiModelObj = finmodel(ModelType,'Alpha',alpha_value,'Sigma',sigma_value)` creates a `BlackKarasinski` model object by specifying `ModelType` and the required name-value pair arguments `Alpha` and `Sigma` to set the properties on page 11-3096. For example, `BlackKarasinskiModelObj = finmodel("BlackKarasinski",'Alpha',0.052,'Sigma',0.34)` creates a `BlackKarasinski` model object.

Input Arguments

ModelType — Model type

string with value "BlackKarasinski" | character vector with value 'BlackKarasinski'

Model type, specified as a string with the value of "BlackKarasinski" or a character vector with the value of 'BlackKarasinski'.

Data Types: char | string

BlackKarasinski Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: BlackKarasinskiModelObj = finmodel("BlackKarasinski",'Alpha',0.052,'Sigma',0.34)

Alpha — Mean reversion speed

numeric | timetable

Mean reversion speed, specified as the comma-separated pair consisting of 'Alpha' and a scalar numeric value or timetable.

Alpha accepts a timetable, where the first column is dates and the second column is the associated Alpha value.

Data Types: double | timetable

Sigma — Volatility

numeric | timetable

Volatility, specified as the comma-separated pair consisting of 'Sigma' and a scalar numeric value or timetable.

Sigma accepts a timetable, where the first column is dates and the second column is the associated Sigma value.

Data Types: double | timetable

Properties

Alpha — Mean reversion speed

numeric

Mean reversion speed, returned as a scalar numeric value or timetable.

Data Types: double | timetable

Sigma — Volatility

numeric

Volatility, returned as a scalar numeric value or timetable.

Data Types: double | timetable

Examples

Use Black-Karasinski Model and Black-Karasinski Tree Pricer to Price FixedBondOption Instrument

This example shows the workflow to price a `FixedBondOption` instrument when you use a `BlackKarasinski` model and an `IRTree` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",'Maturity',datetime(2029,9,15),'CouponRate',.024,'Principal
```

```
BondInst =
  FixedBond with properties:

                 CouponRate: 0.0240
                     Period: 1
                      Basis: 1
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2029
                        Name: "fixed_bond"
```

Create FixedBondOption Instrument Object

Use `fininstrument` to create a `FixedBondOption` instrument object.

```
FixedBOption = fininstrument("FixedBondOption",'ExerciseDate',datetime(2025,9,15),'Strike',800,'
```

```
FixedBOption =
  FixedBondOption with properties:

       OptionType: "put"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2025
           Strike: 800
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option"
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calyears([1:10])]';
```

```
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates, 'Basis',5)

myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 5
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackKarasinski Model Object

Use `finmodel` to create a `BlackKarasinski` model object.

```
BlackKarasinskiModel = finmodel("BlackKarasinski",'Alpha',0.02,'Sigma',0.34)

BlackKarasinskiModel =
  BlackKarasinski with properties:

    Alpha: 0.0200
    Sigma: 0.3400
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
BKTreePricer = finpricer("IRTree",'Model',BlackKarasinskiModel,'DiscountCurve',myRC,'TreeDates',Z

BKTreePricer =
  HWBKTree with properties:

             Tree: [1x1 struct]
        TreeDates: [10x1 datetime]
            Model: [1x1 finmodel.BlackKarasinski]
    DiscountCurve: [1x1 ratecurve]


BKTreePricer.Tree

ans = struct with fields:
        tObs: [0 1 2 3 4 5 6 7 8 9]
        dObs: [15-Sep-2019    15-Sep-2020    15-Sep-2021    ...    ]
      CFlowT: {1x10 cell}
       Probs: {1x9 cell}
     Connect: {1x9 cell}
     FwdTree: {1x10 cell}
    RateTree: {1x10 cell}
```

Price FixedBondOption Instrument

Use `price` to compute the price and sensitivities for the `FixedBondOption` instrument.

```
[Price, outPR] = price(BKTreePricer,FixedBOption,["all"])
```

```
Price = 705.2729

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta      Gamma        Vega
    ______    ______    _______    ___________

    705.27    844.75    -8084.8    -1.1369e-09
```

Use Black-Karasinski Model and IRMonteCarlo Pricer to Price Fixed Bond Instrument

This example shows the workflow to price a `FixedBond` instrument when using a `BlackKarasinski` model and an `IRMonteCarlo` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",Maturity=datetime(2022,9,15),CouponRate=0.05,Name="fixed_bond")
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond"
```

Create BlackKarasinski Model Object

Use `finmodel` to create a `BlackKarasinski` model object.

```
BlackKarasinskiModel = finmodel("BlackKarasinski",'Alpha',0.02,'Sigma',0.34)
```

```
BlackKarasinskiModel =
  BlackKarasinski with properties:

    Alpha: 0.0200
    Sigma: 0.3400
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",Model=BlackKarasinskiModel,DiscountCurve=myRC,SimulationDate
```

```
outPricer =
  BKMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [15-Mar-2019    15-Sep-2019    15-Mar-2020    ...    ]
              Model: [1x1 finmodel.BlackKarasinski]
```

Price FixedBond Instrument

Use `price` to compute the price for the `FixedBond` instrument.

```
[Price,outPR] = price(outPricer,FixB,["all"])
```

```
Price = 113.3046
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta     Gamma      Vega
    _____    _______    ______    ______

    113.3    -310.39    -26741    2.0294
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finpricer` | `timetable`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

BlackDermanToy

Create BlackDermanToy model object for a Cap, Floor, Swaption, Swap, FloatBond, FixedBond, FixedBondOption, FloatBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument

Description

Create and price a Cap, Floor, Swaption, Swap, FloatBond, FixedBond, FixedBondOption, FloatBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object with a BlackDermanToy model using this workflow:

1. Use fininstrument to create a Cap, Floor, Swaption, Swap, FloatBond, FixedBond, FixedBondOption, FloatBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object.
2. Use finmodel to specify a BlackDermanToy model object for the Cap, Floor, Swaption, Swap, FixedBond, FloatBond, FixedBondOption, FloatBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object.
3. Use finpricer to specify an IRTree pricing method for the Cap, Floor, Swaption, Swap, FixedBond, FloatBond, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a Cap, Floor, Swaption, Swap, FixedBond, FloatBond, FixedBondOption, FloatBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BlackDermanToyModelObj = finmodel(ModelType,Sigma=sigma_value)
```

Description

BlackDermanToyModelObj = finmodel(ModelType,Sigma=sigma_value) creates a BlackDermanToy model object by specifying ModelType and the required name-value pair argument Sigma to set the properties on page 11-3103. For example, BlackDermanToyModelObj = finmodel("BlackDermanToy",Sigma=0.34) creates a BlackDermanToy model object with a Sigma volatility of .34.

Input Arguments

ModelType — Model type
string with value "BlackDermanToy" | character vector with value 'BlackDermanToy'

Model type, specified as a string with the value of `"BlackDermanToy"` or a character vector with the value of `'BlackDermanToy'`.

Data Types: `char` | `string`

`BlackDermanToy` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BlackDermanToyModelObj = finmodel("BlackDermanToy",Sigma=0.34)`

`Sigma` — Positive volatility
numeric | timetable

Positive volatility, specified as `Sigma` and a scalar numeric value or timetable.

`Sigma` accepts a `timetable`, where the first column is dates and the second column is the associated `Sigma` value.

Data Types: `double` | `timetable`

Properties

`Sigma` — Positive volatility
numeric

Positive volatility, returned as a scalar numeric value or timetable.

Data Types: `double` | `timetable`

Examples

Use Black-Derman-Toy Model and BlackDermanToy Tree Pricer to Price FixedBondOption Instrument

This example shows the workflow to price a `FixedBondOption` instrument when you use a `BlackDermanToy` model and an `IRTree` pricing method.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",Maturity=datetime(2029,9,15),CouponRate=.024,Principal=100,I
```

```
BondInst =
  FixedBond with properties:

                 CouponRate: 0.0240
                     Period: 1
                      Basis: 1
               EndMonthRule: 1
                  Principal: 100
```

```
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2029
                        Name: "fixed_bond"
```

Create `FixedBondOption` Instrument Object

Use `fininstrument` to create a `FixedBondOption` instrument object.

```
FixedBOption = fininstrument("FixedBondOption",ExerciseDate=datetime(2025,9,15),Strike=80,Bond=Bo
```

```
FixedBOption =
  FixedBondOption with properties:

       OptionType: "put"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2025
           Strike: 80
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calyears(1:10)]';
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates,Basis=5)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 5
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `BlackDermanToy` Model Object

Use `finmodel` to create a `BlackDermanToy` model object.

```
BlackDermanToyModel = finmodel("BlackDermanToy",Sigma=0.14)
```

```
BlackDermanToyModel =
  BlackDermanToy with properties:

    Sigma: 0.1400
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
BKTreePricer = finpricer("IRTree",Model=BlackDermanToyModel,DiscountCurve=myRC,TreeDates=ZeroDate
```

```
BKTreePricer =
  BDTTree with properties:

             Tree: [1x1 struct]
        TreeDates: [10x1 datetime]
            Model: [1x1 finmodel.BlackDermanToy]
    DiscountCurve: [1x1 ratecurve]
```

```
BKTreePricer.Tree
```

```
ans = struct with fields:
        tObs: [0 1 2 3 4 5 6 7 8 9]
        dObs: [15-Sep-2019    15-Sep-2020    15-Sep-2021    ...    ]
     FwdTree: {1x10 cell}
    RateTree: {1x10 cell}
```

Price FixedBondOption Instrument

Use `price` to compute the price and sensitivities for the `FixedBondOption` instrument.

```
[Price, outPR] = price(BKTreePricer,FixedBOption,"all")
```

```
Price = 0.5153
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
     Price     Delta      Gamma      Vega
    _______    ______    _______    ______

    0.51526    106.21    -1973.8    15.402
```

Version History

Introduced in R2022b

See Also

Functions
`fininstrument` | `finpricer` | `timetable`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Heston

Create `Heston` model object for `Vanilla`, `Asian`, `Barrier`, `DoubleBarrier`, `Lookback`, `PartialLookback`, `VarianceSwap`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument

Description

Create and price a `Vanilla`, `Asian`, `Barrier`, `DoubleBarrier`, `Lookback`, `PartialLookback`, `VarianceSwap`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument object with a `Heston` model using this workflow:

1. Use `fininstrument` to create a `Vanilla`, `Barrier`, `Lookback`, `PartialLookback`, `Asian`, `DoubleBarrier`, `VarianceSwap`, `Binary`, `Touch`, `Cliquet`, or `DoubleTouch` instrument object.
2. Use `finmodel` to specify a `Heston` model object for the `Vanilla`, `Asian`, `Barrier`, `DoubleBarrier`, `Lookback`, `PartialLookback`, `VarianceSwap`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument object.
3. Use `finpricer` to specify a `FiniteDifference`, `NumericalIntegration`, or `FFT` pricing method for the `Vanilla` instrument object.

   Use `finpricer` to specify an `AssetMonteCarlo` pricing method for the `Vanilla`, `Asian`, `BarrierDoubleBarrier`, `Lookback`, `PartialLookback`, `VarianceSwap`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Vanilla`, `AsianBarrier`, `DoubleBarrier`, `Lookback`, `PartialLookback`, `VarianceSwap`, `Touch`, `DoubleTouch`, or `Binary` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
HestonModelObj = finmodel(ModelType,'V0'v0_value,'ThetaV',thetav_value,'
Kappa',kappa_value,'SigmaV',sigmav_value,'RhoSV',rhosv_value)
```

Description

`HestonModelObj = finmodel(ModelType,'V0'v0_value,'ThetaV',thetav_value,'Kappa',kappa_value,'SigmaV',sigmav_value,'RhoSV',rhosv_value)` creates a `Black` model object by specifying `ModelType` and the required name-value pair arguments `V0`, `ThetaV`, `Kappa`, `SigmaV`, and `RhoSV` to set properties on page 11-3109 using required name-value pair arguments. For example, `HestonModelObj = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9)` creates a `Heston` model object.

Input Arguments

`ModelType` — Model type

string with value `"Heston"` | character vector with value `'Heston'`

Model type, specified as a string with the value of `"Heston"` or a character vector with the value of `'Heston'`.

Data Types: `char` | `string`

`Heston` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `HestonModelObj = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV', 0.9)`

`V0` — Initial variance of underlying asset

numeric

Initial variance of the underlying asset, specified as the comma-separated pair consisting of `'V0'` and a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset

numeric

Long-term variance of the underlying asset, specified as the comma-separated pair consisting of `'ThetaV'` and a scalar numeric value.

Data Types: `double`

`Kappa` — Mean revision speed for the variance of underlying asset

numeric

Mean revision speed for the underlying asset, specified as the comma-separated pair consisting of `'Kappa'` and a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset

numeric

Volatility of the variance of the underlying asset, specified as the comma-separated pair consisting of `'SigmaV'` and a scalar numeric value.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance

numeric

Correlation between the Weiner processes for the underlying asset and its variance, specified as the comma-separated pair consisting of `'RhoSV'` and a scalar numeric value.

Data Types: `double`

Properties

`V0` — Initial variance of underlying asset

numeric

Initial variance of the underlying asset, returned as a scalar numeric value.

Data Types: `double`

`ThetaV` — Long-term variance of underlying asset

numeric

Long-term variance of the underlying asset, returned as a scalar numeric value.

Data Types: `double`

`Kappa` — Mean revision speed for the variance of underlying asset

numeric

Mean revision speed for the underlying asset, returned as a scalar numeric value.

Data Types: `double`

`SigmaV` — Volatility of the variance of underlying asset

numeric

Volatility of the variance of the underlying asset, returned as a scalar numeric value.

Data Types: `double`

`RhoSV` — Correlation between Weiner processes for underlying asset and its variance

numeric

Correlation between the Weiner processes for the underlying asset and its variance, returned as a scalar numeric value.

Data Types: `double`

Examples

Use Heston Model and FFT Pricer to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `Heston` model and an `FFT` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2022,9,15),'Strike',105,'ExerciseSty
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
           Strike: 105
             Name: "vanilla_option"
```

Create Heston Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9)
```

```
HestonModel =
  Heston with properties:

        V0: 0.0320
    ThetaV: 0.1000
     Kappa: 0.0030
    SigmaV: 0.2000
     RhoSV: 0.9000
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create FFT Pricer Object

Use `finpricer` to create an FFT pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("FFT",'DiscountCurve',myRC,'Model',HestonModel,'SpotPrice',100,'Characteris
```

```
outPricer =
  FFT with properties:
```

```
                  Model: [1x1 finmodel.Heston]
          DiscountCurve: [1x1 ratecurve]
              SpotPrice: 100
           DividendType: "continuous"
          DividendValue: 0
                 NumFFT: 8192
    CharacteristicFcnStep: 0.2000
          LogStrikeStep: 0.0038
      CharacteristicFcn: @characteristicFcnHeston
          DampingFactor: 1.5000
             Quadrature: "simpson"
        VolRiskPremium: 0
             LittleTrap: 1
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 14.7545

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma       Theta       Rho       Vega     VegaLT
    _____    _______    ________    ________    ______    ______    ______

    14.754    0.44868    0.021649    -0.20891    120.45    88.192    1.3248
```

Use Heston Model and Asset Monte-Carlo Pricer to Price Lookback Instrument

This example shows the workflow to price a `LookBack` instrument when you use a `Heston` model and an `AssetMonetCarlo` pricing method.

Create Lookback Instrument Object

Use `fininstrument` to create a `Lookback` instrument object.

```
LookbackOpt = fininstrument("Lookback",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionTy
```

```
LookbackOpt =
  Lookback with properties:

       OptionType: "put"
           Strike: 105
      AssetMinMax: NaN
    ExerciseStyle: "american"
```

```
     ExerciseDate: 15-Sep-2022
             Name: "lookback_option"
```

Create Heston Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.08,'RhoSV',0.9)
```

```
HestonModel =
  Heston with properties:

        V0: 0.0320
    ThetaV: 0.1000
     Kappa: 0.0030
    SigmaV: 0.0800
     RhoSV: 0.9000
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",HestonModel,'SpotPrice',90,
```

```
outPricer =
  HestonMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 90
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Heston]
```

```
       DividendType: "continuous"
      DividendValue: 0
```

Price Lookback Instrument

Use `price` to compute the price and sensitivities for the `Lookback` instrument.

```
[Price, outPR] = price(outPricer,LookbackOpt,["all"])
```

```
Price = 21.9733
```

```
outPR =
  priceresult with properties:

       Results: [1x8 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×8 table
    Price      Delta      Gamma     Lambda       Rho        Theta      Vega      VegaLT
    ______    ________    _____    ________    ________    ________    ______    ______

    21.973    -0.7701       0       -3.1542    -215.94     0.28812    99.825    1.447
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

HullWhite

Create HullWhite model object for Cap, Floor, Swaption, Swap, FixedBond, FloatBond, FloatBondOption, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument

Description

Create and price a Cap, Floor, Swaption, Swap, FloatBond, FloatBondOption, FixedBond, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object with a HullWhite model using this workflow:

1. Use fininstrument to create a Cap, Floor, Swaption, Swap, FixedBond, FloatBond, FloatBondOption FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object.
2. Use finmodel to specify a HullWhite model object for the Cap, Floor, Swaption, Swap, FixedBond, FloatBond, FloatBondOption, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object.
3. Use finpricer to specify a HullWhite pricing method for a Cap, Floor, or Swaption instrument object and use an IRTree or IRMonteCarlo pricing method for the Cap, Floor, Swaption, Swap, FixedBond, FloatBond, FloatBondOption, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a Cap, Floor, Swaption, Swap, FixedBond, FloatBond, FloatBondOption, FixedBondOption, OptionEmbeddedFixedBond, or OptionEmbeddedFloatBond instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
HullWhiteModelObj = finmodel(ModelType,'Alpha'alpha_value,'
Sigma',sigma_value)
```

Description

`HullWhiteModelObj = finmodel(ModelType,'Alpha'alpha_value,'Sigma',sigma_value)` creates a HullWhite model object by specifying ModelType and the required name-value pair argumentsAlpha and Sigma to set properties on page 11-3115 using name-value pair arguments. For example, `HullWhiteModelObj = finmodel("HullWhite",'Alpha',0.052,'Sigma',0.34)` creates a HullWhite model object.

Input Arguments

`ModelType` — Model type
string with value `"HullWhite"` | character vector with value `'HullWhite'`

Model type, specified as a string with the value of `"HullWhite"` or a character vector with the value of `'HullWhite'`.

Data Types: `char` | `string`

`HullWhite` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `HullWhiteModelObj = finmodel("HullWhite",'Alpha',0.052,'Sigma',0.34)`

`Alpha` — Mean reversion speed
numeric

Mean reversion speed, specified as the comma-separated pair consisting of `'Alpha'` and a scalar numeric or timetable.

`Alpha` accepts a `timetable`, where the first column is dates and the second column is the associated `Alpha` value.

Data Types: `double` | `timetable`

`Sigma` — Volatility
numeric | timetable

Volatility, specified as the comma-separated pair consisting of `'Sigma'` and a scalar numeric or timetable.

`Sigma` accepts a `timetable`, where the first column is dates and the second column is the associated `Sigma` value.

Data Types: `double` | `timetable`

Properties

`Alpha` — Mean reversion speed
numeric

Mean reversion speed, returned as a scalar numeric or timetable.

Data Types: `double` | `timetable`

`Sigma` — Volatility
numeric | timetable

Volatility, returned as a scalar numeric value or timetable.

Data Types: `double` | `timetable`

Examples

Use Hull-White Model and Hull-White Pricer to Price Floor Instrument

This example shows the workflow to price a `Floor` instrument when you use a `HullWhite` model and a `HullWhite` pricing method.

Create `Floor` Instrument Object

Use `fininstrument` to create a `Floor` instrument object.

```
FloorOpt = fininstrument("Floor",'Strike',0.045,'Maturity',datetime(2019,1,30),'Reset',4,'Princi
```

```
FloorOpt =
  Floor with properties:

                     Strike: 0.0450
                   Maturity: 30-Jan-2019
                ResetOffset: 0
                      Reset: 4
                      Basis: 1
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "floor_option"
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.032,'Sigma',0.04)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0320
    Sigma: 0.0400
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:
```

```
                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `HullWhite` Pricer Object

Use `finpricer` to create a `HullWhite` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',HullWhiteModel,'DiscountCurve',myRC)
```

```
outPricer =
  HullWhite with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.HullWhite]
```

Price `Floor` Instrument

Use `price` to compute the price for the `Floor` instrument.

```
Price = price(outPricer,FloorOpt)
```

```
Price = 1.4917
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Merton

Create Merton model object for Vanilla, Asian, Barrier, DoubleBarrier, Lookback, PartialLookback, OneTouch, DoubleTouch, Cliquet, or Binary instrument

Description

Create and price a Vanilla, Asian, Barrier, DoubleBarrier, Lookback, PartialLookback, Touch, DoubleTouch, Cliquet, or Binary instrument object with a Merton model using this workflow:

1. Use fininstrument to create a Vanilla, Barrier, Lookback, PartialLookback, Asian, DoubleBarrier, Binary, Touch, Cliquet, or DoubleTouch instrument object.
2. Use finmodel to specify a Merton model object for the Vanilla, Asian, Barrier, DoubleBarrier, Lookback, PartialLookback, Touch, DoubleTouch, Cliquet, or Binary instrument object.
3. Use finpricer to specify a FiniteDifference, NumericalIntegration, or FFT pricing method for the Vanilla instrument object.

   Use finpricer to specify an AssetMonteCarlo pricing method for the Vanilla, Asian, BarrierDoubleBarrier, Lookback, PartialLookback, Touch, DoubleTouch, Cliquet, or Binary instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a Vanilla, Asian, Barrier, DoubleBarrier, Lookback, PartialLookback, Touch, DoubleTouch, or Binary instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
MertonModelObj = finmodel(ModelType,'Volatility',volatility_value,'
MeanJ',meanj_value,'JumpVol',jumpvol_value,'JumpFreq',jumpfreq_value)
```

Description

`MertonModelObj = finmodel(ModelType,'Volatility',volatility_value,'MeanJ',meanj_value,'JumpVol',jumpvol_value,'JumpFreq',jumpfreq_value)` creates a Merton model object by specifying ModelType and the required name-value pair arguments MeanJ, JumpVol, and JumpFreq to set properties on page 11-3120 using name-value pair arguments. For example, `MertonModelObj = finmodel("Merton",'Volatility',0.03,'MeanJ',0.22,'JumpVol',0.007,'JumpFreq',0.009)` creates a Merton model object.

Input Arguments

`ModelType` — Model type

string with value `"Merton"` | character vector with value `'Merton'`

Model type, specified as a string with the value of `"Merton"` or a character vector with the value of `'Merton'`.

Data Types: `char` | `string`

`Merton` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `MertonModelObj = finmodel("Merton",'Volatility',0.03,'MeanJ',0.22,'JumpVol',0.007,'JumpFreq',0.009)`

`Volatility` — Volatility value for the underlying asset

nonnegative numeric

Volatility value for the underlying asset, specified as the comma-separated pair consisting of `'Volatility'` and a scalar nonnegative numeric.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size

decimal

Mean of the random percentage jump size (J), specified as the comma-separated pair consisting of `'MeanJ'` and a scalar decimal value, where `log`(1+J) is normally distributed with mean (`log`(1+`MeanJ`)-0.5*`JumpVol`^2) and the standard deviation `JumpVol`.

Data Types: `double`

`JumpVol` — Standard deviation of `log`(1+J)

decimal

Standard deviation of `log`(1+J), where `J` is the random percentage jump size, specified as the comma-separated pair consisting of `'JumpVol'` and a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process

numeric

Annual frequency of the Poisson jump process, specified as the comma-separated pair consisting of `'JumpFreq'` and a scalar numeric value.

Data Types: `double`

Properties

`Volatility` — Volatility value
nonnegative numeric

Volatility value, returned as a scalar nonnegative numeric.

Data Types: `double`

`MeanJ` — Mean of the random percentage jump size
decimal

Mean of the random percentage jump size (*J*), returned as a scalar decimal value.

Data Types: `double`

`JumpVol` — Standard deviation of `log(1+J)`
decimal

Standard deviation of `log`(1+*J*), where `J` is the random percentage jump size, returned as a scalar decimal value.

Data Types: `double`

`JumpFreq` — Annual frequency of Poisson jump process
numeric

Annual frequency of the Poisson jump process, returned as a scalar numeric value.

Data Types: `double`

Examples

Use Merton Model and Numerical Integration Pricer to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `Merton` model and a `NumericalIntegration` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2022,9,15),'Strike',105,'OptionType
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
           Strike: 105
             Name: "vanilla_option"
```

Create `Merton` Model Object

Use `finmodel` to create a `Merton` model object.

```
MertonModel = finmodel("Merton",'Volatility',0.45,'MeanJ',0.02,'JumpVol',0.07,'JumpFreq',0.09)

MertonModel =
  Merton with properties:

    Volatility: 0.4500
         MeanJ: 0.0200
       JumpVol: 0.0700
      JumpFreq: 0.0900
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)

myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `NumericalIntegration` Pricer Object

Use `finpricer` to create a `NumericalIntegration` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("numericalintegration",'DiscountCurve',myRC,'Model',MertonModel,'SpotPrice

outPricer =
  NumericalIntegration with properties:

                Model: [1x1 finmodel.Merton]
        DiscountCurve: [1x1 ratecurve]
            SpotPrice: 100
         DividendType: "continuous"
        DividendValue: 0.4500
               AbsTol: 0.5000
               RelTol: 0.0400
     IntegrationRange: [1.0000e-09 Inf]
    CharacteristicFcn: @characteristicFcnMerton76
            Framework: "lewis2001"
       VolRiskPremium: 0.0900
           LittleTrap: 0
```

Price Vanilla Instrument

Use price to compute the price and sensitivities for the Vanilla instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 75.1139

outPR =
  priceresult with properties:

       Results: [1x6 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×6 table
    Price      Delta        Gamma         Theta       Rho       Vega
    _____    ________    __________    _______    _______    _____

    75.114    -0.15305    0.00025732    -3.9836    -361.67    4.6317
```

Use Merton Model and Asset Monte-Carlo Pricer to Price Binary Instrument

This example shows the workflow to price a Binary instrument when you use a Merton model and an AssetMonteCarlo pricing method.

Create Binary Instrument Object

Use fininstrument to create an Binary instrument object.

```
BinaryOpt = fininstrument("Binary",'ExerciseDate',datetime(2022,9,15),'Strike',100,'PayoffValue'
```

```
BinaryOpt =
  Binary with properties:

      OptionType: "put"
    ExerciseDate: 15-Sep-2022
          Strike: 100
     PayoffValue: 130
   ExerciseStyle: "european"
            Name: "binary_option"
```

Create Merton Model Object

Use finmodel to create a Merton model object.

```
MertonModel = finmodel("Merton",'Volatility',0.25,'MeanJ',0.02,'JumpVol',0.07,'JumpFreq',0.09)
```

```
MertonModel =
  Merton with properties:

    Volatility: 0.2500
         MeanJ: 0.0200
```

```
       JumpVol: 0.0700
      JumpFreq: 0.0900
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonetCarlo Pricer Object

Use `finpricer` to create an `AssetMonetCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",MertonModel,'SpotPrice',102
```

```
outPricer =
  MertonMonteCarlo with properties:

     DiscountCurve: [1x1 ratecurve]
         SpotPrice: 102
    SimulationDates: 15-Sep-2022
         NumTrials: 1000
     RandomNumbers: []
             Model: [1x1 finmodel.Merton]
      DividendType: "continuous"
     DividendValue: 0
```

Price Binary Instrument

Use `price` to compute the price and sensitivities for the `Binary` instrument.

```
[Price, outPR] = price(outPricer,BinaryOpt,["all"])
```

```
Price = 53.1178
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
```

```
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta     Gamma     Lambda       Rho      Theta     Vega
    ______    _______    _____    ________    _______    ______    ____

    53.118    -0.4432      0      -0.85106    -212.43    1.8604     0
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Normal

Create `Normal` model object for `Cap`, `Floor`, or `Swaption` instrument

Description

Create and price a `Cap`, `Floor`, or `Swaption` instrument object with a `Normal` model using this workflow:

1. Use `fininstrument` to create a `Cap`, `Floor`, or `Swaption` instrument object.
2. Use `finmodel` to specify a `Normal` model object for the `Cap`, `Floor`, or `Swaption` instrument object.
3. Use `finpricer` to specify a `Normal` pricing method for the `Cap`, `Floor`, or `Swaption` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Cap`, `Floor`, or `Swaption` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
NormalModelObj = finmodel(ModelType,'Volatility',volatility_value)
```

Description

`NormalModelObj = finmodel(ModelType,'Volatility',volatility_value)` creates a `Normal` model object by specifying `ModelType` and the required name-value pair argument `Volatility` to set properties on page 11-3126 using name-value pair arguments. For example, `NormalModelObj = finmodel("Normal",'Volatility',0.063)` creates a `Normal` model object.

Input Arguments

`ModelType` — Model type

string with value `"Normal"` | character vector with value `'Normal'`

Model type, specified as a string with the value of `"Normal"` or a character vector with the value of `'Normal'`.

Data Types: `char` | `string`

`Normal` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `NormalModelObj = finmodel("Normal",'Volatility',0.063)`

`Volatility` — Volatility value
nonnegative numeric

Volatility value, specified as the comma-separated pair consisting of `'Volatility'` and a scalar nonnegative numeric.

Data Types: `double`

Properties

`Volatility` — Volatility value
nonnegative numeric

Volatility value, returned as a scalar nonnegative numeric.

Data Types: `double`

Examples

Use Normal Model and Normal Pricer to Price Cap Instrument

This example shows the workflow to price a `Cap` instrument when you use a `Normal` model and a `Normal` pricing method.

Create Cap Instrument Object

Use `fininstrument` to create a `Cap` instrument object.

```
CapOpt = fininstrument("Cap",'Strike',0.51,'Maturity',datetime(2019,6,25),'Reset',4,'Principal',1
```

```
CapOpt =
  Cap with properties:

                     Strike: 0.5100
                   Maturity: 25-Jun-2019
                ResetOffset: 0
                      Reset: 4
                      Basis: 8
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "cap_option"
```

Create `Normal` Model Object

Use `finmodel` to create a `Normal` model object.

```
NormalModel = finmodel("normal",'Volatility',0.063)
```

```
NormalModel =
  Normal with properties:

    Volatility: 0.0630
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Normal Pricer Object

Use finpricer to create a Normal pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("analytic",'Model',NormalModel,'DiscountCurve',myRC)
```

```
outPricer =
  Normal with properties:

    DiscountCurve: [1x1 ratecurve]
            Shift: 0
            Model: [1x1 finmodel.Normal]
```

Price Cap Instrument

Use price to compute the price for the Cap instrument.

```
Price = price(outPricer,CapOpt)
```

```
Price = 9.3325e-30
```

Version History

Introduced in R2020a

See Also

Functions
`fininstrument` | `finpricer`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Work with Negative Interest Rates Using Objects” on page 2-22

Bachelier

Create `Bachelier` model object for `Vanilla`, `Spread`, or `Binary` instrument

Description

Create and price a `Vanilla`, `Spread`, or `Binary` instrument object with a `Bachelier` model using this workflow:

1. Use `fininstrument` to create a `Vanilla`, `Spread`, or `Binary` instrument object.
2. Use `finmodel` to specify a `Bachelier` model object for the `Vanilla`, `Spread`, or `Binary` instrument object.
3. Use `finpricer` to specify an `AssetMonteCarlo` pricing method for the `Vanilla`, `Spread`, or `Binary` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Vanilla`, `Spread`, or `Binary` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BachelierModelObj = finmodel(ModelType,'Volatility',volatility_value)
BachelierModelObj = finmodel(ModelType,Name,Value)
```

Description

`BachelierModelObj = finmodel(ModelType,'Volatility',volatility_value)` creates a `Bachelier` model object by specifying `ModelType` and the required name-value pair argument `Volatility` to set properties on page 11-3130 using name-value pair arguments. For example, `BachelierModelObj = finmodel("Bachelier",'Volatility',0.063)` creates a `Bachelier` model object.

`BachelierModelObj = finmodel(ModelType,Name,Value)` sets optional properties on page 11-3048 using an additional name-value pair argument in addition to the required arguments in the previous syntax. For example, `BachelierModelObj = finmodel("Bachelier",'Volatility',0.063,'Correlation',Corr)` creates a `Bachelier` model object with a specified `Correlation` value.

Input Arguments

`ModelType` — Model type

string with value `"Bachelier"` | character vector with value `'Bachelier'`

Model type, specified as a string with the value `"Bachelier"` or a character vector with the value `'Bachelier'`.

Data Types: `char` | `string`

Bachelier Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BachelierModelObj = finmodel("Bachelier",'Volatility',0.063,'Correlation',Corr)`

`Volatility` — Volatility value
nonnegative numeric

Volatility value, specified as the comma-separated pair consisting of `'Volatility'` and a scalar nonnegative numeric.

Data Types: `double`

`Correlation` — Correlation for underlying assets
1 (default) | positive semidefinite matrix

Correlation for the underlying assets, specified as the comma-separated pair consisting of `'Correlation'` and a positive semidefinite matrix. For more information on creating a positive semidefinite matrix, see `nearcorr`.

Data Types: `double`

Properties

`Volatility` — Volatility value
nonnegative numeric

Volatility value, returned as a scalar nonnegative numeric.

Data Types: `double`

`Correlation` — Correlation for underlying assets
1 (default) | positive semidefinite matrix

Correlation for the underlying assets, returned as a matrix.

Data Types: `double`

Examples

Use Bachelier Model and Asset Monte-Carlo Pricer to Price American Option for Vanilla Instrument

This example shows the workflow to price an American option for a `Vanilla` instrument when you use a `Bachelier` model and an `AssetMonteCarlo` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionType
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2022
           Strike: 105
             Name: "vanilla_option"
```

Create `Bachelier` Model Object

Use `finmodel` to create a `Bachelier` model object.

```
BachelierModel = finmodel("Bachelier","Volatility",0.2)
```

```
BachelierModel =
  Bachelier with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 12
             Dates: 15-Sep-2023
             Rates: 0.0350
            Settle: 15-Sep-2018
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BachelierModel,'SpotPrice',
```

```
outPricer =
  BachelierMonteCarlo with properties:

     DiscountCurve: [1x1 ratecurve]
         SpotPrice: 150
    SimulationDates: 15-Sep-2022
         NumTrials: 1000
     RandomNumbers: []
             Model: [1x1 finmodel.Bachelier]
      DividendType: "continuous"
     DividendValue: 0
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,"all")
```

```
Price = 57.3776
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma        Lambda     Rho       Theta        Vega
    _______    _______    ___________    _______    _______    _______    ___________

    57.378     0.99107    -1.579e-14     2.5909     291.94     -2.5576    -2.1316e-10
```

Version History

Introduced in R2021a

See Also

Functions

`fininstrument` | `finpricer`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Dupire

Create `Dupire` model object for local volatility for `Vanilla` instrument

Description

Create and price a `Vanilla` instrument object with a `Dupire` model using this workflow:

1. Use `fininstrument` to create a `Vanilla` instrument object.
2. Use `finmodel` to specify a `Dupire` model object for the `Vanilla` instrument object.
3. Use `finpricer` to specify a `FiniteDifference` pricing method for the `Vanilla` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Vanilla` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
DupireObj = finmodel(ModelType,'ImpliedVolData',impliedvoldata_value)
```

Description

`DupireObj = finmodel(ModelType,'ImpliedVolData',impliedvoldata_value)` creates a `Dupire` model object by specifying `ModelType` and the required name-value pair argument `ImpliedVolData` to set properties on page 11-3134 using name-value pair arguments. For example, `DupireObj = finmodel("Dupire",'ImpliedVolData',voldata_table)` creates a `Dupire` model object.

Input Arguments

`ModelType` — Model type

string with value `"Dupire"` | character vector with value `'Dupire'`

Model type, specified as the string with the value `"Dupire"` or the character vector with the value `'Dupire'`.

Data Types: `char` | `string`

`Dupire` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `DupireObj = finmodel("Dupire",'ImpliedVolData',voldata_table)`

`ImpliedVolData` — Table of maturity dates, strike or exercise prices, and corresponding implied volatilities
table

Table of maturity dates, strike or exercise prices, and their corresponding implied volatilities, specified as the comma-separated pair consisting of `'ImpliedVolData'` and an `NVOL`-by-3 table.

Data Types: `table`

Properties

`ImpliedVolData` — Table of maturity dates, strike or exercise prices, and corresponding implied volatilities
table

Table of maturity dates, strike or exercise prices, and corresponding implied volatilities, returned as an `NVOL`-by-3 table.

Data Types: `table`

Examples

Use Dupire Model and Finite Difference Pricer to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `Dupire` model and a `FiniteDifference` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2020,1,1),'Strike',105,'ExerciseSty
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2020
           Strike: 105
             Name: "vanilla_option"
```

Create `Dupire` Model Object

Define the implied volatility surface data.

```
AssetPrice = 590;
Maturity = ["06-Mar-2018" "05-Jun-2018" "12-Sep-2018" "10-Dec-2018" "01-Jan-2019" ...
"02-Jul-2019" "01-Jan-2020" "01-Jan-2021" "01-Jan-2022" "01-Jan-2023"];
Maturity = repmat(Maturity,10,1);
Maturity = Maturity(:);
```

```
ExercisePrice = AssetPrice.*[0.85 0.90 0.95 1.00 1.05 1.10 1.15 1.20 1.30 1.40];
ExercisePrice = repmat(ExercisePrice,1,10)';

ImpliedVol = [...
    0.190; 0.168; 0.133; 0.113; 0.102; 0.097; 0.120; 0.142; 0.169; 0.200; ...
    0.177; 0.155; 0.138; 0.125; 0.109; 0.103; 0.100; 0.114; 0.130; 0.150; ...
    0.172; 0.157; 0.144; 0.133; 0.118; 0.104; 0.100; 0.101; 0.108; 0.124; ...
    0.171; 0.159; 0.149; 0.137; 0.127; 0.113; 0.106; 0.103; 0.100; 0.110; ...
    0.171; 0.159; 0.150; 0.138; 0.128; 0.115; 0.107; 0.103; 0.099; 0.108; ...
    0.169; 0.160; 0.151; 0.142; 0.133; 0.124; 0.119; 0.113; 0.107; 0.102; ...
    0.169; 0.161; 0.153; 0.145; 0.137; 0.130; 0.126; 0.119; 0.115; 0.111; ...
    0.168; 0.161; 0.155; 0.149; 0.143; 0.137; 0.133; 0.128; 0.124; 0.123; ...
    0.168; 0.162; 0.157; 0.152; 0.148; 0.143; 0.139; 0.135; 0.130; 0.128; ...
    0.168; 0.164; 0.159; 0.154; 0.151; 0.147; 0.144; 0.140; 0.136; 0.132];

ImpliedVolData = table(Maturity, ExercisePrice, ImpliedVol);
```

Use `finmodel` to create a `Dupire` model object.

```
DupireModel = finmodel("Dupire",'ImpliedVolData',ImpliedVolData)
```

```
DupireModel =
  Dupire with properties:

    ImpliedVolData: [100x3 table]
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2020,9,1);
Rate = 0.06;
myRC = ratecurve('zero',Settle,Maturity,Rate)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: 01-Sep-2020
             Rates: 0.0600
            Settle: 01-Jan-2018
      InterpMethod: "linear"
  ShortExtrapMethod: "next"
   LongExtrapMethod: "previous"
```

Create `FiniteDifference` Pricer Object

Use `finpricer` to create a `FiniteDifference` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("FiniteDifference",'Model',DupireModel,'DiscountCurve',myRC,'SpotPrice',100
```

```
outPricer =
  FiniteDifference with properties:
```

```
     DiscountCurve: [1x1 ratecurve]
             Model: [1x1 finmodel.Dupire]
         SpotPrice: 100
    GridProperties: [1x1 struct]
      DividendType: "continuous"
     DividendValue: 0.0262
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])

Price = 15.5930

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results

ans=1×7 table
    Price      Delta       Gamma       Lambda     Theta      Rho      Vega
    _____    ________    _________    ______    ________    _____    _____

    15.593    0.55004    0.0091484    3.5275    -3.3431    78.792    49.33
```

More About

Local Volatility Model

A local volatility model treats volatility as a function both of the current asset level and of time.

The local volatility can be estimated by using the Dupire formula [2]:

$$\sigma_{loc}^2(K,\tau) = \frac{\sigma_{imp}^2 + 2\tau\sigma_{imp}\frac{\partial\sigma_{imp}}{\partial\tau} + 2(\tau - d)K\tau\sigma_{imp}\frac{\partial\sigma_{imp}}{\partial K}}{\left(1 + Kd_1\sqrt{\tau}\frac{\partial\sigma_{imp}}{\partial K}\right)^2 + K^2\tau\sigma_{imp}\left(\frac{\partial^2\sigma_{imp}}{\partial K^2} - d_1\sqrt{\tau}\left(\frac{\partial\sigma_{imp}}{\partial K}\right)^2\right)}$$

$$d_1 = \frac{\ln(S_0/K) + \left((\tau - d) + \sigma_{imp}^2/2\right)\tau}{\sigma_{imp}\sqrt{\tau}}$$

Version History

Introduced in R2020a

References

[1] Andersen, L. B., and R. Brotherton-Ratcliffe. "The Equity Option Volatility Smile: An Implicit Finite-Difference Approach." *Journal of Computational Finance*. Vol. 1, Number 2, 1997, pp. 5–37.

[2] Dupire, B. "Pricing with a Smile." *Risk*. Vol. 7, Number 1, 1994, pp. 18–20.

See Also

Functions
`fininstrument` | `finpricer`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

SABR

Create SABR model object for Swaption instrument

Description

Create and price a Swaption instrument object with a SABR model using this workflow:

1. Use fininstrument to create a Swaption instrument object.
2. Use finmodel to specify a SABR model object for the Swaption instrument object.
3. Use finpricer to specify a SABR pricing method for the Swaption instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a Swaption instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
SabrModelObj = finmodel(ModelType,'Alpha',alpha_value,'Beta',beta_value,'
Rho',rho_value,'Nu',nu_value)
SabrModelObj = finmodel(___,Name,Value)
```

Description

SabrModelObj = finmodel(ModelType,'Alpha',alpha_value,'Beta',beta_value,'Rho',rho_value,'Nu',nu_value) creates a SABR model object by specifying ModelType and sets the properties on page 11-3140 for the required name-value pair arguments Alpha, Beta, Rho, and Nu.

SabrModelObj = finmodel(___,Name,Value) sets optional properties on page 11-3140 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, SabrModelObj = finmodel("SABR",'Alpha',0.22,'Beta',0.007,'Rho',0.009,'Nu',0.03,'Shift',0.002,'VolatilityType',"black") creates a SABR model object. You can specify multiple name-value pair arguments.

Input Arguments

ModelType — Model type

string with value "SABR" | character vector with value 'SABR'

Model type, specified as a string with the value of "SABR" or a character vector with the value of 'SABR'.

Data Types: char | string

SABR Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `SabrModelObj = finmodel("SABR",'Alpha',0.22,'Beta',0.007,'Rho',0.009,'Nu',0.03,'Shift',0.002,'VolatilityType',"black")`

Required SABR Name-Value Pair Arguments

`Alpha` — Current SABR volatility
scalar numeric

Current SABR volatility, specified as the comma-separated pair consisting of `'Alpha'` and a scalar numeric.

Data Types: `double`

`Beta` — SABR constant elasticity of variance (CEV) exponent
scalar numeric

SABR constant elasticity of variance (CEV) exponent, specified as the comma-separated pair consisting of `'Beta'` and a scalar numeric.

Note Set the `Beta` parameter to `0` to allow a negative `ForwardValue` and `Strike`.

Data Types: `double`

`Rho` — Correlation between forward value and volatility
scalar numeric

Correlation between the forward value and volatility, specified as the comma-separated pair consisting of `'Rho'` and a scalar numeric.

Data Types: `double`

`Nu` — Volatility of volatility
scalar numeric

Volatility of volatility, specified as the comma-separated pair consisting of `'Nu'` and a scalar numeric.

Data Types: `double`

Optional SABR Name-Value Pair Arguments

`Shift` — Shift in decimals for shifted SABR model
`0` (no shift) (default) | scalar positive decimal

Shift in decimals for the shifted SABR model (to be used with the shifted Black model), specified as the comma-separated pair consisting of `'Shift'` and a scalar positive decimal value. For example, a `Shift` value of `0.01` is equal to a 1% positive shift.

Note If you set `VolatilityType` to `'normal'`, the `Shift` value must be `0`.

Data Types: `double`

`VolatilityType` — Model used by the implied volatility sigma
`"black"` (default) | string with value `"black"` or `"normal"` | character vector with value `'black'` or `'normal'`

Model used by the implied volatility sigma, specified as the comma-separated pair consisting of `'VolatilityType'` and a scalar string or character vector.

Note The value of `VolatilityType` affects the interpretation of the implied volatility ("sigma").

- If you set `VolatilityType` to `'black'` (default), “sigma” can be either implied Black (no shift) or implied shifted Black volatility.
- If you set `VolatilityType` to `'normal'`, “sigma” is the implied Normal (Bachelier) volatility and you must also set `Shift` to zero.

Data Types: `char` | `string`

Properties

`Alpha` — Current SABR volatility
scalar numeric

Current SABR volatility, returned as a scalar numeric.

Data Types: `double`

`Beta` — SABR constant elasticity of variance (CEV) exponent
scalar numeric

SABR constant elasticity of variance (CEV) exponent, returned as a scalar numeric.

Data Types: `double`

`Rho` — Correlation between forward value and volatility
scalar numeric

Correlation between forward value and volatility, returned as a scalar numeric.

Data Types: `double`

`Nu` — Volatility of volatility
scalar numeric

Volatility of volatility, returned as a scalar numeric.

Data Types: `double`

`Shift` — Shift in decimals for shifted SABR model
`0` (no shift) (default) | scalar positive decimal

Shift in decimals for the shifted SABR model (to be used with the Shifted Black model), returned as a scalar positive decimal value.

Data Types: `double`

`VolatilityType` — Model used by the implied volatility sigma
`"black"` (default) | string with value `"black"` or `"normal"`

Model used by the implied volatility sigma, returned as a string with a value of `"black"` or `"normal"`.

Data Types: `string`

Examples

Use SABR Model and SABR Pricer to Price Swaption Instrument

This example shows the workflow to price a `Swaption` instrument when you use a SABR model and a SABR pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create `Swap` Instrument Object

Use `fininstrument` to create the underlying `Swap` instrument object.

```
Swap = fininstrument("Swap",'Maturity',datetime(2023,1,30),'LegRate',[0.018 0.24],'LegType',["fi
```

```
Swap =
  Swap with properties:

                    LegRate: [0.0180 0.2400]
                    LegType: ["fixed"    "float"]
```

```
                    Reset: [2 2]
                    Basis: [5 5]
                 Notional: 1000
       LatestFloatingRate: [NaN NaN]
              ResetOffset: [0 0]
  DaycountAdjustedCashFlow: [1 1]
           ProjectionCurve: [1x2 ratecurve]
     BusinessDayConvention: ["follow"    "follow"]
                 Holidays: NaT
              EndMonthRule: [1 1]
                 StartDate: 30-Mar-2020
                  Maturity: 30-Jan-2023
                      Name: "swap_instrument"
```

Create Swaption Instrument Object

Use `fininstrument` to create a `Swaption` instrument object.

```
Swaption = fininstrument("swaption",'Strike',0.25,'ExerciseDate',datetime(2021,7,30),'Swap',Swap
```

```
Swaption =
  Swaption with properties:

      OptionType: "put"
  ExerciseStyle: "european"
   ExerciseDate: 30-Jul-2021
         Strike: 0.2500
           Swap: [1x1 fininstrument.Swap]
           Name: "swaption_instrument"
```

Create SABR Model Object

Use `finmodel` to create a SABR model object.

```
SabrModel = finmodel("SABR",'Alpha',0.032,'Beta',0.04, 'Rho', .08, 'Nu', 0.49,'Shift',0.002)
```

```
SabrModel =
  SABR with properties:

           Alpha: 0.0320
            Beta: 0.0400
             Rho: 0.0800
              Nu: 0.4900
           Shift: 0.0020
  VolatilityType: "black"
```

Create SABR Pricer Object

Use `finpricer` to create a SABR pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',SabrModel,'DiscountCurve',myRC)
```

```
outPricer =
  SABR with properties:
```

```
    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.SABR]
```

Price `Swaption` Instrument

Use `price` to compute the price for the `Swaption` instrument.

```
Price = price(outPricer,Swaption)
```

```
Price = 55.8596
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finpricer`

Topics

“Calibrate SABR Model Using Normal (Bachelier) Volatilities with Analytic Pricer” on page 2-177
“Calibrate SABR Model Using Analytic Pricer” on page 2-181
“Price a Swaption Using SABR Model and Analytic Pricer” on page 2-185
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Work with Negative Interest Rates Using Objects” on page 2-22

AssetMonteCarlo

Create `AssetMonteCarlo` pricer object for equity instruments using `BlackScholes`, `Merton`, `Heston`, or `Bates` model

Description

Create and price a `Vanilla`, `Barrier`, `Lookback`, `PartialLookback`, `Asian`, `Spread`, `DoubleBarrier`, `Cliquet`, `Touch`, `DoubleTouch`, `Binary` instrument object with a `BlackScholes`, `Bachelier`, `Merton`, `Heston`, or `Bates` model and a `AssetMonteCarlo` pricing method using this workflow:

1. Use `fininstrument` to create a `Vanilla`, `Barrier`, `Lookback`, `PartialLookback`, `Asian`, `Spread`, `DoubleBarrier`, `Cliquet`, `Binary`, `Touch`, or `DoubleTouch` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Vanilla`, `Barrier`, `Lookback`, `PartialLookback`, `Asian`, `Spread`, `DoubleBarrier`, `Cliquet`, `Touch`, `DoubleTouch`, or `Binary` instrument object.

   Use `finmodel` to specify a `Bachelier` model for the `Vanilla`, `Spread` or `Binary` instrument object.

   Use `finmodel` to specify a `Merton`, `Bates`, or `Heston` model for the `Vanilla`, `Barrier`, `Lookback`, `PartialLookback`, `Asian`, `DoubleBarrier`, `Touch`, `DoubleTouch`, `Cliquet`, or `Binary` instrument object.
3. Use `finpricer` to specify an `AssetMonteCarlo` pricer object for the `Vanilla`, `Barrier`, `Lookback`, `PartialLookback`, `Asian`, `Spread`, `DoubleBarrier`, `Cliquet`, `Touch`, `DoubleTouch`, or `Binary` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for `Vanilla`, `Barrier`, `Lookback`, `PartialLookback`, `Asian`, `Spread`, `DoubleBarrier`, `Cliquet`, `Touch`, `DoubleTouch`, or `Binary` instruments, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
AssetMonteCarloPricerObj = finpricer(PricerType,'Model',model,'
DiscountCurve',ratecurve_obj,'SpotPrice',spotprice_value,'
SimulationDates',simulation_dates)
AssetMonteCarloPricerObj = finpricer( ___ ,Name,Value)
```

Description

```
AssetMonteCarloPricerObj = finpricer(PricerType,'Model',model,'
DiscountCurve',ratecurve_obj,'SpotPrice',spotprice_value,'
```

`SimulationDates',simulation_dates)` creates a `AssetMonteCarlo` pricer object by specifying `PricerType` and sets the properties on page 11-3147 using the required name-value pair arguments `Model`, `DiscountCurve`, `SpotPrice`, and `SimulationDates`.

`AssetMonteCarloPricerObj = finpricer( ___ ,Name,Value)` sets optional properties on page 11-3147 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `AssetMonteCarloPricerObj = finpricer("assetmontecarlo",'Model',BSModel,'DiscountCurve',ratecurve_obj,'SpotPrice',1000,'SimulationDates',[datetime(2018,1,30); datetime(2019,1,30)],'NumTrials',500,'DividendType','continuous','DividendValue',0.3)` creates an `AssetMonteCarlo` pricer object using a `BlackScholes` model. You can specify multiple name-value pair arguments.

Input Arguments

`PricerType` — Pricer type

string with value `"AssetMonteCarlo"` | character vector with value `'AssetMonteCarlo'`

Pricer type, specified as a string with the value `"AssetMonteCarlo"` or a character vector with the value `'AssetMonteCarlo'`.

Data Types: `char` | `string`

`AssetMonteCarlo` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `AssetMonteCarloPricerObj = finpricer("assetmontecarlo",'Model',BSModel,'DiscountCurve',ratecurve_obj,'SpotPrice',1000,'SimulationDates',[datetime(2018,1,30); datetime(2019,1,30)],'NumTrials',500,'DividendType','continuous','DividendValue',0.3)`

Required AssetMonteCarlo Name-Value Pair Arguments

`Model` — Model

`BlackScholes` object | `Merton` object | `Bates` object | `Heston` object

Model, specified as the comma-separated pair consisting of `'Model'` and the name of a previously created `BlackScholes`, `Merton`, `Bates`, or `Heston` model object. Create the model object using `finmodel`.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows

`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a previously created `ratecurve` object.

Note Specify a flat `ratecurve` object for `DiscountCurve`. If you use a nonflat `ratecurve` object, the software uses the rate in the `ratecurve` object at `Maturity` and assumes that the value is constant for the life of the equity option.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric | positive or negative numeric when using `Bachelier` model

Current price of the underlying asset, specified as the comma-separated pair consisting of `'SpotPrice'` and a scalar nonnegative numeric or scalar positive or negative numeric when using `Bachelier` model.

Note If you use a `Vanilla`, `Binary`, or `Spread` instrument with a `Bachelier` model, the `SpotPrice` can be a negative numeric value.

Data Types: `double`

`SimulationDates` — Simulation dates
`[ ]` (default) | datetime array | string array | date character vector

Simulation dates, specified as the comma-separated pair consisting of `'SimulationDates'` and a scalar or a vector using a datetime array, string array, or date character vectors.

To support existing code, `AssetMonteCarlo` also accepts serial date numbers as inputs, but they are not recommended.

Optional AssetMonteCarlo Name-Value Pair Arguments

`NumTrials` — Simulation trials
`1000` (default) | scalar

Simulation trials, specified as the comma-separated pair consisting of `'NumTrials'` and a scalar number of independent sample paths.

Data Types: `double`

`RandomNumbers` — Dependent random variates
`[ ]` (default) | structure

Dependent random variates, specified as the comma-separated pair consisting of `'RandomNumbers'` and an `NSimulationDates`-by-`NBrownians`-by-`NTrials` 3D time series array. The 3D time series array has the following fields:

- `Z` — `NSimulationDates`-by-`NBrownians`-by-`NTrials` 3D time series array of dependent random variates used to generate the Brownian motion vector (that is, Wiener processes) that drive the simulation.
- `N` — `NSimulationDates`-by-`NBrownians`-by-`NTrials` 3D time series array of dependent random variates used as the number of jumps.
- `SizeJ` — `NSimulationDates`-by-`NBrownians`-by-`NTrials` 3D time series array of dependent random variates used as the jump sizes.

Note `BlackScholes` and `Heston` models only require `Z` field.

Data Types: `struct`

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"cash"` or `"continuous"` | character vector with value `'cash'` or `'continuous'`

Stock dividend type, specified as the comma-separated pair consisting of `'DividendType'` and a character vector or string. `DividendType` must be either `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `char` | `string`

`DividendValue` — Dividend yield or dividend schedule for underlying stock
`0` (default) | scalar numeric | timetable

Dividend yield for the underlying stock, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar numeric for a dividend yield or a timetable for a dividend schedule.

Note Specify a scalar if `DividendType` is `"continuous"` and a timetable if `DividendType` is `"cash"`.

Data Types: `double` | `timetable`

Properties

`Model` — Model
object

Model, returned as an object.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

This property is read-only.

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric | positive or negative numeric when using `Bachelier` model

Current price of underlying asset, returned as a scalar nonnegative numeric or a scalar positive or negative numeric when using `Bachelier` model.

Data Types: `double`

`SimulationDates` — Simulation dates
datetime

Simulation dates, returned as a datetime array.

Data Types: `datetime`

`NumTrials` — Simulation trials
`1000` (default) | scalar

Simulation trials, returned as a scalar number of independent sample paths.

Data Types: `double`

`RandomNumbers` — Dependent random variates
`[ ]` (default) | structure

Dependent random variates, returned as an `NSimulationDates`-by-`NBrownians`-by-`NTrials` 3D time series array.

Data Types: `struct`

`EarlyExerciseFunction` — Calculation for early exercise premium
`@longstaffschwartz_cubic` (default) | function handle

Calculation for the early exercise premium, returned as a scalar function handle. The default `@longstaffschwartz_cubic` uses the Longstaff-Schwartz least squares method.

Data Types: `function_handle`

`DividendType` — Dividend type
`"continuous"` (default) | string with value `"cash"` or `"continuous"`

This property is read-only.

Dividend type, returned as a string. `DividendType` is either `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `string`

`DividendValue` — Dividend yield or dividend schedule for underlying stock
`0` (default) | scalar nonnegative numeric | timetable

Dividend yield or dividend schedule for the underlying stock, returned as a scalar numeric for a dividend yield or a timetable for a dividend schedule.

Data Types: `double` | `timetable`

Object Functions

price Compute price for equity instrument with AssetMonteCarlo pricer

Examples

Use Asset Monte-Carlo Pricer and Black-Scholes Model to Price Double Barrier Instrument

This example shows the workflow to price a `DoubleBarrier` instrument when you use a `BlackScholes` model and an `AssetMonteCarlo` pricing method.

Create DoubleBarrier Instrument Object

Use `fininstrument` to create a `DoubleBarrier` instrument object.

```
DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',100,'ExerciseDate',datetime(2020,8,15),
```

```
DoubleBarrierOpt =
  DoubleBarrier with properties:

       OptionType: "call"
           Strike: 100
     BarrierValue: [110 80]
    ExerciseStyle: "american"
     ExerciseDate: 15-Aug-2020
      BarrierType: "dko"
           Rebate: [0 0]
             Name: "doublebarrier_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes","Volatility",0.3)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2017,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                 Type: "zero"
          Compounding: -1
                Basis: 12
                Dates: 15-Sep-2023
                Rates: 0.0350
               Settle: 15-Sep-2017
         InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
ExerciseDate = datetime(2020,08,15);
Settle = datetime(2017,9,15);
outPricer = finpricer("AssetMonteCarlo","DiscountCurve",myRC,"Model",BlackScholesModel,'SpotPrice
```

Price DoubleBarrier Instrument

Use `price` to compute the price and sensitivities for the `DoubleBarrier` instrument.

```
[Price, outPR] = price(outPricer,DoubleBarrierOpt,"all")
```

```
Price = 6.9667

outPR =
  priceresult with properties:

      Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Lambda      Rho       Theta      Vega
    ______    _______    _________    ______    _______    ______    _______

    6.9667    0.26875    -0.096337    3.8576    0.39855    9.5406    -1.2907
```

Use Asset Monte-Carlo Pricer and Heston Model to Price Asian Instrument

This example shows the workflow to price a fixed-strike `Asian` instrument when you use a `Heston` model and an `AssetMonteCarlo` pricing method.

Create Asian Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',100,'OptionType',"pu
```

```
AsianOpt =
  Asian with properties:

         OptionType: "put"
             Strike: 100
        AverageType: "arithmetic"
       AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_option"
```

Create Heston Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.02,'RhoSV',0.9)
```

```
HestonModel =
  Heston with properties:

       V0: 0.0320
    ThetaV: 0.1000
     Kappa: 0.0030
    SigmaV: 0.0200
     RhoSV: 0.9000
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
              Dates: 15-Sep-2023
              Rates: 0.0350
             Settle: 15-Sep-2018
       InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",HestonModel,'SpotPrice',80,
```

```
outPricer =
  HestonMonteCarlo with properties:

     DiscountCurve: [1x1 ratecurve]
         SpotPrice: 80
    SimulationDates: [15-Oct-2018    15-Nov-2018    15-Dec-2018    ...    ]
         NumTrials: 1000
     RandomNumbers: []
             Model: [1x1 finmodel.Heston]
      DividendType: "continuous"
     DividendValue: 0
```

Price `Asian` Instrument

Use `price` to compute the price and sensitivities for the `Asian` instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,"all")
```

```
Price = 14.7999

outPR =
  priceresult with properties:

      Results: [1x8 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×8 table
    Price     Delta       Gamma       Lambda       Rho       Theta      Vega     VegaLT
    _____    ________    ________    _______    _______    _______    ______    _______

    14.8     -0.71073    0.023453    -3.8418    -173.12    0.61794    27.992    0.15319
```

Use Asset Monte-Carlo Pricer and Bachelier Model to Price American Option for Vanilla Instrument

This example shows the workflow to price an American option for a `Vanilla` instrument when you use a `Bachelier` model and an `AssetMonteCarlo` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'Strike',105,'ExerciseDate',datetime(2022,9,15),'OptionType
```

```
VanillaOpt =
  Vanilla with properties:

      OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2022
           Strike: 105
             Name: "vanilla_option"
```

Create `Bachelier` Model Object

Use `finmodel` to create a `Bachelier` model object.

```
BachelierModel = finmodel("Bachelier","Volatility",0.2)
```

```
BachelierModel =
  Bachelier with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `AssetMonteCarlo` Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BachelierModel,'SpotPrice',1
```

```
outPricer =
  BachelierMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: 150
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Bachelier]
       DividendType: "continuous"
      DividendValue: 0
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 57.3776

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Lambda     Rho       Theta        Vega
    ______    _______    __________    ______    ______    _______    __________
```

```
    57.378    0.99107    -1.579e-14    2.5909    291.94    -2.5576    -2.1316e-10
```

Use Asset Monte-Carlo Pricer and Bachelier Model to Price Binary Instrument With Underlying Negative Value

This example shows the workflow to price a `Binary` instrument with an underlying negatively valued asset when you use a `Bachelier` model and an `AssetMonteCarlo` pricing method.

Create `Binary` Instrument Object

Use `fininstrument` to create a `Binary` instrument object.

```
BinaryOpt = fininstrument("Binary",'ExerciseDate',datetime(2022,9,15),'Strike',15,'PayoffValue',1
```

```
BinaryOpt =
  Binary with properties:

       OptionType: "put"
     ExerciseDate: 15-Sep-2022
           Strike: 15
      PayoffValue: 13
    ExerciseStyle: "european"
             Name: "binary_option"
```

Create `Bachelier` Model Object

Use `finmodel` to create a `Bachelier` model object.

```
BachelierModel = finmodel("Bachelier","Volatility",0.2)
```

```
BachelierModel =
  Bachelier with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
        Compounding: -1
              Basis: 12
              Dates: 15-Sep-2023
              Rates: 0.0350
```

```
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetMonteCarlo Pricer Object

Use `finpricer` to create an `AssetMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument. Note that when using a Bachelier model with a Vanilla, Binary, or Spread instrument, the `SpotPrice` can be a positive or negative numeric value.

```
outPricer = finpricer("AssetMonteCarlo",'DiscountCurve',myRC,"Model",BachelierModel,'SpotPrice',
```

```
outPricer =
  BachelierMonteCarlo with properties:

      DiscountCurve: [1x1 ratecurve]
          SpotPrice: -6
    SimulationDates: 15-Sep-2022
          NumTrials: 1000
      RandomNumbers: []
              Model: [1x1 finmodel.Bachelier]
       DividendType: "continuous"
      DividendValue: 0
```

Price Binary Instrument

Use `price` to compute the price and sensitivities for the `Binary` instrument.

```
[Price, outPR] = price(outPricer,BinaryOpt,["all"])
```

```
Price = 11.3017

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta    Gamma    Lambda      Rho       Theta      Vega
    ______    _____    _____    ______    _______    _______    ____

    11.302      0        0        0       -45.198    0.39582     0
```

Version History

Introduced in R2020b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `AssetMonteCarlo` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions
`fininstrument` | `finmodel`

Topics
“Use Deep Learning to Approximate Barrier Option Prices with Heston Model” on page 3-148
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

HeynenKat

Create `HeynenKat` pricer object for `PartialLookback` instrument using `BlackScholes` model

Description

Create and price a `PartialLookback` instrument object with a `BlackScholes` model and a `HeynenKat` pricing method using this workflow:

1. Use `fininstrument` to create a `PartialLookback` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `PartialLookback` instrument object.
3. Use `finpricer` to specify a `HeynenKat` pricer object for the `PartialLookback` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for an `PartialLookback` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
HeynenKatPricerObj = finpricer(PricerType,DiscountCurve=ratecurve_obj,
Model=model,SpotPrice=spotprice_value)
HeynenKatPricerObj = finpricer(___,Name=Value)
```

Description

`HeynenKatPricerObj = finpricer(PricerType,DiscountCurve=ratecurve_obj, Model=model,SpotPrice=spotprice_value)` creates a `HeynenKat` pricer object by specifying `PricerType` and sets the properties on page 11-3159 for the required name-value arguments `DiscountCurve`, `Model`, and `SpotPrice`.

`HeynenKatPricerObj = finpricer(___,Name=Value)` to set optional properties on page 11-3159 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `HeynenKatPricerObj = finpricer("Analytic",DiscountCurve=ratecurve_obj,Model=BSModel,SpotPrice=1000,DividendType="continuous",DividendValue=100,PricingMethod="HeynenKat")` creates a `HeynenKat` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: `char` | `string`

HeynenKat Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `HeynenKatPricerObj = finpricer("Analytic",DiscountCurve=ratecurve_obj,Model=BSModel,SpotPrice=1000,DividendType="continuous",DividendValue=100,PricingMethod="HeynenKat")`

Required HeynenKat Name-Value Arguments

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as `DiscountCurve` and the name of the previously created `ratecurve` object.

Note Specify a flat `ratecurve` object for `DiscountCurve`. If you use a nonflat `ratecurve` object, the software uses the rate in the `ratecurve` object at `Maturity` and assumes that the value is constant for the life of the equity option.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, specified as `Model` and the name of a previously created `BlackScholes` model object using `finmodel`.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, specified as `SpotPrice` and a scalar nonnegative numeric.

Data Types: `double`

Optional HeynenKat Name-Value Arguments

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"continuous"` | character vector with value `'continuous'`

Stock dividend type, specified as `DividendType` and a string or character vector.

Data Types: `char` | `string`

`DividendValue` — Dividend yield for underlying stock
`0` (default) | scalar numeric

Dividend yield for the underlying stock, specified as `DividendValue` and a scalar numeric.

Data Types: `double`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"HeynenKat"` | character vector with value `'HeynenKat'`

Analytic pricing method, specified as `PricingMethod` and a string or character vector.

Note The default pricing method for a `BlackScholes` model is a `BlackScholes` pricer.

Data Types: `double`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
object

This property is read-only.

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"continuous"`

This property is read-only.

Stock dividend type, returned as a string.

Data Types: `string`

`DividendValue` — Dividend yield for underlying stock
`0` (default) | scalar nonnegative numeric

Dividend yield for the underlying stock, returned as a scalar numeric.

Data Types: `double`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"HeynenKat"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Heynen-Kat Pricer and Black-Scholes Model to Price Partial Lookback Instrument

This example shows the workflow to price a `PartialLookback` instrument when you use a `BlackScholes` model and a `HeynenKat` pricing method.

Create `PartialLookback` Instrument Object

Use `fininstrument` to create an `PartialLookback` instrument object.

```
PartialLookbackOpt = fininstrument("PartialLookback",ExerciseDate=datetime(2022,9,15),Strike=100
```

```
PartialLookbackOpt =
  PartialLookback with properties:

     MonitorDate: 15-Sep-2021
    StrikeScaler: 1
      OptionType: "put"
          Strike: 100
     AssetMinMax: NaN
   ExerciseStyle: "european"
    ExerciseDate: 15-Sep-2022
            Name: "partial_lookback_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",Volatility=0.32)
```

```
BlackScholesModel =
  BlackScholes with properties:

    Volatility: 0.3200
   Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,Basis=12)
```

```
myRC =
  ratecurve with properties:
```

```
                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create HeynenKat Pricer Object

Use `finpricer` to create a `HeynenKat` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",Model=BlackScholesModel,DiscountCurve=myRC,SpotPrice=100,Divider
```

```
outPricer =
  HeynenKat with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0500
     DividendType: "continuous"
```

Price PartialLookback Instrument

Use `price` to compute the price and sensitivities for the `PartialLookback` instrument.

```
[Price, outPR] = price(outPricer,PartialLookbackOpt,["all"])
```

```
Price = 31.4405

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta        Gamma       Lambda      Vega      Theta       Rho
    _____    ________    _________    _______    ______    _______    _______

    31.44    -0.37693    0.0042263    -1.1989    76.886    -1.6249    -259.77
```

Version History

Introduced in R2021b

See Also

Functions
`fininstrument` | `finmodel` | `ratecurve`

Topics
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

IkedaKunitomo

Create `IkedaKunitomo` pricer object for `DoubleBarrier` instrument using `BlackScholes` model

Description

Create and price a `DoubleBarrier` instrument object with a `BlackScholes` model and a `IkedaKunitomo` pricing method using this workflow:

1. Use `fininstrument` to create a `DoubleBarrier` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `DoubleBarrier` instrument object.
3. Use `finpricer` to specify a `IkedaKunitomo` pricer object for the `DoubleBarrier` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `DoubleBarrier` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
IkedaKunitomoPricerObj = finpricer(PricerType,'Model',model,'
DiscountCurve',ratecurve_obj,'SpotPrice',spotprice_value)
IkedaKunitomoPricerObj = finpricer( ___ ,Name,Value)
```

Description

`IkedaKunitomoPricerObj = finpricer(PricerType,'Model',model,'DiscountCurve',ratecurve_obj,'SpotPrice',spotprice_value)` creates a `IkedaKunitomo` pricer object by specifying `PricerType` and sets properties on page 11-3165 using the required name-value pair arguments `Model`, `DividendType`, and `SpotPrice`.

`IkedaKunitomoPricerObj = finpricer( ___ ,Name,Value)` sets optional properties on page 11-3165 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `IkedaKunitomoPricerObj = finpricer("Analytic",'Model',BSModel,'DiscountCurve',ratecurve_obj,'SpotPrice',100,'DividendValue',0.025,'PricingMethod',"IkedaKunitomo")` creates a `IkedaKunitomo` pricer object. You can specify multiple name-value pair arguments.

Input Arguments

PricerType — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value `"Analytic"` or a character vector with the value `'Analytic'`.

Data Types: `char` | `string`

`IkedaKunitomo` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `IkedaKunitomoPricerObj = finpricer("Analytic",'Model',BSModel,'DiscountCurve',ratecurve_obj,'SpotPrice',100,'DividendValue',0.025,'PricingMethod',"IkedaKunitomo")`

Required IkedaKunitomo Name-Value Pair Arguments

`Model` — `IkedaKunitomo` model object
object

Model object, specified as the comma-separated pair consisting of `'Model'` and the name of a previously created `BlackScholes` model object using `finmodel`.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of the previously created `ratecurve` object.

Note Specify a flat `ratecurve` object for `DiscountCurve`.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
scalar nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of `'SpotPrice'` and a scalar nonnegative numeric value.

Data Types: `double`

Optional IkedaKunitomo Name-Value Pair Arguments

`DividendValue` — Dividend yield
`0` (default) | scalar nonnegative numeric

Dividend yield, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar nonnegative numeric value.

Data Types: `double`

`DividendType` — Dividend type
`"continuous"` (default) | string with value `"continuous"` | character vector with value `'continuous'`

Dividend type, specified as the comma-separated pair consisting of `'DividendType'` and string or character vector.

Data Types: `char` | `string`

`Curvature` — Curvature levels of the upper and lower barriers
`[0 0]` (default) | vector

Curvature levels of the upper and lower barriers, specified as the comma-separated pair consisting of `'Curvature'` and a vector where the first level is the upper barrier curvature (d1) and second level is the lower barrier curvature (d2). The possible curvature levels are as follows:

- d1 = d2 = 0 corresponds to two flat boundaries.
- d1 < 0 < d2 corresponds to an exponentially growing lower boundary and an exponentially decaying upper boundary.
- d1 > 0 > d2 corresponds to a convex downward lower boundary and a convex upward upper boundary.

Data Types: `char` | `string`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"IkedaKunitomo"` | character vector with value `'IkedaKunitomo'`

Analytic pricing method, specified as the comma-separated pair consisting of `'PricingMethod'` and a character vector or string.

Note The default pricing method for a `BlackScholes` model is a `BlackScholes` pricer.

Data Types: `string` | `char`

Properties

`Model` — Model
model object

Model, returned as a model object.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

This property is read-only.

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric value.

Data Types: `double`

`DividendValue` — Dividend yield
`0` (default) | scalar nonnegative numeric

Dividend yield, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendType` — Dividend type
`"continuous"` (default) | string with value `"continuous"`

This property is read-only.

Dividend type, returned as a string.

Data Types: `string`

`Curvature` — Curvature levels of the upper and lower barriers
`[0 0]` (default) | vector

Curvature levels of the upper and lower barriers, returned as a vector where the first level is the upper barrier curvature (d1) and second level is the lower barrier curvature (d2).

Data Types: `char` | `string`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"IkedaKunitomo"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Ikeda-Kunitomo Pricer and Black-Scholes Model to Price Double Barrier Instrument

This example shows the workflow to price a `DoubleBarrier` instrument when you use a `BlackScholes` model and an `IkedaKunitomo` pricing method.

Create `DoubleBarrier` Instrument Object

Use `fininstrument` to create a `DoubleBarrier` instrument object.

```
DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',100,'ExerciseDate',datetime(2020,8,15),
```

```
DoubleBarrierOpt =
  DoubleBarrier with properties:

       OptionType: "call"
           Strike: 100
     BarrierValue: [110 80]
    ExerciseStyle: "european"
```

```
     ExerciseDate: 15-Aug-2020
      BarrierType: "dko"
           Rebate: [0 0]
             Name: "doublebarrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes","Volatility",.3)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2017,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2017
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `IkedaKunitomo` Pricer Object

Use `finpricer` to create an `IkedaKunitomo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Analytic","DiscountCurve",myRC,"Model",BlackScholesModel,'SpotPrice',100,
```

```
outPricer =
  IkedaKunitomo with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0290
     DividendType: "continuous"
        Curvature: [0.0300 -0.0300]
```

Price DoubleBarrier Instrument

Use `price` to compute the price and sensitivities for the `DoubleBarrier` instrument.

```
[Price, outPR] = price(outPricer,DoubleBarrierOpt,["all"])
```

```
Price = 5.6848e-04
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
      Price          Delta          Gamma        Lambda       Vega          Theta          Rho
    __________    ___________    ___________    _______    _________    _________    ___________

    0.00056848    -3.7713e-05    -4.2071e-06    -6.6339    -0.031332    0.0008912    -0.00035113
```

Version History

Introduced in R2020b

See Also

Functions

`fininstrument` | `finmodel`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

VannaVolga

Create `VannaVolga` pricer object for `Vanilla`, `Barrier`, `DoubleBarrier`, `Touch`, or `DoubleTouch` instrument using `BlackScholes` model

Description

Create and price a `Vanilla`, `Barrier`, `DoubleBarrier`, `Touch`, or `DoubleTouch` instrument object with a `BlackScholes` model and a `VannaVolga` pricing method using this workflow:

1. Use `fininstrument` to create a `Vanilla`, `Barrier`, `DoubleBarrier`, `Touch`, or `DoubleTouch` instrument object.
2. Use `finmodel` to specify the `BlackScholes` model for the `Vanilla`, `Barrier`, `DoubleBarrier`, `Touch`, or `DoubleTouch` instrument object.
3. Use `finpricer` to specify the `VannaVolga` pricer object for the `Vanilla`, `Barrier`, `DoubleBarrier`, `Touch`, or `DoubleTouch` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Vanilla`, `Barrier`, `DoubleBarrier`, `Touch`, or `DoubleTouch` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
VannaVolgaPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model,'SpotPrice',spot_price,'VolatilityRR',volatilityrr_value,'
VolatilityBF',volatilitybf_value)
```

Description

`VannaVolgaPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model,'SpotPrice',spot_price,'VolatilityRR',volatilityrr_value,'VolatilityBF',volatilitybf_value)` creates a `VannaVolga` pricer object by specifying `PricerType` and sets properties on page 11-3171 using the required name-value pair arguments `DiscountCurve`, `Model`, `SpotPrice`, `VolatilityRR`, and `VolatilityBF`. For example, `VannaVolgaPricerObj = finpricer("VannaVolga",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',Spot,'VolatilityRR',VolRR,'VolatilityBF',VolBF)` creates a `VannaVolga` pricer object.

`VannaVolgaPricerObj = finpricer(___,Name,Value)` sets optional properties on page 11-3171 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `VannaVolgaPricerObj = finpricer("VannaVolga",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPri`

ce','Spot','VolatilityRR',VolRR,'VolatilityBF',VolBF,'DividendValue',0.0210) creates a VannaVolga pricer object. You can specify multiple name-value pair arguments.

Input Arguments

PricerType — Pricer type

string with value "VannaVolga" | character vector with value 'VannaVolga'

Pricer type, specified as a string with the value "VannaVolga" or a character vector with the value 'VannaVolga'.

Data Types: char | string

Required VannaVolga Name-Value Pair Arguments

Specify required pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: VannaVolgaPricerObj = finpricer("VannaVolga",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',Spot,'VolatilityRR',VolRR,'VolatilityBF',VolBF,'DividendValue',0.0210)

DiscountCurve — ratecurve object for discounting cash flows

ratecurve object

ratecurve object for discounting cash flows, specified as the comma-separated pair consisting of 'DiscountCurve' and the name of a previously created ratecurve object.

Data Types: object

Model — Model object

BlackScholes model object

Model object, specified as the comma-separated pair consisting of 'Model' and the name of a previously created BlackScholes model object using finmodel.

Data Types: object

SpotPrice — Current price of underlying asset

numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of 'SpotPrice' and a scalar numeric.

Data Types: double

VolatilityRR — 25-delta risk reversal (RR) volatility

numeric

25-delta risk reversal (RR) volatility, specified as the comma-separated pair consisting of 'VolatilityRR' and a scalar numeric.

Data Types: double

`VolatilityBF` — 25-delta butterfly (BF) volatility
numeric

25-delta butterfly (BF) volatility, specified as the comma-separated pair consisting of `'VolatilityBF'` and a scalar numeric.

Data Types: `double`

Optional VannaVolga Name-Value Pair Arguments

`DividendType` — Dividend type
`"continuous"` (default) | string with value of `"continuous"` | character vector with value of `'continuous'`

Dividend type, specified as the comma-separated pair consisting of `'DividendType'` and a string or character vector for a continuous dividend yield.

Data Types: `char` | `string`

`DividendValue` — Continuous dividend yield
`0` (default) | scalar numeric

Continuous dividend yield, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar numeric.

Note When pricing currency (FX) options, specify the optional input argument `'DividendValue'` as the continuously compounded risk-free interest rate in the foreign country.

Data Types: `double`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
numeric

Current price of the underlying asset, returned as a scalar numeric.

Data Types: `double`

`VolatilityRR` — 25-delta risk reversal (RR) volatility
numeric

25-delta risk reversal (RR) volatility, returned as a scalar numeric.

Data Types: `double`

`VolatilityBF` — 25-delta butterfly (BF) volatility
numeric

25-delta butterfly (BF) volatility, returned as a scalar numeric.

Data Types: `double`

`DividendType` — Dividend type
`"continuous"` (default) | string with value of `"continuous"`

This property is read-only.

Dividend type, returned as a string.

Data Types: `string`

`DividendValue` — Continuous dividend yield
`0` (default) | scalar numeric

Continuous dividend yield, returned as a scalar numeric.

Data Types: `double`

Object Functions

price Compute price for equity instrument with VannaVolga pricer

Examples

Use Vanna Volga Pricer and Black-Scholes Model to Price Double Barrier Instrument

This example shows the workflow to price a `DoubleBarrier` instrument when you use a `BlackScholes` model and a `VannaVolga` pricing method.

Create `DoubleBarrier` Instrument Object

Use `fininstrument` to create a `DoubleBarrier` instrument object.

```
DoubleBarrierOpt = fininstrument("DoubleBarrier",'Strike',100,'ExerciseDate',datetime(2020,8,15),
```

```
DoubleBarrierOpt =
  DoubleBarrier with properties:

       OptionType: "call"
           Strike: 100
     BarrierValue: [110 80]
    ExerciseStyle: "european"
     ExerciseDate: 15-Aug-2020
      BarrierType: "dko"
           Rebate: [0 0]
             Name: "doublebarrier_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes","Volatility",0.02)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.0200
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `VannaVolga` Pricer Object

Use `finpricer` to create a `VannaVolga` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
VolRR = -0.0045;
VolBF = 0.0037;
RateF = 0.0210;
outPricer = finpricer("VannaVolga","DiscountCurve",myRC,"Model",BlackScholesModel,'SpotPrice',100
```

```
outPricer =
  VannaVolga with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
     DividendType: "continuous"
    DividendValue: 0.0210
     VolatilityRR: -0.0045
     VolatilityBF: 0.0037
```

Price DoubleBarrier Instrument

Use `price` to compute the price and sensitivities for the `DoubleBarrier` instrument.

```
[Price, outPR] = price(outPricer,DoubleBarrierOpt,["all"])
```

```
Price = 1.6450

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta     Gamma     Lambda     Vega      Theta      Rho
    _____    _______    _____    ______    ______    _______    _____

    1.645    0.82818    75.662    50.346    14.697    -1.3145    74.666
```

More About

Vanna-Volga Method

The Vanna Volga method is an empirical procedure based on adding an analytically derived correction to the Black-Scholes price of the instrument.

The Vanna-Volga method consists of adjusting the Black-Scholes theoretical value (BSTV) by the cost of a portfolio which hedges three main risks associated to the volatility of the option: the Vega, the Vanna and the Volga.

The general formulation of the Vanna-Volga method suggests that the Vega, Vanna, and Volga values can be replicated by the weighted sum of at-the-money (ATM), risk-reversal (RR), and butterfly (BF) strategies.

$$X_i = \omega_{ATM} ATM_i + \omega_{RR} RR_i + \omega_{BF} BF_i \; (i = vega, vanna, volga)$$

Here, the weights are obtained by solving the system $x = A\omega$

$$A = \begin{matrix} ATM_{vega} & RR_{vega} & BF_{vega} \\ ATM_{vanna} & RR_{vanna} & BF_{vanna,} \\ ATM_{volga} & RR_{volga} & BF_{volga} \end{matrix} \quad \omega = \begin{matrix} \omega_{ATM} \\ \omega_{RR,} \\ \omega_{BF} \end{matrix} \quad X = \begin{matrix} X_{vega} \\ X_{vanna} \\ X_{volga} \end{matrix}$$

Given this replication, the Vanna-Volga method adjusts the Black-Scholes price of the option by the smile cost of the above weighted sum:

$$X^{VV} = X^{BS} + \omega_{ATM}(ATM^{mkt} - ATM^{BS}) + \omega_{RR}(RR^{mkt} - RR^{BS}) + \omega_{BF}(BF^{mkt} - BF^{BS})$$

$$= X^{BS} + x^{T}(A^{T}I$$

$$= X^{BS} + X_{vega}\Omega_{vega} + X_{vanna}\Omega_{vanna} + X_{volga}\Omega_{volga}$$

where $I = \begin{matrix} ATM^{mkt} - ATM^{BS} \\ RR^{mkt} - RR^{BS} \\ BF^{mkt} - BF^{mkt} \end{matrix}$, $\Omega_{vanna} = (A^{T}I$ Ω_{vega} Ω_{volga}

The resulting correction or overhedge turns out to be too large. So, the option value is modified as follows:

$$X^{VV} = X^{BS} + X_{vega}\Omega_{vega} + X_{vanna}\Omega_{vanna} + X_{volga}\Omega_{volga}$$

where, $\rho_{vega} = \frac{1}{2} + \frac{1}{2}\rho$, and $\rho_{volga} = \frac{1}{2} + \frac{1}{2}\rho$.

ρ is the risk-neutral probability of not hitting the barrier.

Version History

Introduced in R2020b

References

[1] Bossens, Frédéric, Grégory Rayée, Nikos S. Skantzos, and Griselda Deelstra. "Vanna-Volga Methods Applied to FX Derivatives: From Theory to Market Practice." *International Journal of Theoretical and Applied Finance.* 13, no. 08 (December 2010): 1293–1324.

[2] Castagna, Antonio, and Fabio Mercurio. "The Vanna-Volga Method for Implied Volatilities." *Risk.* 20 (January 2007): 106–111.

[3] Castagna, Antonio, and Fabio Mercurio. "Consistent Pricing of FX Options." Working Papers Series, Banca IMI, 2006.

[4] Fisher, Travis. "Variations on the Vanna-Volga Adjustment." Bloomberg Research Paper, 2007.

[5] Wystup, Uwe. *FX Options and Structured Products.* Hoboken, NJ: Wiley Finance, 2006.

See Also

Functions

`fininstrument` | `finmodel` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22

“Choose Instruments, Models, and Pricers” on page 1-53

Heston

Create `Heston` pricer object for `VarianceSwap` instrument using `Heston` model

Description

Create and price a `VarianceSwap` instrument object with a `Heston` model and a `Heston` pricing method using this workflow:

1. Use `fininstrument` to create a `VarianceSwap` instrument object.
2. Use `finmodel` to specify the `Heston` model for the `VarianceSwap` instrument object.
3. Use `finpricer` to specify the `Heston` pricer object for the `VarianceSwap` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `VarianceSwap` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
HestonPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model)
```

Description

`HestonPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model)` creates a `Heston` pricer object by specifying `PricerType` and sets properties on page 11-3178 using the required name-value pair arguments `DiscountCurve` and `Model`. For example, `HestonPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',HWModel)` creates a `Heston` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value `"Analytic"` or a character vector with the value `'Analytic'`.

Data Types: `char` | `string`

`Heston` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `HestonPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',HWModel)`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a previously created `ratecurve` object.

Note The software uses the `Basis` value of the specified `ratecurve` object to calculate both the discounting and accrual for the `VarianceSwap` instrument object.

Data Types: `object`

`Model` — Model object
`Heston` model object

Model object, specified as the comma-separated pair consisting of `'Model'` and the name of the previously created `Heston` model object using `finmodel`.

Data Types: `object`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`Heston` model object

Model, returned as a `Heston` model object.

Data Types: `object`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Heston Pricer and Heston Model to Price Variance Swap Instrument

This example shows the workflow to price a `VarianceSwap` instrument when you use a `Heston` model and a `Heston` pricing method.

Create `VarianceSwap` Instrument Object

Use `fininstrument` to create a `VarianceSwap` instrument object.

```
VarianceSwapInst = fininstrument("VarianceSwap",'Maturity',datetime(2020,9,15),'Notional',100,'S
```

```
VarianceSwapInst =
  VarianceSwap with properties:

           Notional: 100
    RealizedVariance: 0.0200
             Strike: 0.1000
          StartDate: 15-Jun-2020
           Maturity: 15-Sep-2020
               Name: "variance_swap_instrument"
```

Create `Heston` Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.06,'ThetaV',0.1,'Kappa',0.9,'SigmaV',0.7,'RhoSV',-.3)
```

```
HestonModel =
  Heston with properties:

       V0: 0.0600
    ThetaV: 0.1000
     Kappa: 0.9000
    SigmaV: 0.7000
     RhoSV: -0.3000
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020, 1, 1);
ZeroTimes = calmonths(3);
ZeroRates = 0.05;
ZeroDates = Settle + ZeroTimes;
Basis = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Basis',Basis)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Apr-2020
               Rates: 0.0500
              Settle: 01-Jan-2020
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Heston` Pricer Object

Use `finpricer` to create a `Heston` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Analytic",'DiscountCurve',ZeroCurve,'Model',HestonModel)

outPricer =
  Heston with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.Heston]
```

Price `VarianceSwap` Instrument

Use `price` to compute the price and fair variance for the `VarianceSwap` instrument.

```
[Price, outPR] = price(outPricer,VarianceSwapInst,["all"])

Price = 6.0054

outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: []
```

```
outPR.Results

ans=1×2 table
    Price     FairVariance
    ______    ____________

    6.0054      0.07039
```

Algorithms

Variance swaps can be priced with the calibrated Heston model by using the following closed-form expression for the fair variance:

$$K_{\text{var}} = \frac{1 - e^{-kT}}{kT}(n_0 - \theta) + \theta$$

Here:

- ν_0 is the initial variance of the underlying asset at $t = 0$ $\nu_0 > 0$.
- θ is the long-term variance level $\theta > 0$.
- k is the mean reversion speed for the variance ($k > 0$).

Once the fair variance is computed, the actual price paid in the market at time t for the variance swap with a start date at time 0 is computed as follows:

$$VarianceSwap(t) = Notional \times Disc(t,T) \times \left[\frac{t}{T}RealizedVariance(0,t) + \frac{T-t}{T}FairVariance(t,T) - StrikeVariance\right]$$

Here:

- t is the time from the start date of the variance swap to the settle date.
- T is the time from the start date to the maturity date of the variance swap.
- $Disc(t,T)$ is the discount factor from settle to the maturity date.
- $RealizedVariance(0,t)$ is the realized variance from start date to the settle date, in basis points.
- $FairVariance(t,T)$ is the fair variance for the remaining life of the contract as of the settle date, in basis points.
- *StrikeVariance* is the strike variance predetermined at inception (start date), in basis points.

Version History

Introduced in R2020b

See Also

Functions

`fininstrument` | `finmodel` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

ReplicatingVarianceSwap

Create `ReplicatingVarianceSwap` pricer object for `VarianceSwap` instrument using `ratecurve` object

Description

Create and price a `VarianceSwap` instrument object with a `ratecurve` object and a `ReplicatingVarianceSwap` pricing method using this workflow:

1. Use `fininstrument` to create a `VarianceSwap` instrument object.
2. Use `ratecurve` to specify a curve model for the `VarianceSwap` instrument object.
3. Use `finpricer` to specify a `ReplicatingVarianceSwap` pricer object for the `VarianceSwap` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `VarianceSwap` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
ReplicatingVarianceSwapPricerObj = finpricer(PricerType,'
DiscountCurve',ratecurve_obj,'VolatilitySmile',volatilitysmile_value,'
SpotPrce',spotprice_value)
ReplicatingVarianceSwapPricerObj = finpricer( ___ ,Name,Value)
```

Description

`ReplicatingVarianceSwapPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'VolatilitySmile',volatilitysmile_value,'SpotPrce',spotprice_value)` creates an `ReplicatingVarianceSwap` pricer object by specifying `PricerType` and sets properties on page 11-3184 using the required name-value pair arguments `DiscountCurve`, `VolatilitySmile`, and `SpotPrice`.

`ReplicatingVarianceSwapPricerObj = finpricer( ___ ,Name,Value)` sets optional properties on page 11-3184 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `ReplicatingVarianceSwapPricerObj = finpricer("ReplicatingVarianceSwap",'DiscountCurve',ratecurve_obj,'VolatilitySmile',smiletable,'SpotPrice',1000,'CallPutBoundary',"forwardprice",'InterpMethod',"cubic")` creates a `ReplicatingVarianceSwap` pricer object. You can specify multiple name-value pair arguments.

Input Arguments

`PricerType` — Pricer type

string with value `"ReplicatingVarianceSwap"` | character vector with value `'ReplicatingVarianceSwap'`

Pricer type, specified as a string with the value `"ReplicatingVarianceSwap"` or a character vector with the value `'ReplicatingVarianceSwap'`.

Data Types: `char` | `string`

`ReplicatingVarianceSwap` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ReplicatingVarianceSwapPricerObj = finpricer("ReplicatingVarianceSwap",'DiscountCurve',ratecurve_obj,'VolatilitySmile',smiletable,'SpotPrice',1000,'CallPutBoundary',"forwardprice",'InterpMethod',"cubic")`

Required `ReplicatingVarianceSwap` Name-Value Pair Arguments

`DiscountCurve` — `ratecurve` object for discounting cash flows

`ratecurve` object

This property is read-only.

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a `ratecurve` object.

Note

- Specify a flat `ratecurve` object for `DiscountCurve`. If you use a nonflat `ratecurve` object, the software uses the rate in the `ratecurve` object at `Maturity` and assumes that the value is constant for the life of the equity option.
- The software uses the `Basis` value of the specified `ratecurve` object to calculate both the discounting and accrual for the `VarianceSwap` instrument object.

Data Types: `object`

`VolatilitySmile` — Volatility smile table

table | matrix

Volatility smile table, specified as the comma-separated pair consisting of `'VolatilitySmile'` and a table with the columns `"Strike"` and `"Volatility"` or a `NumVols`-by-2 matrix where the first column is the strikes and the second column is the volatilities in decimals.

Data Types: `table` | `double`

SpotPrice — Spot price of underlying asset
nonnegative numeric

Spot price of the underlying asset, specified as the comma-separated pair consisting of `'SpotPrice'` and a scalar nonnegative numeric.

Note `SpotPrice` must be covered by the range of strikes in `VolatilitySmile`.

Data Types: `double`

Optional ReplicatingVarianceSwap Name-Value Pair Arguments

`CallPutBoundary` — Boundary strike for call and put options
`"spotprice"` (default) | scalar numeric | character vector with value `'spotprice'` or `'forwardprice'` | string with value `"spotprice"` or `"forwardprice"`

Boundary strike for call and put options, specified as the comma-separated pair consisting of `'CallPutBoundary'` and a scalar numeric or one of the following character vectors or strings:

- `"spotprice"` — The call and put option boundary strike is the spot price.
- `"forwardprice"` — The call and put option boundary strike is the forward price.

Note `CallPutBoundary` must be covered by the range of strikes in `VolatilitySmile`.

Data Types: `double` | `char` | `string`

`InterpMethod` — Interpolation method for `SmileTable`
`"linear"` (default) | string with value `"linear"`, `"cubic"`, `"next"`, `"previous"`, `"pchip"`, `"v5cubic"`, `"makima"`, or `"spline"` | character vector with value `'linear'`, `'cubic'`, `'next'`, `'previous'`, `'pchip'`, `'v5cubic'`, `'makima'`, or `'spline'`

Interpolation method for `SmileTable`, specified as the comma-separated pair consisting of `'InterpMethod'` and a scalar string or character vector using a supported value. For more information on interpolation methods, see `interp1`.

Data Types: `char` | `string`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as the `ratecurve` object.

Note The software uses the `Basis` value of the specified `ratecurve` object to calculate both the discounting and accrual for the `VarianceSwap` instrument object.

Data Types: `object`

`VolatilitySmile` — Volatility smile table
table | matrix

Volatility smile table, returned as a table with the columns `"Strike"` and `"Volatility"` or a `NumVols`-by-2 matrix where the first column is the strikes and the second column is the volatilities in decimals.

Data Types: `table` | `double`

`SpotPrice` — Strike price of underlying asset
nonnegative numeric

Strike price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`CallPutBoundary` — Boundary strike for call and put options
`"spotprice"` (default) | scalar numeric | string with value `"spotprice"` or `"forwardprice"`

Boundary strike for call and put options, returned as a numeric or as a string with the value `"spotprice"` or `"forwardprice"`.

Data Types: `double` | `char` | `string`

`InterpMethod` — Interpolation method
`"linear"` (default) | string with value `"linear"`, `"cubic"`, `"next"`, `"previous"`, `"pchip"`, `"v5cubic"`, `"makima"`, or `"spline"`

Interpolation method, returned as a scalar string.

Data Types: `string`

Object Functions

price Compute price for equity instrument with ReplicatingVarianceSwap pricer

Examples

Use Replicating Variance Swap Pricer and `ratecurve` to Price Variance Swap Instrument

This example shows the workflow to price a `VarianceSwap` instrument when you use a `ratecurve` and a `ReplicatingVarianceSwap` pricing method.

Create `VarianceSwap` Instrument Object

Use `fininstrument` to create a `VarianceSwap` instrument object.

```
VarianceSwapInst = fininstrument("VarianceSwap",'Maturity',datetime(2021,5,1),'Notional',150,'Sta
```

```
VarianceSwapInst =
  VarianceSwap with properties:

            Notional: 150
    RealizedVariance: 0.0500
              Strike: 0.1000
           StartDate: 01-May-2020
```

```
            Maturity: 01-May-2021
                Name: "variance_swap_instrument"
```

Create ratecurve Object

Create a flat ratecurve object using ratecurve.

```
Settle = datetime(2020, 9, 15);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Basis = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,'Basis',Basis)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2020
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create ReplicatingVarianceSwap Pricer Object

Use finpricer to create a ReplicatingVarianceSwap pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
Strike = (50:5:135)';
Volatility = [.49;.45;.42;.38;.34;.31;.28;.25;.23;.21;.2;.21;.21;.22;.23;.24;.25;.26];
VolatilitySmile = table(Strike, Volatility);
SpotPrice = 100;
CallPutBoundary = 100;

outPricer =  finpricer("ReplicatingVarianceSwap",'DiscountCurve', ZeroCurve, 'VolatilitySmile', V
'SpotPrice', SpotPrice, 'CallPutBoundary', CallPutBoundary)
```

```
outPricer =
  ReplicatingVarianceSwap with properties:

     DiscountCurve: [1x1 ratecurve]
      InterpMethod: "linear"
   VolatilitySmile: [18x2 table]
         SpotPrice: 100
   CallPutBoundary: 100
```

Price VarianceSwap Instrument

Use price to compute the price and fair variance for the VarianceSwap instrument.

```
[Price, outPR] = price(outPricer,VarianceSwapInst,["all"])
```

```
Price = 8.1997

outPR =
  priceresult with properties:

       Results: [1x2 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×2 table
    Price     FairVariance
    ______    ____________

    8.1997      0.21701
```

```
outPR.PricerData.ReplicatingPortfolio
```

```
ans=19×6 table
    CallPut    Strike    Volatility      Weight       Value     Contribution
    _______    ______    __________    __________    _______    ____________

    "put"        50         0.49        0.0064038    0.39164      0.002508
    "put"        55         0.45        0.0052877    0.49353     0.0026097
    "put"        60         0.42        0.0044402    0.67329     0.0029895
    "put"        65         0.38        0.0037814    0.80343     0.0030381
    "put"        70         0.34        0.0032592     0.9419     0.0030698
    "put"        75         0.31        0.0028382      1.223     0.0034711
    "put"        80         0.28        0.0024938       1.58     0.0039403
    "put"        85         0.25        0.0022086     2.0456     0.0045177
    "put"        90         0.23        0.0019696     2.9221     0.0057554
    "put"        95         0.21        0.0017675     4.1406     0.0073183
    "put"       100          0.2       0.00082405     6.1408     0.0050603
    "call"      100          0.2       0.00077087     6.4715     0.0049887
    "call"      105         0.21        0.0014465     4.7094     0.0068119
    "call"      110         0.21        0.0013178     3.1644     0.0041701
    "call"      115         0.22        0.0012056      2.307     0.0027814
    "call"      120         0.23        0.0011072     1.7127     0.0018962
      ⋮
```

More About

Replicating Variance Swap

A replicating variance swap uses a portfolio of options.

The variance swap replication is accomplished using a portfolio of options with different strikes.

Algorithms

The fair value of the future variance *Kvar* is approximated in terms of the following portfolio of options Π_{CP}:

$$K_{\text{var}} = \frac{2}{T}\left\{rT - \left(\frac{S_0}{S_*}e^{rT} - 1\right) - \log\frac{S_*}{S_0} + e^{rT}\Pi_{CP}\right\}$$

$$\Pi_{CP} = \sum_i w(K_{ip})P(S, K_{ip}) + \sum_i w(K_{ic})C(S, K_{ic})$$

Here:

- Call option strikes — The call option strike are $K_0 < K_{1c} < K_{2c} < K_{3c} \ldots < K_{nc}$.
- Put option strikes — The put option strikes are $K_{mp} < \ldots < K_{3p} < K_{2p} < K_{1p} < K_0 = S_*$.
- K_{var} — is the fair value of future variance
- Π_{CP} — is the portfolio of call and put options
- S_0 — is the current asset price
- S_* — is the boundary between the call and put option strikes (for example, the spot price S_0 or forward price S_0e^{rT})
- $P(K)$ — is the current put option price with strike K
- $C(K)$ — is the current call option price with strike K

If the options portfolio Π_{CP} has an infinite number of options with continuously varying strikes, it has the following payoff function at maturity:

$$f(S_T) = \frac{2}{T}\left[\frac{S_T - S_*}{S_*} - \log\frac{S_T}{S_*}\right]$$

Since it is not possible to construct such a portfolio with an infinite number of options and continuously varying strikes, the appropriate weights $w(K_{ip})$ and $w(K_{ic})$ for a portfolio with a finite number of options and discretely varying strikes can be computed by approximating the continuous payoff function $f(S_T)$ in a piecewise linear fashion. Starting with the strike at K_0, the first call option weight can be computed as the slope of the first piecewise linear function:

$$w_c(K_0) = \frac{f(K_{1c}) - f(K_0)}{K_{1c} - K_0}$$

The next call option weight with the strike K_{1c} is computed as the slope of the next piece-wise linear function minus the previous weight:

$$w_c(K_{1c}) = \frac{f(K_{2c}) - f(K_{1c})}{K_{2c} - K_{1c}} - w_c(K_0)$$

This procedure is continued for the remaining call option strikes:

$$w_c(K_{n,c}) = \frac{f(K_{n+1,c}) - f(K_{nc})}{K_{n+1,c} - K_{n,c}} - \sum_{i=0}^{n-1} w_c(K_{i,c})$$

To compute the put option weights, a similar procedure can be used in the opposite direction (starting from K_0):

$$w_p(K_{m,p}) = \frac{f(K_{m+1,p}) - f(K_{mp})}{K_{m,p} - K_{m+1,p}} - \sum_{i=0}^{m-1} w_p(K_{i,p})$$

Once the fair variance is computed, the actual price paid in the market at time t for the variance swap with a StartDate at time 0 is computed as follows:

$$VarianceSwap(t) = Notional \times Disc(t,T) \times \left[\frac{t}{T}\text{Re}alizedVariance(0,t) + \frac{T-t}{T}FairVariance(t,T) - StrikeVariance\right]$$

Here:

- t is the time from the start date of the variance swap to the settle date.
- T is the time from the start date to the maturity date of the variance swap.
- $Disc(t,T)$ is the discount factor from settle to the maturity date.
- $RealizedVariance(0,t)$ is the realized variance from start date to the settle date, in basis points.
- $FairVariance(t,T)$ is the fair variance for the remaining life of the contract as of the settle date, in basis points.
- *StrikeVariance* is the strike variance predetermined at inception (start date), in basis points.

Version History

Introduced in R2020b

References

[1] Demeterfi, K., Derman, E., Kamal, M., and J. Zou. “More Than You Ever Wanted To Know About Volatility Swaps.” *Quantitative Strategies Research Notes.* Goldman Sachs, 1999.

See Also

Functions

`fininstrument` | `finmodel`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22

“Choose Instruments, Models, and Pricers” on page 1-53

BjerksundStensland

Create `BjerksundStensland` pricer object for `Vanilla` or `Spread` instrument using `BlackScholes` model

Description

Create and price a `Vanilla` or `Spread` instrument object with a `BlackScholes` model and a `BjerksundStensland` pricing method using this workflow:

1. Use `fininstrument` to create a `Vanilla` or `Spread` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Vanilla` or `Spread` instrument object.
3. Use `finpricer` to specify a `BjerksundStensland` pricer object for the `Vanilla` instrument (American exercise) or `Spread` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Vanilla` or `Spread` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BjerksundStenslandPricerObj = finpricer(PricerType,'Model',model,'
DiscountCurve',ratecurve_obj,'SpotPrice',spotprice_value)
BjerksundStenslandPricerObj = finpricer( ___ ,Name,Value)
```

Description

`BjerksundStenslandPricerObj = finpricer(PricerType,'Model',model,'DiscountCurve',ratecurve_obj,'SpotPrice',spotprice_value)` creates a `BjerksundStensland` pricer object by specifying `PricerType` and sets the properties on page 11-3192 for the required name-value pair arguments `Model`, `DividendType`, and `SpotPrice`.

`BjerksundStenslandPricerObj = finpricer( ___ ,Name,Value)` to set optional properties on page 11-3192 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `BjerksundStenslandPricerObj = finpricer("Analytic",'Model',BSModel,'DiscountCurve',ratecurve_obj,'SpotPrice',[100;105],'DividendValue',[2.5,2.8],'PricingMethod',"BjerksundStensland")` creates a `BjerksundStensland` pricer object.

Input Arguments

`PricerType` — Pricer type
string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: `char` | `string`

BjerksundStensland Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BjerksundStenslandPricerObj = finpricer("Analytic",'Model',BSModel,'DiscountCurve',ratecurve_obj,'SpotPrice',[100;105],'DividendValue',[2.5,2.8],'PricingMethod',"BjerksundStensland")`

Required BjerksundStensland Name-Value Pair Arguments

`Model` — Model
object

Model, specified as the comma-separated pair consisting of `'Model'` and the name of the previously created `BlackScholes` model object using `finmodel`.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of the previously created `ratecurve` object.

Note Specify a flat `ratecurve` object for `DiscountCurve`. If you use a nonflat `ratecurve` object, the software uses the rate in the `ratecurve` object at `Maturity` and assumes that the value is constant for the life of the equity option.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
scalar nonnegative numeric | vector of nonnegative numerics

Current price of the underlying asset, specified as the comma-separated pair consisting of `'SpotPrice'` and a scalar or vector of nonnegative numeric values. Use a vector for `SpotPrice` when pricing a `Spread` instrument.

Data Types: `double`

Optional BjerksundStensland Name-Value Pair Arguments

`DividendValue` — Dividend yield
`[0,0]` (default) | scalar nonnegative numeric | vector of nonnegative numerics

Dividend yield, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar or vector of nonnegative numeric values. Use a 1-by-2 vector of nonnegative values for `DividendValue` when pricing a `Spread` instrument.

Data Types: double

DividendType — Dividend type
"continuous" (default) | string with value "continuous" | character vector with value 'continuous'

Dividend type, specified as the comma-separated pair consisting of 'DividendType' and string or character vector.

Data Types: char | string

PricingMethod — Analytic pricing method
default pricer associated with BlackScholes model (default) | string with value "BjerksundStensland" | character vector with value 'BjerksundStensland'

Analytic pricing method, specified as the comma-separated pair consisting of 'PricingMethod' and a character vector or string.

Note The default pricing method for a BlackScholes model is a BlackScholes pricer.

Data Types: string | char

Properties

Model — Model
object

Model, returned as a model object.

Data Types: object

DiscountCurve — ratecurve object for discounting cash flows
ratecurve object

This property is read-only.

ratecurve object for discounting cash flows, returned as a ratecurve object.

Data Types: object

SpotPrice — Current price of underlying asset
nonnegative numeric | vector of nonnegative numerics

Current price of the underlying asset, returned as a scalar or vector of nonnegative numeric values.

Data Types: double

DividendValue — Dividend yield
scalar nonnegative numeric | vector of nonnegative numerics

Dividend yield, returned as a scalar or vector of nonnegative numeric.

Data Types: double

DividendType — Dividend type
"continuous" (default) | string with value "continuous"

This property is read-only.

Dividend type, returned as a string.

Data Types: `string`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"BjerksundStensland"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Bjerksund-Stensland Pricer and Black-Scholes Model to Price Spread Instrument

This example shows the workflow to price a European exercise `Spread` instrument when you use a `BlackScholes` model and a `BjerksundStensland` pricing method.

Create Spread Instrument Object

Use `fininstrument` to create a `Spread` instrument object.

```
SpreadOpt = fininstrument("Spread",'Strike',5,'ExerciseDate',datetime(2021,9,15),'OptionType',"pu
```

```
SpreadOpt =
  Spread with properties:

       OptionType: "put"
           Strike: 5
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2021
             Name: "spread_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',[0.2,0.1])
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: [0.2000 0.1000]
    Correlation: [2x2 double]
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)

myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `BjerksundStensland` Pricer Object

Use `finpricer` to create a `BjerksundStensland` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',[100

outPricer =
  BjerksundStensland with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: [100 105]
    DividendValue: [0.0900 0.1700]
     DividendType: "continuous"
```

Price Spread Instrument

Use `price` to compute the price and sensitivities for the `Spread` instrument.

```
[Price, outPR] = price(outPricer,SpreadOpt,["all"])

Price = 7.0596

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []


outPR.Results

ans=1×7 table
    Price            Delta                    Gamma                   Lambda                Vega
    _____    ____________________    ______________________    ________________    ___________
```

```
    7.0596    -0.23249     0.27057    0.0069887    0.0055319    -3.2932     3.8327    41.938
```

Use Bjerksund-Stensland Pricer and Black-Scholes Model to Price an American Exercise Vanilla Instrument

This example shows the workflow to price an American exercise `Vanilla` instrument when you use a `BlackScholes` model and a `BjerksundStensland` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create an American exercise `Vanilla` instrument object.

```
VanillaObj = fininstrument("Vanilla",'Strike',120,'ExerciseDate',datetime(2019,1,30),'OptionType
```

```
VanillaObj =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "american"
     ExerciseDate: 30-Jan-2019
           Strike: 120
             Name: "vanilla_instrument"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',.2)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2000
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 12
             Dates: 15-Sep-2023
             Rates: 0.0350
            Settle: 15-Sep-2018
      InterpMethod: "linear"
```

```
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `BjerksundStensland` Pricer Object

Use `finpricer` to create a `BjerksundStensland` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',120,
```

```
outPricer =
  BjerksundStensland with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 120
    DividendValue: 0.0500
     DividendType: "continuous"
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaObj,["all"])
```

```
Price = 6.1080
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price     Delta       Gamma       Lambda      Vega      Theta       Rho
    _____    ________    ________    _______    ______    _______    _______

    6.108    -0.48471    0.026611    -9.5227    28.781    -8.3418    -24.115
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22

“Choose Instruments, Models, and Pricers” on page 1-53

Black

Create Black pricer object for Cap, Floor, or Swaption instrument using Black model

Description

Create and price a Cap, Floor, or Swaption instrument object with a Black model and a Black pricing method using this workflow:

1. Use fininstrument to create a Cap, Floor, or Swaption instrument object.
2. Use finmodel to specify a Black model for the Cap, Floor, or Swaption instrument object.
3. Use finpricer to specify a Black pricer object for the Cap, Floor, or Swaption instrument object.

Note If you do not specify ProjectionCurve when you create a Cap, Floor, or Swaption instrument with the Black pricer, the ProjectionCurve value defaults to the DiscountCurve value.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a Cap, Floor, or Swaption instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BlackPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model)
```

Description

`BlackPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model)` creates a Black pricer object by specifying PricerType and the required name-value pair arguments for DiscountCurve and Model to set properties on page 11-3199 using name-value pairs. For example, `BlackPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BlackModel)` creates a Black pricer object.

Input Arguments

PricerType — Pricer type

string with value "Analytic" | character vector with value 'Analytic'

Pricer type, specified as a string with the value of "Analytic" or a character vector with the value of 'Analytic'.

Data Types: `char` | `string`

`Black` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BlackPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BlackModel)`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of the previously created `ratecurve` object.

Data Types: `object`

`Model` — Model
`Black` model object

Model, specified as the comma-separated pair consisting of `'Model'` and the name of a previously created `Black` model object using `finmodel`.

Data Types: `object`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as the `ratecurve` object

Data Types: `object`

`Model` — Model
`Black` model object

Model, returned as a `Black` model object.

Data Types: `object`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Black Pricer and Black Model to Price Cap Instrument

This example shows the workflow to price a `Cap` instrument when you use a `Black` model and a `Black` pricing method.

Create Cap Instrument Object

Use `fininstrument` to create a `Cap` instrument object.

```
CapOpt = fininstrument("Cap",'Strike',.02,'Maturity',datetime(2021,12,30),'Reset',4,'Principal',1
```

```
CapOpt =
  Cap with properties:

                     Strike: 0.0200
                   Maturity: 30-Dec-2021
                ResetOffset: 0
                      Reset: 4
                      Basis: 12
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "cap_option"
```

Create Black Model Object

Use `finmodel` to create a `Black` model object.

```
BlackModel = finmodel("Black",'Volatility',0.09,'Shift',0.002)
```

```
BlackModel =
  Black with properties:

    Volatility: 0.0900
         Shift: 0.0020
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2020,9,14);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: [10x1 datetime]
             Rates: [10x1 double]
            Settle: 14-Sep-2020
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
```

```
     LongExtrapMethod: "previous"
```

Create Black Pricer Object

Use `finpricer` to create a `Black` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackModel,'DiscountCurve',myRC)
```

```
outPricer =
  Black with properties:

            Model: [1x1 finmodel.Black]
    DiscountCurve: [1x1 ratecurve]
```

Price Cap Instrument

Use `price` to compute the price for the `Cap` instrument.

```
Price = price(outPricer,CapOpt)
```

```
Price = 4.6412e-29
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Work with Negative Interest Rates Using Objects” on page 2-22

BlackScholes

Create `BlackScholes` pricer object for `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument using `BlackScholes` model

Description

Create and price a `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument object with a `BlackScholes` model and a `BlackScholes` pricing method using this workflow:

1. Use `fininstrument` to create a `Vanilla`, `Barrier`, `DoubleTouch`, `Binary` or , `Touch` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument object.
3. Use `finpricer` to specify a `BlackScholes` pricer object for the `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Vanilla`, `Lookback`, `Barrier`, `Asian`, `Spread`, `Touch`, `DoubleTouch`, or `Binary` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
BlackScholesPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model,'SpotPrice',spotprice_value)
BlackScholesPricerObj = finpricer(___,Name,Value)
```

Description

`BlackScholesPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model,'SpotPrice',spotprice_value)` creates a `BlackScholes` pricer object by specifying `PricerType` and sets properties on page 11-3204 using the required name-value pair arguments `DiscountCurve`, `Model`, and `SpotPrice`.

`BlackScholesPricerObj = finpricer(___,Name,Value)` sets optional properties on page 11-3204 using additional name-value pair arguments in addition to the required arguments in the previous syntax. For example, `BlackScholesPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',100)` creates a `BlackScholes` pricer object. You can specify multiple name-value pair arguments.

Input Arguments

`PricerType` — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value `"Analytic"` or a character vector with the value `'Analytic'`.

Data Types: `char` | `string`

`BlackScholes` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BlackScholesPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',100)`

Required BlackScholes Name-Value Pair Arguments

`DiscountCurve` — `ratecurve` object for discounting cash flows

`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a previously created `ratecurve` object.

Note Specify a flat `ratecurve` object for `DiscountCurve`. If you use a nonflat `ratecurve` object, the software uses the rate in the `ratecurve` object at `Maturity` and assumes that the value is constant for the life of the equity option.

Data Types: `object`

`Model` — Model object

`BlackScholes` model object

Model, specified as the comma-separated pair consisting of `'Model'` and the name of a previously created `BlackScholes` model object using `finmodel`.

Data Types: `object`

`SpotPrice` — Current price of underlying asset

nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of `'SpotPrice'` and a scalar nonnegative numeric.

Data Types: `double`

Optional BlackScholes Name-Value Pair Arguments

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"continuous"` | character vector with value `'continuous'`

Stock dividend type, specified as the comma-separated pair consisting of `'DividendType'` and a character vector or string.

Note When you price currencies using a `Vanilla` instrument, `DividendType` must be `"continuous"` and `DividendValue` is the annualized risk-free interest rate in the foreign country.

Data Types: `char` | `string`

`DividendValue` — Dividend amount or dividend schedule for underlying stock
`0` (default) | scalar numeric | timetable

Dividend amount or dividend schedule for the underlying stock, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar numeric for a dividend amount or a timetable for a dividend schedule.

Note When you price currencies using a `Vanilla` instrument, the `DividendType` must be `"continuous"` and `DividendValue` is the annualized risk-free interest rate in the foreign country.

Data Types: `double` | `timetable`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"continuous"`

This property is read-only.

Stock dividend type, returned as a string.

Data Types: `string`

`DividendValue` — Dividend amount or dividend schedule for underlying stock

`0` (default) | scalar nonnegative numeric | timetable

Dividend amount or dividend schedule for the underlying stock, returned as a scalar numeric for a dividend amount or a timetable for a dividend schedule.

Data Types: `double` | `timetable`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Price Vanilla Instrument Using Black-Scholes Model and Black-Scholes Pricer

This example shows the workflow to price a `Vanilla` instrument when you use a `BlackScholes` model and a `BlackScholes` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2018,5,1),'Strike',29,'OptionType','
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 01-May-2018
           Strike: 29
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2500
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2019,1,1);
```

```
Rate = 0.05;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2019
               Rates: 0.0500
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `BlackScholes` Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',30,'D
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 30
    DividendValue: 0.0450
     DividendType: "continuous"
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 1.2046
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma      Lambda      Vega       Rho       Theta
    ______    _________    ________    _______    ______    _______    _______

    1.2046    -0.36943    0.086269    -9.3396    6.4702    -4.0959    -2.3107
```

Use Black-Scholes Pricer and Black-Scholes Model to Price Vanilla Instrument for Foreign Exchange

This example shows the workflow to price a `Vanilla` instrument for foreign exchange (FX) when you use a `BlackScholes` model and a `BlackScholes` pricing method. Assume that the current exchange rate is $0.52 and has a volatility of 12% per year. The annualized continuously compounded foreign risk-free rate is 8% per year.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2022,9,15),'Strike',.50,'OptionType
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
           Strike: 0.5000
             Name: "vanilla_fx_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
Sigma = .12;
BlackScholesModel = finmodel("BlackScholes",'Volatility',Sigma)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.1200
    Correlation: 1
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
```

```
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackScholes Pricer Object

Use `finpricer` to create a `BlackScholes` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument. When you price currencies using a `Vanilla` instrument, `DividendType` must be `'continuous'` and `DividendValue` is the annualized risk-free interest rate in the foreign country.

```
ForeignRate = 0.08;
outPricer = finpricer("analytic",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',.52,
```

```
outPricer =
  BlackScholes with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 0.5200
    DividendValue: 0.0800
     DividendType: "continuous"
```

Price Vanilla Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` FX instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 0.1123
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
     Price      Delta      Gamma     Lambda      Vega        Rho        Theta
    _______    ________    ______    _______    _______    _______    _________

    0.11229    -0.59114    1.5562    -3.7706    0.20212    -1.6799    -0.023676
```

Version History

Introduced in R2020a

See Also

Functions
`fininstrument` | `finmodel` | `timetable` | `ratecurve`

Topics
“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

ConzeViswanathan

Create `ConzeViswanathan` pricer object for `Lookback` instrument using `BlackScholes` model

Description

Create and price a `Lookback` instrument object with a `BlackScholes` model and a `ConzeViswanathan` pricing method using this workflow:

1. Use `fininstrument` to create a `Lookback` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Lookback` instrument object.
3. Use `finpricer` to specify a `ConzeViswanathan` pricer object for the `Lookback` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Lookback` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
ConzeViswanathanPricerObj = finpricer(PricerType,'
DiscountCurve',ratecurve_obj,'Model',model,'SpotPrice',spotprice_value)
ConzeViswanathanPricerObj = finpricer(___,Name,Value)
```

Description

`ConzeViswanathanPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model,'SpotPrice',spotprice_value)` creates a `ConzeViswanathan` pricer object by specifying `PricerType` and sets the properties on page 11-3212 for the required name-value pair argument `Model`, `DiscountCurve`, and `SpotPrice`.

`ConzeViswanathanPricerObj = finpricer(___,Name,Value)` to set optional properties on page 11-3212 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `ConzeViswanathanPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',100,'PricingMethod',"ConzeViswanathan")` creates a `ConzeViswanathan` pricer object.

Input Arguments

PricerType — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: `char` | `string`

ConzeViswanathan Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Specify required and optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `ConzeViswanathanPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',100,'PricingMethod',"ConzeViswanathan")`

Required ConzeViswanathan Name-Value Pair Arguments

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of the previously created `ratecurve` object.

Note Specify a flat `ratecurve` object for `DiscountCurve`. If you use a nonflat `ratecurve` object, the software uses the rate in the `ratecurve` object at `Maturity` and assumes that the value is constant for the life of the equity option.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, specified as the comma-separated pair consisting of `'Model'` and the name of a previously created `BlackScholes` model object using `finmodel`.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of `'SpotPrice'` and a scalar nonnegative numeric.

Data Types: `double`

Optional ConzeViswanathan Name-Value Pair Arguments

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"cash"` or `"continuous"` | character vector with value `'cash'` or `'continuous'`

Stock dividend type, specified as the comma-separated pair consisting of `'DividendType'` and character vector or string. `DividendType` must be `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `char` | `string`

`DividendValue` — Dividend amount or dividend schedule for underlying stock
`0` (default) | scalar numeric | timetable

Dividend amount for the underlying stock, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar numeric for a dividend amount or a `timetable` for a dividend schedule.

Note Specify a scalar if `DividendType` is `"continuous"` and a timetable if `DividendType` is `"cash"`.

Data Types: `double` | `timetable`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"ConzeViswanathan"` | character vector with value `'ConzeViswanathan'`

Analytic pricing method, specified as the comma-separated pair consisting of `'PricingMethod'` and a string or character vector.

Note The default pricing method for a `BlackScholes` model is a `BlackScholes` pricer.

Data Types: `double`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as the `ratecurve` object.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"cash"` or `"continuous"`

This property is read-only.

Stock dividend type, returned as a string. `DividendType` is `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `string`

`DividendValue` — Dividend amount or dividend schedule for underlying stock
`0` (default) | scalar nonnegative numeric | timetable

Dividend amounts or dividend schedule for underlying stock, returned as a scalar numeric for a dividend yield or a timetable for a dividend schedule.

Data Types: `double` | `timetable`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"ConzeViswanathan"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Conze-Viswanathan Pricer and Black-Scholes Model to Price Lookback Instrument

This example shows the workflow to price a fixed strike `Lookback` instrument when you use a `BlackScholes` model and a `ConzeViswanathan` pricing method.

Create Lookback Instrument Object

Use `fininstrument` to create a fixed strike `Lookback` instrument object.

```
LookbackOpt = fininstrument("Lookback",'Strike',90,'ExerciseDate',datetime(2021,9,15),'OptionType
```

```
LookbackOpt =
  Lookback with properties:

       OptionType: "put"
           Strike: 90
      AssetMinMax: NaN
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2021
             Name: "lookback_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',.358)

BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3580
    Correlation: 1
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)

myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `ConzeViswanathan` Pricer Object

Use `finpricer` to create a `ConzeViswanathan` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',95,'D

outPricer =
  ConzeViswanathan with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 95
    DividendValue: 0.0250
     DividendType: "continuous"
```

Price `Lookback` Instrument

Use `price` to compute the price and sensitivities for the `Lookback` instrument.

```
[Price, outPR] = price(outPricer,LookbackOpt,["all"])

Price = 29.6209

outPR =
  priceresult with properties:
```

```
       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta        Gamma       Lambda      Vega      Theta       Rho
    _____    _________    _________    _______    ______    _______    _______

    29.621    -0.49834    0.0085048    -1.5983    78.578    -3.4045    -163.55
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `timetable` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Credit

Create `Credit` pricer object for `CDS` instrument using `defprobcurve`

Description

Create and price a `CDS` instrument object with a `defprobcurve` and a `Credit` pricing method using this workflow:

1. Create a default probability curve object using `defprobcurve`.
2. Use `finpricer` to specify a `Credit` pricer object for the `CDS` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `CDS` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
CreditPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
DefaultProbabilityCurve',defprobcurve_object)
```

Description

`CreditPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'DefaultProbabilityCurve',defprobcurve_object)` creates a `Credit` pricer object by specifying `PricerType` and the required name-value pair arguments `DiscountCurve` and `DefaultProbabilityCurve` to set properties on page 11-3217 using name-value pairs. For example, `CreditPricerObj = finpricer("Credit",'DiscountCurve',ratecurve_obj,'DefaultProbabilityCurve',defprobcurve_obj)` creates a `Credit` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Credit"` | character vector with value `'Credit'`

Pricer type, specified as a string with the value of `"Credit"` or a character vector with the value of `'Credit'`.

Data Types: `char` | `string`

`Credit` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `CreditPricerObj = finpricer("Credit",'DiscountCurve',ratecurve_obj,'DefaultProbabilityCurve',defprobcurve_obj)`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of the previously created `ratecurve` object

Data Types: `object`

`DefaultProbabilityCurve` — Default probability curve
`defprobcurve` object

Default probability curve, specified as the comma-separated pair consisting of `'DefaultProbabilityCurve'` and the name of a previously created `defprobcurve` object.

Data Types: `object`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as the `ratecurve` object

Data Types: `object`

`DefaultProbabilityCurve` — Default probability curve
`defprobcurve` object

Default probability curve, returned as a `defprobcurve` object.

Data Types: `object`

Object Functions

price Compute price for credit derivative instrument with Credit pricer

Examples

Use Credit Pricer and Default Probability Curve to Price CDS Instrument

This example shows the workflow to price a `CDS` instrument when you use a `defprobcurve` model and a `Credit` pricing method.

Create `CDS` Instrument Object

Use `fininstrument` to create a `CDS` instrument object.

```
CDS = fininstrument("cds",'Maturity',datetime(2027,9,20),'ContractSpread', 50,'Name',"CDS_instru
```

```
CDS =
  CDS with properties:
```

```
          ContractSpread: 50
                Maturity: 20-Sep-2027
                  Period: 4
                   Basis: 2
            RecoveryRate: 0.4000
  BusinessDayConvention: "actual"
                Holidays: NaT
      PayAccruedPremium: 1
                Notional: 10000000
                    Name: "CDS_instrument"
```

Create `defprobcurve` Object

Create a `defprobcurve` object using `defprobcurve`.

```
Settle = datetime(2017, 9, 20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                 Settle: 20-Sep-2017
                  Basis: 2
                  Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 20-Sep-2017
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `Credit` Pricer Object

Use `finpricer` to create a `Credit` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
CDSpricer = finpricer("credit",'DiscountCurve',ZeroCurve,'DefaultProbabilityCurve',DefaultProbCu
```

```
CDSpricer =
  Credit with properties:

              DiscountCurve: [1x1 ratecurve]
                   TimeStep: 10
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price CDS Instrument

Use `price` to compute the price for the `CDS` instrument.

```
outPrice = price(CDSpricer, CDS)
```

```
outPrice = 6.9363e+04
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

CDSBlack

Create CDSBlack pricer object for CDSOption instrument using CDSBlack model

Description

Create and price a CDSOption instrument object with a CDSBlack model and a CDSBlack pricing method using this workflow:

1. Use fininstrument to create the CDSOption instrument object. By default, this creates a single-name CDS option. You can create a CDS index option by specifying the optional name-value argument AdjustedForwardSpread.
2. Use finmodel to specify the CDSBlack model for the CDSOption instrument object.
3. Use finpricer to specify the CDSBlack pricer object for the CDSOption instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a CDSOption instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
CDSBlackPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model,'DefaultProbabilityCurve',defaultprobabilitycurve_obj)
```

Description

`CDSBlackPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model,'DefaultProbabilityCurve',defaultprobabilitycurve_obj)` creates a CDSBlack pricer object by specifying PricerType and the required name-value pair arguments for DiscountCurve, Model, and DefaultProbabilityCurve to set properties on page 11-3221 using name-value pairs. For example, `CDSBlackPricerObj = finpricer("Analytic",'Model',CDSBlack,'DiscountCurve',ratecurve_obj,'DefaultProbabilityCurve',defaultprobabilitycurve_obj)` creates a CDSBlack pricer object.

Input Arguments

PricerType — Pricer type

string with value "Analytic" | character vector with value 'Analytic'

Pricer type, specified as a string with the value of "Analytic" or the character vector with a value of 'Analytic'.

Data Types: char | string

CDSBlack Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `CDSBlackPricerObj = finpricer("Analytic",'Model',CDSBlack,'DiscountCurve',ratecurve_obj,'DefaultProbabilityCurve',defaultprobabilitycurve_obj)`

Required CDSBlack Name-Value Pair Arguments

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of the previously created `ratecurve` object.

Data Types: `object`

`Model` — Model
`CDSBlack` model object

Model object, specified as the comma-separated pair consisting of `'Model'` and the name of a previously created `CDSBlack` model object using `finmodel`.

Data Types: `object`

`DefaultProbabilityCurve` — Default probability curve
`defprobcurve` object

Default probability curve, specified as the comma-separated pair consisting of `'DefaultProbabilityCurve'` and a name of a previously created `defprobcurve`.

Data Types: `object`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as the `ratecurve` object.

Data Types: `object`

`Model` — Model
`CDSBlack` model object

Model, returned as a `CDSBlack` model object.

Data Types: `object`

`DefaultProbabilityCurve` — Default probability curve
object

Default probability curve, returned as a `defprobcurve` object.

Data Types: `object`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use CDS Black Pricer and CDS Black Model to Price CDS Option Instrument

This example shows the workflow to price a `CDSOption` instrument when you use a `CDSBlack` model and a `CDSBlack` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2017,9,20);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero", Settle, ZeroDates ,ZeroRates)

ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 20-Sep-2017
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create `defprobcurve` Object

Create a `defprobcurve` object using `defprobcurve`.

```
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;
DefaultProbCurve = defprobcurve(Settle, ProbDates, DefaultProbabilities)

DefaultProbCurve =
  defprobcurve with properties:

                 Settle: 20-Sep-2017
                  Basis: 2
                  Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Create CDS Instrument Object

Use `fininstrument` to create an underlying `CDS` instrument object.

```
ContractSpreadBP = 0; % Contractual spread is determined on ExerciseDate
CDS = fininstrument("CDS",'Maturity',datetime(2027,9,20),'ContractSpread',ContractSpreadBP)
```

```
CDS =
  CDS with properties:

          ContractSpread: 0
                Maturity: 20-Sep-2027
                  Period: 4
                   Basis: 2
            RecoveryRate: 0.4000
    BusinessDayConvention: "actual"
                Holidays: NaT
        PayAccruedPremium: 1
                Notional: 10000000
                    Name: ""
```

Create CDSOption Instrument Object

Use `fininstrument` to create a `CDSOption` instrument object.

```
ExerciseDate = datetime(2017, 12, 20);
Strike = 50;
CDSOption = fininstrument("CDSOption",'Strike',Strike,'ExerciseDate',ExerciseDate,'OptionType',"
```

```
CDSOption =
  CDSOption with properties:

              OptionType: "put"
                  Strike: 50
               Knockout: 0
    AdjustedForwardSpread: NaN
             ExerciseDate: 20-Dec-2017
                      CDS: [1x1 fininstrument.CDS]
                     Name: ""
```

Create CDSBlack Model Object

Use `finmodel` to create a `CDSBlack` model object.

```
SpreadVolatility = 0.3;
CDSOptionModel = finmodel("CDSBlack",'SpreadVolatility',SpreadVolatility)
```

```
CDSOptionModel =
  CDSBlack with properties:

    SpreadVolatility: 0.3000
```

Create CDSBlack Pricer Object

Use `finpricer` to create a `CDSBlack` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
CDSOptionpricer = finpricer("analytic",'Model',CDSOptionModel,'DiscountCurve',ZeroCurve,'DefaultI

CDSOptionpricer =
  CDSBlack with properties:

                      Model: [1x1 finmodel.CDSBlack]
              DiscountCurve: [1x1 ratecurve]
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price `CDSOption` Instrument

Use `price` to compute the price for the `CDSOption` instrument.

```
outPrice = price(CDSOptionpricer,CDSOption)

outPrice = 6.5054
```

Use CDS Black Pricer and CDS Black Model to Price CDS Index Options

This example shows the workflow to use a `CDSOption` instrument to price CDS index options when you use a `CDSBlack` model and a `CDSBlack` pricing method.

Set Up Data for CDS Index

```
% CDS index and option data
Recovery = .4;
Basis = 2;
Period = 4;
CDSMaturity = datetime(2017, 6, 20);
ContractSpread = 100;
IndexSpread = 140;
BusDayConvention = 'follow';
Settle = datetime(2012, 4, 13);
OptionMaturity = datetime(2012, 6, 20);
OptionStrike = 140;
SpreadVolatility = .69;
```

Create `ratecurve` Object for Zero Curve Using `irbootstrap`

Create `ratecurve` object for a zero curve using `irbootstrap`.

```
% Zero curve data
DepRates = [0.004111 0.00563 0.00757 0.01053]';
DepTimes = calmonths([1 2 3 6]');
DepDates = Settle + DepTimes;
nDeposits = length(DepTimes);

SwapRates = [0.01387 0.01035 0.01145 0.01318 0.01508 0.01700 0.01868 ...
    0.02012 0.02132 0.02237 0.02408 0.02564 0.02612 0.02524]';
SwapTimes = calyears([1 2 3 4 5 6 7 8 9 10 12 15 20 30]');
SwapDates = Settle + SwapTimes;
nSwaps = length(SwapTimes);

nInst = nDeposits + nSwaps;

BootInstruments(nInst,1) = fininstrument.FinInstrument;
```

```
for ii=1:length(DepDates)
    BootInstruments(ii) = fininstrument("deposit","Maturity",DepDates(ii),"Rate",DepRates(ii));
end

for ii=1:length(SwapDates)
    BootInstruments(ii+nDeposits) = fininstrument("swap","Maturity",SwapDates(ii),"LegRate",[Swap
end

ZeroCurve = irbootstrap(BootInstruments,Settle)
```

```
ZeroCurve =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [18x1 datetime]
               Rates: [18x1 double]
              Settle: 13-Apr-2012
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Bootstrap Default Probability Curve

Use `defprobstrip` to bootstrap default probability curve assuming a flat index spread.

```
ProbDates = datemnth(OptionMaturity,(0:5*12)');
MarketCDSInstruments = fininstrument("cds", ...
    'ContractSpread', ContractSpread, 'Maturity', CDSMaturity);
DefaultProbCurve = defprobstrip(ZeroCurve, MarketCDSInstruments, IndexSpread, 'ProbDates', ProbDa
```

```
DefaultProbCurve =
  defprobcurve with properties:

                 Settle: 13-Apr-2012
                  Basis: 2
                  Dates: [61x1 datetime]
    DefaultProbabilities: [61x1 double]
```

Compute Spot and Forward RPV01s

Compute the spot and forward RPV01s using `cdsrpv01`.

```
ProbData = [datenum(DefaultProbCurve.Dates) DefaultProbCurve.DefaultProbabilities];

% RPV01(t,T)
RPV01_CDSMaturity = cdsrpv01(ZeroCurve,ProbData,Settle,CDSMaturity)
```

```
RPV01_CDSMaturity = 4.7853
```

```
% RPV01(t,t_E,T)
RPV01_OptionExpiryForward = cdsrpv01(ZeroCurve,ProbData,Settle,CDSMaturity,...
    'StartDate',OptionMaturity)
```

```
RPV01_OptionExpiryForward = 4.5972
```

```
% RPV01(t,t_E) = RPV01(t,T) - RPV01(t,t_E,T)
RPV01_OptionExpiry = RPV01_CDSMaturity - RPV01_OptionExpiryForward

RPV01_OptionExpiry = 0.1882
```

Compute Spot Spreads

Compute the spot spreads using `cdsspread`.

```
% S(t,t_E)
Spread_OptionExpiry = cdsspread(ZeroCurve,ProbData,Settle,OptionMaturity,...
    'Period',Period,'Basis',Basis,'BusDayConvention',BusDayConvention,...
    'PayAccruedPremium',true,'recoveryrate',Recovery)

Spread_OptionExpiry = 139.8995


% S(t,T)
Spread_CDSMaturity = cdsspread(ZeroCurve,ProbData,Settle,CDSMaturity,...
    'Period',Period,'Basis',Basis,'BusDayConvention',BusDayConvention,...
    'PayAccruedPremium',true,'recoveryrate',Recovery)

Spread_CDSMaturity = 139.9999
```

Compute Forward Spread

Compute the forward spread using the spot spreads and RPV01s.

```
% F = S(t,t_E,T)
ForwardSpread = (Spread_CDSMaturity.*RPV01_CDSMaturity - Spread_OptionExpiry.*RPV01_OptionExpiry

ForwardSpread = 140.0040
```

Compute Front-End Protection

Compute the front-end protection (FEP).

```
FEP = 10000*(1-Recovery)*ZeroCurve.discountfactors(OptionMaturity)*DefaultProbCurve.DefaultProbal

FEP = 26.3108
```

Compute Adjusted Forward Spread

Compute the adjusted forward spread to use when creating an `CDSOption` instrument.

```
AdjustedForwardSpread = ForwardSpread + FEP./RPV01_OptionExpiryForward

AdjustedForwardSpread = 145.7273
```

Compute CDS Option Prices with Adjusted Forward Spread

Use `fininstrument` to create a `CDSOption` instrument for a single-name CDS option.

```
CDS = fininstrument("cds",'ContractSpread', ContractSpread, 'Maturity', CDSMaturity)

CDS =
  CDS with properties:

           ContractSpread: 100
```

```
                Maturity: 20-Jun-2017
                  Period: 4
                   Basis: 2
            RecoveryRate: 0.4000
    BusinessDayConvention: "actual"
                Holidays: NaT
        PayAccruedPremium: 1
                Notional: 10000000
                    Name: ""
```

Use `fininstrument` to create a `CDSOption` instrument for two CDS index option instruments.

```
CDSCallOption = fininstrument("cdsoption", 'Strike', OptionStrike, ...
    'ExerciseDate', OptionMaturity, 'OptionType', 'call', 'CDS', CDS, ...
    'Knockout',true, 'AdjustedForwardSpread', AdjustedForwardSpread)
```

```
CDSCallOption =
  CDSOption with properties:

              OptionType: "call"
                  Strike: 140
                Knockout: 1
    AdjustedForwardSpread: 145.7273
            ExerciseDate: 20-Jun-2012
                     CDS: [1x1 fininstrument.CDS]
                    Name: ""
```

```
CDSPutOption = fininstrument("cdsoption", 'Strike', OptionStrike, ...
    'ExerciseDate', OptionMaturity, 'OptionType', 'put', 'CDS', CDS, ...
    'Knockout',true, 'AdjustedForwardSpread', AdjustedForwardSpread)
```

```
CDSPutOption =
  CDSOption with properties:

              OptionType: "put"
                  Strike: 140
                Knockout: 1
    AdjustedForwardSpread: 145.7273
            ExerciseDate: 20-Jun-2012
                     CDS: [1x1 fininstrument.CDS]
                    Name: ""
```

Create CDSBlack Model Object

Use `finmodel` to create a `CDSBlack` model object.

```
CDSOptionModel = finmodel("cdsblack",'SpreadVolatility',SpreadVolatility)
```

```
CDSOptionModel =
  CDSBlack with properties:

    SpreadVolatility: 0.6900
```

Create CDSBlack Pricer Object

Use finpricer to create a CDSBlack pricer object and use the ratecurve object for the zero curve for the 'DiscountCurve' name-value pair argument.

```
CDSOptionpricer = finpricer("analytic",'Model',CDSOptionModel,'DiscountCurve',ZeroCurve,'DefaultI
```

```
CDSOptionpricer =
  CDSBlack with properties:

                     Model: [1x1 finmodel.CDSBlack]
             DiscountCurve: [1x1 ratecurve]
    DefaultProbabilityCurve: [1x1 defprobcurve]
```

Price CDS Index Options

Use price to compute the price for the CDS index options.

```
outPrice = price(CDSOptionpricer, [CDSCallOption;CDSPutOption]);
fprintf('    Payer: %.0f   Receiver: %.0f  \n',outPrice(1),outPrice(2));
```

```
    Payer: 92   Receiver: 66
```

Version History

Introduced in R2020a

See Also

Functions

fininstrument | finmodel

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

NumericalIntegration

Create `NumericalIntegration` pricer object for `Vanilla` instrument using `Heston`, `Bates`, or `Merton` model

Description

Create and price a `Vanilla` instrument object with a `Heston`, `Bates`, or `Merton` model and a `NumericalIntegration` pricing method using this workflow:

1. Use `fininstrument` to create a `Vanilla` instrument object.
2. Use `finmodel` to specify a `Heston`, `Bates`, or `Merton` model for the `Vanilla` instrument object.
3. Use `finpricer` to specify a `NumericalIntegration` pricer object for the `Vanilla` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a `Vanilla` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
NumericalIntegrationPricerObj = finpricer(PricerType,'Model',model,'
DiscountCurve',ratecurve_obj,'SpotPrice',spotprice_value)
NumericalIntegrationPricerObj = finpricer(___,Name,Value)
```

Description

`NumericalIntegrationPricerObj = finpricer(PricerType,'Model',model,'DiscountCurve',ratecurve_obj,'SpotPrice',spotprice_value)` creates a `NumericalIntegration` pricer object by specifying `PricerType` and sets the properties on page 11-3232 for the required name-value pair arguments `Model`, `DiscountCurve`, and `SpotPrice`.

`NumericalIntegrationPricerObj = finpricer(___,Name,Value)` sets optional properties on page 11-3232 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `NumericalIntegrationPricerObj = finpricer("NumericalIntegration",'Model',NIModel,'DiscountCurve',ratecurve_obj,'SpotPrice',1000,'DividendValue',100,'VolRiskPremium',0.9)` creates a `NumericalIntegration` pricer object. You can specify multiple name-value pair arguments.

Input Arguments

`PricerType` — Pricer type

string with value `"NumericalIntegration"` | character vector with value `'NumericalIntegration'`

Pricer type, specified as a string with the value of "NumericalIntegration" or a character vector with the value of 'NumericalIntegration'.

Data Types: char | string

NumericalIntegration Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: NumericalIntegrationPricerObj = finpricer("NumericalIntegration",'Model',NIModel,'DiscountCurve',ratecurve_obj,'SpotPrice',1000,'DividendValue',100,'VolRiskPremium',0.9)

Required NumericalIntegration Name-Value Pair Arguments

Model — Model
model object

Model, specified as the comma-separated pair consisting of 'Model' and the name of a previously created Merton, Bates, or Heston model object using finmodel.

Data Types: object

DiscountCurve — ratecurve object for discounting cash flows
ratecurve object

This property is read-only.

ratecurve object for discounting cash flows, specified as the comma-separated pair consisting of 'DiscountCurve' and the name of a ratecurve object.

Note Specify a flat ratecurve object for DiscountCurve. If you use a nonflat ratecurve object, the software uses the rate in the ratecurve object at Maturity and assumes that the value is constant for the life of the equity option.

Data Types: object

SpotPrice — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of 'SpotPrice' and a scalar nonnegative numeric.

Data Types: double

Optional NumericalIntegration Name-Value Pair Arguments

DividendValue — Dividend yield
0 (default) | scalar numeric

Dividend yield, specified as the comma-separated pair consisting of 'DividendValue' and a scalar numeric.

Data Types: `double`

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, specified as the comma-separated pair consisting of `'VolRiskPremium'` and a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating Little Heston Trap formulation
`true` (default) | logical with values `true` or `false`

Flag indicating Little Heston Trap formulation by Albrecher et al., specified as the comma-separated pair consisting of `'LittleTrap'` and a logical:

- `true` — Use the Albrecher et al. formulation.

  For more information on the `LittleTrap`, see [1] and also the Little Trap formulation is defined by C_j and D_j, see "Heston Stochastic Volatility Model" on page 11-3235 and "Bates Stochastic Volatility Jump Diffusion Model" on page 11-3237.
- `false` — Use the original Heston formation.

Note `LittleTrap` is supported only for `Heston` and `Bates` models.

Data Types: `logical`

`AbsTol` — Absolute error tolerance for numerical integration
`1e-10` (default) | numeric

Absolute error tolerance for numerical integration, specified as the comma-separated pair consisting of `'AbsTol'` and a scalar numeric value.

Data Types: `double`

`RelTol` — Relative error tolerance for numerical integration
`1e-6` (default) | numeric

Relative error tolerance for numerical integration, specified as the comma-separated pair consisting of `'RelTol'` and a scalar numeric value.

Data Types: `double`

`IntegrationRange` — Numerical integration range used to approximate continuous integral over `[0 Inf]`
`[1e-9 Inf]` (default) | vector

Numerical integration range used to approximate the continuous integral over `[0 Inf]`, specified as the comma-separated pair consisting of `'IntegrationRange'` and a 1-by-2 vector representing `[LowerLimit UpperLimit]`.

Data Types: `double`

`Framework` — Framework for computing option prices and sensitivities using numerical integration of models
`"heston1993"` (default) | string with values `"heston1993"` or `"lewis2001"` | character vector with values `'heston1993'` or `'lewis2001'`

Framework for computing option prices and sensitivities using the numerical integration of models, specified as the comma-separated pair consisting of `'Framework'` and a scalar string or character vector with the following values:

- `"heston1993"` or `'heston1993'` — Method used in Heston (1993)
- `"lewis2001"` or `'lewis2001'` — Method used in Lewis (2001)

Data Types: `char` | `string`

Properties

`Model` — Model
model object

Model, returned as a model object.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendValue` — Dividend yield
`0` (default) | scalar numeric

Dividend yield, returned as a scalar numeric.

Data Types: `double`

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, returned as a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating Little Heston Trap formulation
`true` (default) | logical with value `true` or `false`

Flag indicating Little Heston Trap formulation by Albrecher et al., returned as a logical.

Data Types: `logical`

`AbsTol` — Absolute error tolerance for numerical integration
`1e-10` (default) | numeric

Absolute error tolerance for numerical integration, returned as a scalar numeric value.

Data Types: `double`

`RelTol` — Relative error tolerance for numerical integration
`1e-6` (default) | numeric

Relative error tolerance for numerical integration, returned as a scalar numeric value.

Data Types: `double`

`IntegrationRange` — Numerical integration range used to approximate continuous integral over `[0 Inf]`
`[1e-9 Inf]` (default) | vector

Numerical integration range used to approximate the continuous integral over `[0 Inf]`, returned as a 1-by-2 vector representing `[LowerLimit UpperLimit]`.

Data Types: `double`

`Framework` — Framework for computing option prices and sensitivities using numerical integration of models
`"heston1993"` (default) | string with value `"heston1993"` or `"lewis2001"`

Framework for computing option prices and sensitivities using the numerical integration of models, returned as a scalar string.

Data Types: `string`

Object Functions

price Compute price for equity instrument with NumericalIntegration pricer

Examples

Use Numerical Integration Pricer and Merton Model to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `Merton` model and a `NumericalIntegration` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2020,3,15),'ExerciseStyle',"europea
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Mar-2020
           Strike: 105
```

```
             Name: "vanilla_option"
```

Create Merton Model Object

Use `finmodel` to create a `Merton` model object.

```
MertonModel = finmodel("Merton",'Volatility',0.45,'MeanJ',0.02,'JumpVol',0.07,'JumpFreq',0.09)
```

```
MertonModel =
  Merton with properties:

    Volatility: 0.4500
         MeanJ: 0.0200
       JumpVol: 0.0700
      JumpFreq: 0.0900
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
myRC = ratecurve('zero',datetime(2019,9,15),datetime(2020,3,15),0.02)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: 15-Mar-2020
               Rates: 0.0200
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create NumericalIntegration Pricer Object

Use `finpricer` to create a `NumericalIntegration` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("numericalintegration",'Model',MertonModel,'DiscountCurve',myRC,'SpotPrice
```

```
outPricer =
  NumericalIntegration with properties:

                Model: [1x1 finmodel.Merton]
        DiscountCurve: [1x1 ratecurve]
            SpotPrice: 100
         DividendType: "continuous"
        DividendValue: 0.0100
               AbsTol: 0.5000
               RelTol: 0.4000
     IntegrationRange: [1.0000e-09 Inf]
    CharacteristicFcn: @characteristicFcnMerton76
            Framework: "lewis2001"
       VolRiskPremium: 0.9000
```

```
           LittleTrap: 0
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 10.7325

outPR =
  priceresult with properties:

       Results: [1x6 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×6 table
    Price     Delta      Gamma       Theta      Rho       Vega
    ______    ______    _________    _______    ______    ______

    10.732    0.5058    0.012492    -12.969    19.815    27.954
```

More About

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: $\max(K - St, 0)$

Here:

St is the price of the underlying asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Heston Stochastic Volatility Model

The Heston model is an extension of the Black-Scholes model, where the volatility (square root of variance) is no longer assumed to be constant, and the variance now follows a stochastic (CIR) process. This allows modeling the implied volatility smiles observed in the market.

The stochastic differential equation is

$$dS_t = (r - q)S_t dt + \sqrt{v_t} S_t dW_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = pdt$$

Here:

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

v_0 is the initial variance of the asset price at $t = 0$ for ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for the variance for ($\kappa > 0$).

σ_v is the volatility of the variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and $W^v{}_t$ for ($-1 \le p \le 1$).

The characteristic function $f_{Heston_j}(\phi)$ for $j = 1$ (asset price measure) and $j = 2$ (risk-neutral measure) is

$$f_{Heston_j}(\phi) = \exp(C_j + D_j v_0 + i\phi \ln S_t)$$

$$C_j = (r - q)i\phi\tau + \frac{\kappa\theta}{\sigma_v^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1 - g_j e^{d_j \tau}}{1 - g_j}\right)\right]$$

$$D_j = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1 - e^{d_j\tau}}{1 - g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

Here:

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

The definitions for C_j and D_j for the Little Heston Trap by Albrecher et al. (2007) are

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1-\varepsilon_j e^{-d_j\tau}}{1-\varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1-e^{-d_j\tau}}{1-\varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Bates Stochastic Volatility Jump Diffusion Model

The Bates model (Bates 1996) is an extension of the Heston model where, in addition to stochastic volatility, the jump diffusion parameters similar to Merton (1976) are also added to model sudden asset price movements.

The stochastic differential equation is

$$dS_t = (r - q - \lambda_p\mu_J)S_t dt + \sqrt{v_t}S_t dW_t + JS_t dP_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v\sqrt{v_t}dW_t$$

$$\mathrm{E}\left[dW_t dW_t^{\mathrm{v}}\right] = pdt$$

$$\mathrm{prob}(dP_t = 1) = \lambda_p dt$$

Here:

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

J is the random percentage jump size conditional on the jump occurring, where `ln`$(1+J)$ is normally distributed with mean $\ln(1+\mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1+J)\delta\sqrt{2\pi}}\exp\left\{\frac{-\left[\ln(1+J) - \left(\ln(1+\mu_J) - \frac{\delta^2}{2}\right]^2\right.}{2\delta^2}\right\}$$

v_0 is the initial variance of the asset price at $t = 0$ ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for ($\kappa > 0$).

σ_v is the volatility of variance for ($\sigma_{\mathrm{v}} > 0$).

p is the correlation between the Weiner processes W_t and W_t^v for (-1 ≤ p ≤ 1).

μ_J is the mean of J for (μ_J > -1).

δ is the standard deviation of $\ln(1+J)$ for ($\delta \geq 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \geq 0$).

The characteristic function $f_{Bates_j(\phi)}$ for $j = 1$ (asset price mean measure) and $j = 2$ (risk-neutral measure) is

$$f_{Bates(\phi)} = \exp(C_j + D_j v_0 + i\phi \ln S_t)\exp\left(\lambda_p \tau (1+\mu_J)^{m_j + \frac{1}{2}}\left[(1+\mu_j)^{i\phi} e^{\delta^2 (m_j i\phi + \frac{(i\phi)^2}{2})} - 1\right] - \lambda_p \tau \mu_J i\phi\right)$$

$$m_j = \begin{Bmatrix} m_1 = \frac{1}{2} \\ m_2 = -\frac{1}{2} \end{Bmatrix}$$

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1-g_j e^{d_j\tau}}{1-g_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1-e^{d_j\tau}}{1-g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

Here:

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity for ($\tau = T - t$).

i is the unit imaginary number for ($i^2 = -1$).

The definitions for C_j and D_j for the Little Heston Trap by Albrecher et al. (2007) are

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v^2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1-\varepsilon_j e^{-d_j\tau}}{1-\varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1-e^{-d_j\tau}}{1-\varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Merton Jump Diffusion Model

The Merton jump diffusion model (Merton 1976) is an extension of the Black-Scholes model, where sudden asset price movements (both up and down) are modeled by adding the jump diffusion parameters with the Poisson process.

The stochastic differential equation is

$$dS_t = (r - q - \lambda_p \mu_j) S_t dt + \sigma S_t dW_t + J S_t dP_t$$
$$\text{prob}(dP_t = 1) = \lambda_p dt$$

Here:

r is the continuous risk-free rate.

q is the continuous dividend yield.

W_t is the Weiner process.

J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1 + \mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution

$$\frac{1}{(1+J)\delta\sqrt{2\pi}} \exp\left\{ \frac{-\left[\ln(1+J) - \left(\ln(1+\mu_J) - \frac{\delta^2}{2}\right]\right]^2}{2\delta^2} \right\}$$

μ_J is the mean of J for (μ_J > -1).

δ is the standard deviation of $\ln(1+J)$ for ($\delta \geq 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \geq 0$).

σ is the volatility of the asset price for ($\sigma > 0$).

The characteristic function $f_{Merton76_j}(\phi)$ for $j = 1$ (asset prices measure) and $j = 2$ (risk-neutral measure) is

$$f_{Merton76_j} = f_{BS_j} \exp\left(\lambda_p \tau \left(1 + \mu_j\right)^{m_j} + \frac{1}{2} \left[\left(1 + \mu_j\right)^{i\phi} e^{\delta^2 \left(m_j i\phi + \frac{(i\phi)^2}{2} \right)} - 1 \right] - \lambda_p \tau \mu_j i\phi \right)$$

where for $j = 1, 2$:

$$f_{BS_1}(\phi) = \frac{f_{BS_2}(\phi - i)}{f_{BS_2}(-i)}$$

$$f_{BS2}(\phi) = \exp\left(i\phi \left[\ln S_t + \left(r - q - \frac{\sigma^2}{2} \right) \tau \right] - \frac{\phi^2 \sigma^2}{2} \tau \right)$$

$$m_1 = \frac{1}{2}, m_2 = -\frac{1}{2}$$

Here:

ϕ is the characteristic function variable

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

Numerical Integration Method Under Heston (1993) Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Heston (1993) framework is based on the following expressions

$$Call(K) = S_t e^{-q\tau} P_1 - K e^{-r\tau} P_2$$

$$Put(K) = Call(K) + K e^{-r\tau} - S_t e^{-q\tau}$$

$$P_j = \frac{1}{2} + \frac{1}{\pi} \int_0^{\infty} \mathrm{Re}\left[\frac{e^{-i\phi \ln(K)} f_j(\phi)}{i\phi} \right] d\phi$$

Here:

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity ($\tau = T - t$).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number ($i^2 = -1$).

ϕ is the characteristic function variable.

$f_j(\phi)$ is the characteristic function for $P_j (j = 1,2)$.

P_1 is the probability of $S_t > K$ under the asset price measure for the model.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

Where $j = 1,2$ so that $f_1(\phi)$ and $f_2(\phi)$ are the characteristic functions for probabilities P_1 and P_2, respectively.

Choose this framework by specifying the default value `"Heston1993"` for the `Framework` name-value pair argument.

Numerical Integration Method Under Lewis Framework

Numerical integration is used to evaluate the continuous integral for the inverse Fourier transform.

The numerical integration method under the Lewis (2001) framework is based on the following expressions:

$$Call(k) = S_t e^{-q\tau} - \frac{\sqrt{K}e^{-\tau t}}{\pi}\int_0^{\infty} \mathrm{Re}\left[K^{-iu} f_2\left(\phi = \left(u - \frac{i}{2}\right)\right)\frac{1}{u^2 + \frac{1}{4}}\right]du$$

$$Put(K) = Call(K) = Ke^{-\tau t} - S_t e^{-q\tau}$$

Here

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

K is the strike.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K.

i is a unit imaginary number (i^2= -1).

ϕ is the characteristic function variable.

u is the characteristic function variable for integration, where $\phi = \left(u - \frac{i}{2}\right)$.

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

Choose this framework by specifying the value `"Lewis2001"` for the `Framework` name-value pair argument.

Version History

Introduced in R2020a

References

[1] Albrecher, H., P. Mayer, W. Schoutens, and J. Tistaert. “The Little Heston Trap.” Working Paper, Linz and Graz University of Technology, K.U. Leuven, ING Financial Markets, 2006.

See Also

Functions

`fininstrument` | `finmodel` | `timetable`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Discount

Create `Discount` pricer object for `Deposit`, `FRA`, `Swap`, `FixedBond`, `FloatBond`, `OISFuture`, `STIRFuture`, and `OvernightIndexedSwap` using `ratecurve` object

Description

Create and price a `Deposit`, `FRA`, `Swap`, `FixedBond`, `FloatBond`, `OISFuture`, `STIRFuture`, and `OvernightIndexedSwap` instrument object with a `ratecurve` and a `Discount` pricing method using this workflow:

1. Create an interest-rate curve object using `ratecurve`.
2. Use `finpricer` to specify a `Discount` pricer object for the `Deposit`, `FRA`, `Swap`, `FixedBond`, `FloatBond`, `STIRFuture`, `OISFuture`, or `OvernightIndexedSwap` instrument object.

Note If you do not specify `ProjectionCurve` when you create a `Swap` or `FloatBond` instrument with the `Discount` pricer, the `ProjectionCurve` value defaults to the `DiscountCurve` value.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Deposit`, `FRA`, `Swap`, `FixedBond`, or `FloatBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
DiscountPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_object)
```

Description

`DiscountPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_object)` creates a `Discount` pricer object by specifying `PricerType` and the required name-value pair argument `DiscountCurve` to set properties on page 11-3244 using name-value pairs. For example, `DiscountPricerObj = finpricer("Discount",'DiscountCurve',ratecurve_obj)` creates a `Discount` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Discount"` | character vector with value `'Discount'`

Pricer type, specified as a string with the value of `"Discount"` or a character vector with the value of `'Discount'`.

Data Types: `char` | `string`

`Discount` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `DiscountPricerObj = finpricer("Discount",'DiscountCurve',ratecurve_obj)`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a previously created `ratecurve` object

Data Types: `object`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as the `ratecurve` object.

Data Types: `object`

Object Functions

price Compute price for interest-rate instrument with Discount pricer

Examples

Use Discount Pricer and `ratecurve` to Price Swap Instrument

This example shows the workflow to price a `Swap` instrument when using a `ratecurve` and a `Discount` pricing method.

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve` for the underlying interest-rate curve for the `Swap` instrument.

```
Settle = datetime(2022,1,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:
```

```
                Type: "zero"
        Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Jan-2022
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use `fininstrument` to create a Swap instrument object.

```
SwapOpt = fininstrument("Swap",'Maturity',datetime(2027,1,15),'LegRate',[0.024 0.015],'LegType',
```

```
SwapOpt =
  Swap with properties:

                     LegRate: [0.0240 0.0150]
                     LegType: ["fixed"    "float"]
                       Reset: [2 2]
                       Basis: [0 0]
                    Notional: 100
          LatestFloatingRate: [NaN NaN]
                 ResetOffset: [0 0]
    DaycountAdjustedCashFlow: [0 0]
             ProjectionCurve: [1x2 ratecurve]
       BusinessDayConvention: ["actual"    "actual"]
                    Holidays: NaT
                EndMonthRule: [1 1]
                   StartDate: NaT
                    Maturity: 15-Jan-2027
                        Name: "swap_instrument"
```

Create `Discount` Pricer Object

Use `finpricer` to create a `Discount` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("Discount", 'DiscountCurve',myRC)
```

```
outPricer =
  Discount with properties:

    DiscountCurve: [1x1 ratecurve]
```

Price Swap Instrument

Use `price` to compute the price and sensitivities for the Swap instrument.

```
[Price, outPR] = price(outPricer, SwapOpt,["all"])
```

```
Price = -1.3834
```

```
outPR =
  priceresult with properties:

      Results: [1x2 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×2 table
     Price       DV01
    _______    ________

    -1.3834    0.048336
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Future

Create `Future` pricer object for `BondFuture`, `CommodityFuture`, `EquityIndexFuture`, and `FXFuture` using `ratecurve` object

Description

Create and price a `BondFuture`, `CommodityFuture`, `EquityIndexFuture`, and `FXFuture` instrument object with a `ratecurve` object and a `Future` pricing method using this workflow:

1. Create an interest-rate curve object using `ratecurve`.
2. Use `fininstrument` to create a `BondFuture`, `CommodityFuture`, `FXFuture`, or `EquityIndexFuture` instrument object.
3. Use `finpricer` to specify a `Future` pricer object for the `BondFuture`, `CommodityFuture`, `FXFuture`, or `EquityIndexFuture` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `BondFuture`, `CommodityFuture`, `EquityIncomeFuture`, or `FXFuture` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
FuturePricerObj = finpricer(PricerType,DiscountCurve=ratecurve_object,
SpotPrice=spot_price)
```

Description

`FuturePricerObj = finpricer(PricerType,DiscountCurve=ratecurve_object, SpotPrice=spot_price)` creates a `Future` pricer object by specifying `PricerType` and the required name-value arguments for `DiscountCurve` and `SpotPrice` to set properties on page 11-3248. For example, `FuturePricerObj = finpricer("Future",DiscountCurve=ratecurve_obj,SpotPrice=125)` creates a `Future` pricer object.

Input Arguments

`PricerType` — Pricer type
string with value `"Future"` | character vector with value `'Future'`

Pricer type, specified as a string with the value of `"Future"` or a character vector with the value of `'Future'`.

Data Types: `char` | `string`

Future Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `FuturePricerObj = finpricer("Future",DiscountCurve=ratecurve_obj,SpotPrice=125)`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as `DiscountCurve` and the name of a previously created `ratecurve` object

Data Types: `object`

`SpotPrice` — Quoted spot price for underlying asset to be delivered
numeric

Quoted spot price for underlying asset to be delivered, specified as `SpotPrice` and a numeric value that depends on the type of future instrument being priced:

- `BondFuture` instrument — Clean spot price quoted for $100 face value of underlying bond
- `CommodityFuture` instrument — Spot price for underlying commodity quantity specified in contract
- `EquityIndexFuture` instrument — Spot equity index value
- `FXFuture` instrument — Spot price quoted in domestic currency for one unit of foreign currency

Data Types: `double`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as the `ratecurve` object.

Data Types: `object`

`SpotPrice` — Quoted spot price for underlying asset to be delivered
numeric

Quoted spot price for underlying asset to be delivered, returned as a numeric value that depends on the type of future instrument being priced.

Data Types: `double`

Object Functions

price Compute price for interest-rate instrument with Future pricer

Examples

Use Future Pricer and ratecurve to Price BondFuture Instrument

This example shows the workflow to price a BondFuture instrument when you use a ratecurve object and a Future pricing method.

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2022,3,1);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates,Compounding=2);
```

Create Underlying FixedBond Instrument Object

Use fininstrument to create a FixedBond instrument object.

```
FixB = fininstrument("FixedBond",Maturity=datetime(2032,9,1),CouponRate=0.05,Name="fixed_bond_ins
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Sep-2032
                        Name: "fixed_bond_instrument"
```

Create BondFuture Instrument Object

Use fininstrument to create a BondFuture instrument object.

```
BondFut = fininstrument("BondFuture",Maturity=datetime(2022,9,1),QuotedPrice=86,Bond=FixB,Convers
```

```
BondFut =
  BondFuture with properties:

           Maturity: 01-Sep-2022
        QuotedPrice: 86
               Bond: [1x1 fininstrument.FixedBond]
    ConversionFactor: 1.4300
           Notional: 100000
               Name: "bondfuture_instrument"
```

Create Future Pricer Object

Use `finpricer` to create a `Future` pricer object and use the `ratecurve` object with the `DiscountCurve` name-value argument.

```
outPricer = finpricer("Future",DiscountCurve=ZeroCurve,SpotPrice=125)
```

```
outPricer =
  Future with properties:

    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 125
```

Price BondFuture Instrument

Use `price` to compute the price and price result for the `BondFuture` instrument.

```
[Price,outPR] = price(outPricer,BondFut)
```

```
Price = -151.9270

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×4 table
     Price     FairDeliveryPrice    FairFuturePrice    AccruedInterest
    _______    _________________    _______________    _______________

    -151.93       1.2283e+05            85.893                0
```

Version History

Introduced in R2022a

See Also

Functions

`fininstrument` | `finmodel` | `ratecurve` | `fairdelivery` | `cashsettle`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

FFT

Create FFT pricer object for Vanilla instrument using Merton, Heston, or Bates model

Description

Create and price a Vanilla instrument object with a Heston, Bates, or Merton model and an FFT pricing method using this workflow:

1. Use finstrument to create a Vanilla instrument object.
2. Use finmodel to specify a Heston, Bates, or Merton model for the Vanilla instrument object.
3. Use finpricer to specify an FFT pricer object for the Vanilla instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a Vanilla instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
FFTPricerObj = finpricer(PricerType,'Model',model,'
DiscountCurve',ratecurve_obj)
FFTPricerObj = finpricer( ___ ,Name,Value)
```

Description

`FFTPricerObj = finpricer(PricerType,'Model',model,'DiscountCurve',ratecurve_obj)` creates an FFT pricer object by specifying PricerType and sets the properties on page 11-3254 for the required name-value pair arguments Model and DiscountCurve.

`FFTPricerObj = finpricer( ___ ,Name,Value)` sets optional properties on page 11-3254 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `FFTPricerObj = finpricer("FFT",'Model',FFTModel, 'DiscountCurve',ratecurve_obj,'SpotPrice',1000,'DividendValue',0.01,'VolRiskPremium',0.9)` creates an FFT pricer object. You can specify multiple name-value pair arguments.

Input Arguments

PricerType — Pricer type
string with value "FFT" | character vector with value 'FFT'

Pricer type, specified as a string with the value of "FFT" or a character vector with the value of 'FFT'.

Data Types: char | string

FFT Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: FFTPricerObj = finpricer("FFT",'Model',FFTModel, 'DiscountCurve',ratecurve_obj,'SpotPrice',1000,'DividendValue',0.01,'VolRiskPremium',0.9)

Required FFT Name-Value Pair Arguments

Model — Model
model object

Model, specified as the comma-separated pair consisting of 'Model' and the name of the previously created Merton, Bates, or Heston model object using finmodel.

Data Types: object

DiscountCurve — ratecurve object for discounting cash flows
ratecurve object

This property is read-only.

ratecurve object for discounting cash flows, specified as the comma-separated pair consisting of 'DiscountCurve' and the name of the ratecurve object.

Note Specify a flat ratecurve object for DiscountCurve. If you use a nonflat ratecurve object, the software uses the rate in the ratecurve object at Maturity and assumes that the value is constant for the life of the equity option.

Data Types: object

SpotPrice — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of 'SpotPrice' and a scalar nonnegative numeric.

Data Types: double

Optional FFT Name-Value Pair Arguments

DividendValue — Dividend yield
0 (default) | scalar nonnegative numeric

Dividend yield, specified as the comma-separated pair consisting of 'DividendValue' and a scalar nonnegative numeric in decimals.

Data Types: double

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, specified as the comma-separated pair consisting of `'VolRiskPremium'` and a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating Little Heston Trap formulation
`true` (default) | logical with value `true` or `false`

Flag indicating Little Heston Trap formulation by Albrecher et al., specified as the comma-separated pair consisting of `'LittleTrap'` and a logical:

- `true` — Use the Albrecher et al. formulation.

  For more information on the `LittleTrap`, see [1] and also the Little Trap formulation is defined by C_j and D_j in "Heston Stochastic Volatility Model" on page 11-3258 and "Bates Stochastic Volatility Jump Diffusion Model" on page 11-3259.
- `false` — Use the original Heston formation.

Note `LittleTrap` is supported only for `Heston` and `Bates` models.

Data Types: `logical`

`NumFFT` — Number of grid points in characteristic function variable
`4096` (default) | numeric

Number of grid points in the characteristic function variable and in each column of the log-strike grid, specified as the comma-separated pair consisting of `'NumFFT'` and a scalar numeric value.

Data Types: `double`

`CharacteristicFcnStep` — Characteristic function variable grid spacing
`0.01` (default) | numeric

Characteristic function variable grid spacing, specified as the comma-separated pair consisting of `'CharacteristicFcnStep'` and a scalar numeric value.

Data Types: `double`

`LogStrikeStep` — Log-strike grid spacing
`2*pi/NumFFT/CharacteristicFcnStep` (default) | numeric

Log-strike grid spacing, specified as the comma-separated pair consisting of `'LogStrikeStep'` and a scalar numeric value.

Note If (`LogStrikeStep*CharacteristicFcnStep`) is `2*pi/NumFFT`, FFT is used. Otherwise, FRFT is used.

Data Types: `double`

`DampingFactor` — Damping factor for Carr-Madan formulation
`1.5` (default) | numeric

Damping factor for the Carr-Madan formulation, specified as the comma-separated pair consisting of `'DampingFactor'` and a scalar numeric value.

Data Types: `double`

`Quadrature` — Type of quadrature
`"simpson"` (default) | string with value `"simpson"` or `"trapezoidal"` | character vector with value `'simpson'` or `'trapezoidal'`

Type of quadrature, specified as the comma-separated pair consisting of `'Quadrature'` and a scalar string or character vector.

Data Types: `char` | `string`

Properties

`Model` — Model
model object

Model, returned as a model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendValue` — Dividend yield
0 (default) | scalar nonnegative numeric

Dividend yield, returned as a scalar nonnegative numeric in decimals.

Data Types: `double`

`VolRiskPremium` — Volatility risk premium
`0` (default) | numeric

Volatility risk premium, returned as a scalar numeric value.

Data Types: `double`

`LittleTrap` — Flag indicating Little Heston Trap formulation
`true` (default) | logical with value `true` or `false`

Flag indicating Little Heston Trap formulation by Albrecher et al., returned as a logical.

Data Types: `logical`

`NumFFT` — Number of grid points in the characteristic function variable
`4096` (default) | numeric

Number of grid points in the characteristic function variable and in each column of the log-strike grid, returned as a scalar numeric value.

Data Types: `double`

`CharacteristicFcnStep` — Characteristic function variable grid spacing
`0.01` (default) | numeric

Characteristic function variable grid spacing, returned as a scalar numeric value.

Data Types: `double`

`LogStrikeStep` — Log-strike grid spacing
`2*pi/NumFFT/CharacteristicFcnStep` (default) | numeric

Log-strike grid spacing, returned as a scalar numeric value.

Data Types: `double`

`DampingFactor` — Damping factor for Carr-Madan formulation
`1.5` (default) | numeric

Damping factor for the Carr-Madan formulation, returned as a scalar numeric value.

Data Types: `double`

`Quadrature` — Type of quadrature
`"simpson"` (default) | string with value `"simpson"` or `"trapezoidal"`

Type of quadrature, returned as a string.

Data Types: `string`

Object Functions

price Compute price for equity instrument with FFT pricer

Examples

Use FFT Pricer and Heston Model to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `Heston` model and an `FFT` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2022,9,15),'Strike',105,'ExerciseSty
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2022
           Strike: 105
```

```
             Name: "vanilla_option"
```

Create Heston Model Object

Use `finmodel` to create a `Heston` model object.

```
HestonModel = finmodel("Heston",'V0',0.032,'ThetaV',0.1,'Kappa',0.003,'SigmaV',0.2,'RhoSV',0.9)
```

```
HestonModel =
  Heston with properties:

       V0: 0.0320
    ThetaV: 0.1000
     Kappa: 0.0030
    SigmaV: 0.2000
     RhoSV: 0.9000
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create FFT Pricer Object

Use `finpricer` to create an `FFT` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("fft",'DiscountCurve',myRC,'Model',HestonModel,'SpotPrice',100,'Characteris
```

```
outPricer =
  FFT with properties:

                  Model: [1x1 finmodel.Heston]
          DiscountCurve: [1x1 ratecurve]
              SpotPrice: 100
           DividendType: "continuous"
          DividendValue: 0
                 NumFFT: 8192
    CharacteristicFcnStep: 0.2000
```

```
        LogStrikeStep: 0.0038
    CharacteristicFcn: @characteristicFcnHeston
        DampingFactor: 1.5000
           Quadrature: "simpson"
       VolRiskPremium: 0
           LittleTrap: 1
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 14.7545

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma       Theta       Rho       Vega     VegaLT
    _____    _______    _________    ________    ______    ______    ______

    14.754    0.44868    0.021649    -0.20891    120.45    88.192    1.3248
```

More About

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: $\max(K - St, 0)$

Here:

St is the price of the underlying asset at time t.

K is the strike price.

For more information, see “Vanilla Option” on page 3-27.

Heston Stochastic Volatility Model

The Heston model is an extension of the Black-Scholes model, where the volatility (the square root of the variance) is no longer assumed to be constant, and the variance now follows a stochastic (CIR) process. This process allows modeling the implied volatility smiles observed in the market.

The stochastic differential equation is

$$dS_t = (r - q)S_t dt + \sqrt{v_t} S_t dW_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = p dt$$

Here:

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

v_0 is the initial variance of the asset price at $t = 0$ for ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for the variance for ($\kappa > 0$).

σ_v is the volatility of the variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and $W^v{}_t$ for ($-1 \le p \le 1$).

The characteristic function $f_{Heston_j}(\phi)$ for $j = 1$ (asset price measure) and $j = 2$ (risk-neutral measure) is

$$f_{Heston_j}(\phi) = \exp(C_j + D_j v_0 + i\phi \ln S_t)$$

$$C_j = (r - q)i\phi\tau + \frac{\kappa\theta}{\sigma_v^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1 - g_j e^{d_j \tau}}{1 - g_j}\right)\right]$$

$$D_j = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1 - e^{d_j\tau}}{1 - g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

Here:

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

The definitions for C_j and D_j in the Little Heston Trap by Albrecher et al. (2007) are

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1-\varepsilon_j e^{-d_j\tau}}{1-\varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1-e^{-d_j\tau}}{1-\varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Bates Stochastic Volatility Jump Diffusion Model

The Bates model (Bates 1996) is an extension of the Heston model where, in addition to stochastic volatility, the jump diffusion parameters similar to Merton (1976) are also added to model sudden asset price movements.

The stochastic differential equation is

$$dS_t = (r - q - \lambda_p\mu_J)S_t dt + \sqrt{v_t}S_t dW_t + JS_t dP_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma_v\sqrt{v_t}dW_t$$

$$\mathrm{E}\left[dW_t dW_t^v\right] = pdt$$

$$\mathrm{prob}(dP_t = 1) = \lambda_p dt$$

Here:

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

v_t is the asset price variance at time t.

J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1+\mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1+J)\delta\sqrt{2\pi}}\exp\left\{\frac{-\left[\ln(1+J) - \left(\ln(1+\mu_J) - \frac{\delta^2}{2}\right]^2\right.}{2\delta^2}\right\}$$

Here:

v_0 is the initial variance of the asset price at $t = 0$ ($v_0 > 0$).

θ is the long-term variance level for ($\theta > 0$).

κ is the mean reversion speed for ($\kappa > 0$).

σ_v is the volatility of variance for ($\sigma_v > 0$).

p is the correlation between the Weiner processes W_t and W_t^v for ($-1 \le p \le 1$).

μ_J is the mean of J for ($\mu_J > -1$).

δ is the standard deviation of $\ln(1+J)$ for ($\delta \ge 0$).

λ_p is the annual frequency (intensity) of Poisson process P_t for ($\lambda_p \ge 0$).

The characteristic function $f_{Bates_j(\phi)}$ for $j = 1$ (asset price mean measure) and $j = 2$ (risk-neutral measure) is

$$f_{Bates(\phi)} = \exp(C_j + D_j v_0 + i\phi \ln S_t)\exp\left(\lambda_p \tau(1+\mu_J)^{m_j} + \frac{1}{2}\left[(1+\mu_j)^{i\phi} e^{\delta^2\left(m_j i\phi + \frac{(i\phi)^2}{2}\right)} - 1\right] - \lambda_p \tau \mu_J i\phi\right)$$

$$m_j = \begin{Bmatrix} m_1 = \frac{1}{2} \\ m_2 = -\frac{1}{2} \end{Bmatrix}$$

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v^2}\left[\left(b_j - p\sigma_v i\phi + d_j\right)\tau - 2\ln\left(\frac{1 - g_j e^{d_j\tau}}{1 - g_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi + d_j}{\sigma_v^2}\left(\frac{1 - e^{d_j\tau}}{1 - g_j e^{d_j\tau}}\right)$$

$$g_j = \frac{b_j - p\sigma_v i\phi + d_j}{b_j - p\sigma_v i\phi - d_j}$$

$$d_j = \sqrt{\left(b_j - p\sigma_v i\phi\right)^2 - \sigma_v^2\left(2u_j i\phi - \phi^2\right)}$$

where for $j = 1, 2$:

$$u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda_{VolRisk} - p\sigma_v, b_2 = \kappa + \lambda_{VolRisk}$$

Here:

ϕ is the characteristic function variable.

$\lambda_{VolRisk}$ is the volatility risk premium.

τ is the time to maturity for ($\tau = T - t$).

i is the unit imaginary number for ($i^2 = -1$).

The definitions for C_j and D_j in the Little Heston Trap by Albrecher et al. (2007) are

$$C_j = (r-q)i\phi\tau + \frac{\kappa\theta}{\sigma_v{}^2}\left[\left(b_j - p\sigma_v i\phi - d_j\right)\tau - 2\ln\left(\frac{1-\varepsilon_j e^{-d_j\tau}}{1-\varepsilon_j}\right)\right]$$

$$Dj = \frac{b_j - p\sigma_v i\phi - d_j}{\sigma_v^2}\left(\frac{1-e^{-d_j\tau}}{1-\varepsilon_j e^{-d_j\tau}}\right)$$

$$\varepsilon_j = \frac{b_j - p\sigma_v i\phi - d_j}{b_j - p\sigma_v i\phi + d_j}$$

Merton Jump Diffusion Model

The Merton jump diffusion model (Merton 1976) is an extension of the Black-Scholes model, where sudden asset price movements (both up and down) are modeled by adding the jump diffusion parameters with the Poisson process.

The stochastic differential equation is

$$dS_t = (r - q - \lambda_p\mu_j)S_t dt + \sigma S_t dW_t + JS_t dP_t$$
$$\text{prob}(dP_t = 1) = \lambda_p dt$$

Here:

r is the continuous risk-free rate.

q is the continuous dividend yield.

W_t is the Weiner process.

J is the random percentage jump size conditional on the jump occurring, where $\ln(1+J)$ is normally distributed with mean $\ln(1+\mu_J) - \frac{\delta^2}{2}$ and the standard deviation δ, and $(1+J)$ has a lognormal distribution:

$$\frac{1}{(1+J)\delta\sqrt{2\pi}}\exp\left\{\frac{-\left[\ln(1+J) - \left(\ln(1+\mu_J) - \frac{\delta^2}{2}\right]^2\right.}{2\delta^2}\right\}$$

Here:

μ_J is the mean of J for ($\mu_J > -1$).

δ is the standard deviation of $\ln(1+J)$ for ($\delta \geq 0$).

λ_p is the annual frequency (intensity) of the Poisson process P_t for ($\lambda_p \geq 0$).

σ is the volatility of the asset price for ($\sigma > 0$).

The characteristic function $f_{Merton76_j}(\phi)$ for $j = 1$ (asset price measure) and $j = 2$ (risk-neutral measure) is

$$f_{Merton76_j} = f_{BS_j}\exp\left(\lambda_p\tau\left(1+\mu_j\right)^{m_j+\frac{1}{2}}\left[\left(1+\mu_j\right)^{i\phi}e^{\delta^2\left(m_j i\phi+\frac{(i\phi)^2}{2}\right)}-1\right]-\lambda_p\tau\mu_j i\phi\right)$$

where for $j = 1, 2$:

$$f_{BS_1}(\phi) = \frac{f_{BS_2}(\phi - i)}{f_{BS_2}(-i)}$$

$$f_{BS2}(\phi) = \exp\left(i\phi\left[\ln S_t + \left(r - q - \frac{\sigma^2}{2}\right)\tau\right] - \frac{\phi^2\sigma^2}{2}\tau\right)$$

$$m_1 = \frac{1}{2}, m_2 = -\frac{1}{2}$$

Here:

ϕ is the characteristic function variable.

τ is the time to maturity ($\tau = T - t$).

i is the unit imaginary number ($i^2 = -1$).

Carr-Madan Formulation

The Carr-Madan (1999) formulation is a popular modified implementation of Heston (1993) framework.

Rather than computing the probabilities P_1 and P_2 as intermediate steps, Carr and Madan developed an alternative expression so that taking its inverse Fourier transform gives the option price itself directly.

$$Call(k) = \frac{e^{-\alpha k}}{\pi}\int_0^\infty \mathrm{Re}\left[e^{-iuk}\psi(u)\right]du$$

$$\psi(u) = \frac{e^{-r\tau}f_2(\phi = (u - (\alpha+1)i))}{\alpha^2 + \alpha - u^2 + iu(2\alpha+1)}$$

$$Put(K) = Call(K) + Ke^{-r\tau} - S_t e^{-q\tau}$$

Here:

r is the continuous risk-free rate.

q is the continuous dividend yield.

S_t is the asset price at time t.

τ is time to maturity (τ = T-t).

$Call(K)$ is the call price at strike K.

$Put(K)$ is the put price at strike K

i is a unit imaginary number (i^2= -1)

ϕ is the characteristic function variable.

α is the damping factor.

u is the characteristic function variable for integration, where ϕ = (u - (α+1)i).

$f_2(\phi)$ is the characteristic function for P_2.

P_2 is the probability of $S_t > K$ under the risk-neutral measure for the model.

To apply FFT or FRFT to this formulation, the characteristic function variable for integration u is discretized into `NumFFT`(N) points with the step size `CharacteristicFcnStep` (Δu), and the log-strike k is discretized into N points with the step size `LogStrikeStep`(Δk).

The discretized characteristic function variable for integration u_j(for j = 1,2,3,...,N) has a minimum value of 0 and a maximum value of (N-1) (Δu), and it approximates the continuous integration range from 0 to infinity.

The discretized log-strike grid k_n(for n = 1, 2, 3, N) is approximately centered around `ln`(S_t), with a minimum value of

$$\ln(S_t) - \frac{N}{2}\Delta k$$

and a maximum value of

$$\ln(S_t) + \left(\frac{N}{2} - 1\right)\Delta k$$

Where the minimum allowable strike is

$$S_t \exp\left(-\frac{N}{2}\Delta k\right)$$

and the maximum allowable strike is

$$S_t \exp\left[\left(\frac{N}{2} - 1\right)\Delta k\right]$$

As a result of the discretization, the expression for the call option becomes

$$Call(k_n) = \Delta u \frac{e^{-\alpha k_n}}{\pi} \sum_{j=1}^{N} \mathrm{Re}\left[e^{-i\Delta k \Delta u (j-1)(n-1)} e^{iuj\left[\frac{N\Delta k}{2} - \ln(S_t)\right]} \psi(u_j)\right] w_j$$

Here:

Δu is the step size of the discretized characteristic function variable for integration.

Δk is the step size of the discretized log strike.

N is the number of FFT or FRFT points.

w_j is the weights for quadrature used for approximating the integral.

FFT is used to evaluate the above expression if Δk and Δu are subject to the following constraint:

$$\Delta k \Delta u = \left(\frac{2\pi}{N}\right)$$

Otherwise, the functions use the FRFT method described in Chourdakis (2005).

Version History

Introduced in R2020a

References

[1] Albrecher, H., P. Mayer, W. Schoutens, and J. Tistaert. “The Little Heston Trap.” Working Paper, Linz and Graz University of Technology, K.U. Leuven, ING Financial Markets, 2006.

See Also

Functions

`fininstrument` | `finmodel`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

GoldmanSosinGatto

Create `GoldmanSosinGatto` pricer object for `Lookback` instrument using `BlackScholes` model

Description

Create and price a `Lookback` instrument object with a `BlackScholes` model and a `GoldmanSosinGatto` pricing method using this workflow:

1. Use `fininstrument` to create a `Lookback` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Lookback` instrument object.
3. Use `finpricer` to specify a `GoldmanSosinGatto` pricer object for the `Lookback` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Lookback` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
GoldmanSosinGattoPricerObj = finpricer(PricerType,'
DiscountCurve,ratecurve_obj,'Model,model,'SpotPrice',spotprice_value)
GoldmanSosinGattoPricerObj = finpricer(___,Name,Value)
```

Description

`GoldmanSosinGattoPricerObj = finpricer(PricerType,'DiscountCurve,ratecurve_obj,'Model,model,'SpotPrice',spotprice_value)` creates a `GoldmanSosinGatto` pricer object by specifying `PricerType` and sets the properties on page 11-3267 for the required name-value pair arguments `DiscountCurve`, `Model`, and `SpotPrice`.

`GoldmanSosinGattoPricerObj = finpricer(___,Name,Value)` to set optional properties on page 11-3267 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `GoldmanSosinGattoPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',500,'PricingMethod',"GoldmanSosinGatto")` creates a `GoldmanSosinGatto` pricer object.

Input Arguments

PricerType — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: `char` | `string`

GoldmanSosinGatto Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `GoldmanSosinGattoPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',500,'PricingMethod',"GoldmanSosinGatto")`

Required GoldmanSosinGatto Name-Value Pair Arguments

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of the previously created `ratecurve` object.

Note Specify a flat `ratecurve` object for `DiscountCurve`. If you use a nonflat `ratecurve` object, the software uses the rate in the `ratecurve` object at `Maturity` and assumes that the value is constant for the life of the equity option.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, specified as the comma-separated pair consisting of `'Model'` and the name of a previously created `BlackScholes` model object using `finmodel`.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of `'SpotPrice'` and a scalar nonnegative numeric.

Data Types: `double`

Optional GoldmanSosinGatto Name-Value Pair Arguments

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"cash"` or `"continuous"` | character vector with value `'cash'` or `'continuous'`

Stock dividend type, specified as the comma-separated pair consisting of `'DividendType'` and a character vector or string. `DividendType` must be `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `char` | `string`

`DividendValue` — Dividend amount or dividend schedule for underlying stock
`0` (default) | scalar numeric | timetable

Dividend amount for underlying stock, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar numeric for a dividend amount or a timetable for a dividend schedule.

Note Specify a scalar if `DividendType` is `"continuous"` and a timetable if `DividendType` is `"cash"`.

Data Types: `double` | `timetable`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"GoldmanSosinGatto"` | character vector with value `'GoldmanSosinGatto'`

Analytic pricing method, specified as the comma-separated pair consisting of `'PricingMethod'` and a character vector or string.

Note The default pricing method for a `BlackScholes` model is a `BlackScholes` pricer.

Data Types: `double`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"cash"` or `"continuous"`

This property is read-only.

Stock dividend type, returned as a string. `DividendType` is either `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `string`

`DividendValue` — Dividend amount or dividend schedule for underlying stock
`0` (default) | scalar nonnegative numeric | timetable

Dividend amount or dividend schedule for the underlying stock, returned as a scalar numeric for a dividend amount or a timetable for a dividend schedule.

Data Types: `double` | `timetable`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"GoldmanSosinGatto"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Goldman-Sosin-Gatto Pricer and Black-Scholes Model to Price Lookback Instrument

This example shows the workflow to price a floating-strike `Lookback` instrument when you use a `BlackScholes` model and a `GoldmanSosinGatto` pricing method.

Create Lookback Instrument Object

Use `fininstrument` to create a floating-strike `Lookback` instrument object where the `Strike` argument is specified as `NaN`.

```
LookbackOpt = fininstrument("lookback",'Strike',NaN,'ExerciseDate',datetime(2021,9,15),'OptionTy
```

```
LookbackOpt =
  Lookback with properties:

       OptionType: "put"
           Strike: NaN
      AssetMinMax: NaN
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2021
             Name: "lookback_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',.358)
```

```
BlackScholesModel =
  BlackScholes with properties:
```

```
     Volatility: 0.3580
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                 Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create GoldmanSosinGatto Pricer Object

Use `finpricer` to create a `GoldmanSosinGatto` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',100,
```

```
outPricer =
  GoldmanSosinGatto with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0250
     DividendType: "continuous"
```

Price Lookback Instrument

Use `price` to compute the price and sensitivities for the floating-strike `Lookback` instrument.

```
[Price, outPR] = price(outPricer,LookbackOpt,["all"])
```

```
Price = 53.3720
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta        Gamma        Lambda     Vega      Theta       Rho
    _____    ________    ___________    ______    ______    _______    ________

    53.372    0.53372    -1.4211e-06       1      181.36    -8.7793    -213.01
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `timetable` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

HullWhite

Create `HullWhite` pricer object for `Cap`, `Floor`, or `Swaption` instrument using `HullWhite` model

Description

Create and price a `Cap`, `Floor`, or `Swaption` instrument object with a `HullWhite` model and a `HullWhite` pricing method using this workflow:

1. Use `fininstrument` to create a `Cap`, `Floor`, or `Swaption` instrument object.
2. Use `finmodel` to specify the `HullWhite` model for the `Cap`, `Floor`, or `Swaption` instrument object.
3. Use `finpricer` to specify the `HullWhite` pricer object for the `Cap`, `Floor`, or `Swaption` instrument object.

Note If you do not specify `ProjectionCurve` when you create a `Cap`, `Floor`, or `Swaption` instrument with the `HullWhite` pricer, the `ProjectionCurve` value defaults to the `DiscountCurve` value.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Cap`, `Floor`, or `Swaption` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
HullWhitePricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model)
```

Description

`HullWhitePricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model)` creates a `HullWhite` pricer object by specifying `PricerType` and the required name-value pair arguments `DiscountCurve` and `Model` to set properties on page 11-3272 using name-value pairs. For example, `HullWhitePricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',HWModel)` creates a `HullWhite` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: `char` | `string`

`HullWhite` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `HullWhitePricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',HWModel)`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a previously created `ratecurve` object.

Data Types: `object`

`Model` — Model
`HullWhite` model object

Model object, specified as the comma-separated pair consisting of `'Model'` and the name of the previously created `HullWhite` model object using `finmodel`.

Data Types: `object`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`HullWhite` model object

Model, returned as a `HullWhite` model object.

Data Types: `object`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Hull-White Pricer and Hull-White Model to Price a Floor Instrument

This example shows the workflow to price a `Floor` instrument when you use a `HullWhite` model and a `HullWhite` pricing method.

Create `Floor` Instrument Object

Use `fininstrument` to create a `Floor` instrument object.

```
FloorOpt = fininstrument("Floor",'Strike',0.02,'Maturity',datetime(2019,1,30),'Reset',4,'Principa
```

```
FloorOpt =
  Floor with properties:

                     Strike: 0.0200
                   Maturity: 30-Jan-2019
                ResetOffset: 0
                      Reset: 4
                      Basis: 8
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "floor_option"
```

Create `HullWhite` Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.032,'Sigma',0.04)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0320
    Sigma: 0.0400
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = "zero";
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: [10x1 datetime]
             Rates: [10x1 double]
            Settle: 15-Sep-2018
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
```

```
     LongExtrapMethod: "previous"
```

Create `HullWhite` Pricer Object

Use `finpricer` to create a `HullWhite` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',HullWhiteModel,'DiscountCurve',myRC)
```

```
outPricer =
  HullWhite with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.HullWhite]
```

Price `Floor` Instrument

Use `price` to compute the price for the `Floor` instrument.

```
Price = price(outPricer,FloorOpt)
```

```
Price = 0.5809
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

IRMonteCarlo

Create `IRMonteCarlo` pricer object for interest-rate instruments using `HullWhite`, `BraceGatarekMusiela`, `BlackKarasinski`, or `LinearGaussian2F` model

Description

Create and price a `Cap`, `Floor`, `Swap`, `Swaption`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object with a `HullWhite`, `BraceGatarekMusiela`, `SABRBraceGatarekMusiela`, `BlackKarasinski`, or `LinearGaussian2F` model and a `IRMonteCarlo` pricing method using this workflow:

1. Use `fininstrument` to create a `FixedBond`, `FloatBond`, `Cap`, `Floor`, `Swap`, `Swaption`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.
2. Use `finmodel` to specify a `HullWhite`, `BlackKarasinski`, or `LinearGaussian2F` model for the `Cap`, `Floor`, `Swap`, `Swaption`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.

   Use `finmodel` to specify a `BraceGatarekMusiela` or `SABRBraceGatarekMusiela` model for the `Cap`, `Floor`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.
3. When using a `HullWhite`, `BlackKarasinski`, or `LinearGaussian2F` model, use `finpricer` to specify an `IRMonteCarlo` pricer object for the `Cap`, `Floor`, `Swap`, `Swaption`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.

   When using a `BraceGatarekMusiela` or `SABRBraceGatarekMusiela` model, use `finpricer` to specify an `IRMonteCarlo` pricer object for the `Cap`, `Floor`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for `Cap`, `Floor`, `Swap`, `Swaption`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instruments, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
IRMonteCarloPricerObj = finpricer(PricerType,Model=model,
DiscountCurve=ratecurve_obj,SimulationDates=simulation_dates)
```

```
IRMonteCarloPricerObj = finpricer( ___ ,Name=Value)
```

Description

`IRMonteCarloPricerObj = finpricer(PricerType,Model=model, DiscountCurve=ratecurve_obj,SimulationDates=simulation_dates)` creates an `IRMonteCarlo` pricer object by specifying `PricerType` and sets the properties on page 11-3277 using the required name-value arguments `Model`, `DiscountCurve`, and `SimulationDates`.

`IRMonteCarloPricerObj = finpricer( ___ ,Name=Value)` sets optional properties on page 11-3277 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `IRMonteCarloPricerObj = finpricer("irmontecarlo",Model=HWModel,DiscountCurve=ratecurve_obj,SimulationDates=[datetime(2018,1,30); datetime(2019,1,30)],NumTrials=500)` creates an `IRMonteCarlo` pricer object using a `HullWhite` model. You can specify multiple name-value arguments.

Input Arguments

`PricerType` — Pricer type
string with value `"IRMonteCarlo"` | character vector with value `'IRMonteCarlo'`

Pricer type, specified as a string with the value `"IRMonteCarlo"` or a character vector with the value `'IRMonteCarlo'`.

Data Types: `char` | `string`

`IRMonteCarlo` Name-Value Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `IRMonteCarloPricerObj = finpricer("irmontecarlo",Model=HWModel,DiscountCurve=ratecurve_obj,SimulationDates=[datetime(2018,1,30); datetime(2019,1,30)],NumTrials=500)`

Required IRMonteCarlo Name-Value Arguments

`Model` — Model object
`HullWhite` object | `BlackKarasinski` object | `LinearGaussian2F` object | `BraceGatarekMusiela` object | `SABRBraceGatarekMusiela` object

Model object, specified as `Model` and the name of a previously created `HullWhite`, `BlackKarasinski`, `LinearGaussian2F`, `BraceGatarekMusiela`, or `SABRBraceGatarekMusiela` model object. Create the model object using `finmodel`.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as `DiscountCurve` and the name of a previously created `ratecurve` object.

Note Specify a flat `ratecurve` object for `DiscountCurve`. If you use a nonflat `ratecurve` object, the software uses the rate in the `ratecurve` object at `Maturity` and assumes that the value is constant for the life of the interest-rate option.

Data Types: `object`

`SimulationDates` — Simulation dates
datetime array | string array | date character vector

Simulation dates, specified as `SimulationDates` and a scalar or a vector using a datetime array, string array, or date character vectors.

To support existing code, `IRMonteCarlo` also accepts serial date numbers as inputs, but they are not recommended.

Optional IRMonteCarlo Name-Value Arguments

`NumTrials` — Simulation trials
`1000` (default) | scalar

Simulation trials, specified as `NumTrials` and a scalar number of independent sample paths.

Data Types: `double`

`RandomNumbers` — Dependent random variates
`[ ]` (default) | structure

Dependent random variates, specified as `RandomNumbers` and an `NSimulationDates`-by-`NBrownians`-by-`NTrials` 3D time series array. The 3D time series array has the following field:

- `Z` — `NSimulationDates`-by-`NBrownians`-by-`NTrials` 3D time series array of dependent random variates used to generate the Brownian motion vector (that is, Wiener processes) that drive the simulation.

Data Types: `struct`

Properties

`Model` — Model object
object

This property is read-only.

Model object, returned as an object.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`SimulationDates` — Simulation dates
datetime

Simulation dates, returned as a datetime array.

Data Types: `datetime`

`NumTrials` — Simulation trials
`1000` (default) | scalar

Simulation trials, returned as a scalar number of independent sample paths.

Data Types: `double`

`RandomNumbers` — Dependent random variates
`[ ]` (default) | structure

Dependent random variates, returned as an `NSimulationDates`-by-`NBrownians`-by-`NTrials` 3D time series array.

Data Types: `struct`

Object Functions

price Compute price for interest-rate instrument with IRMonteCarlo pricer

Examples

Use IRMonteCarlo Pricer and Hull-White Model to Price Fixed Bond Instrument

This example shows the workflow to price a `FixedBond` instrument when using a `HullWhite` model and an `IRMonteCarlo` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",Maturity=datetime(2022,9,15),CouponRate=0.05,Name="fixed_bond")
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond"
```

Create HullWhite Model Object

Use finmodel to create a HullWhite model object.

```
HullWhiteModel = finmodel("HullWhite",Alpha=0.32,Sigma=0.49)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.3200
    Sigma: 0.4900
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use finpricer to create an IRMonteCarlo pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",Model=HullWhiteModel,DiscountCurve=myRC,SimulationDates=Zero
```

```
outPricer =
  HWMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jan-2021    ...    ]
              Model: [1x1 finmodel.HullWhite]
```

Price FixedBond Instrument

Use price to compute the price for the FixedBond instrument.

```
[Price,outPR] = price(outPricer,FixB,["all"])

Price = 115.0303

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×4 table
    Price      Delta     Gamma     Vega
    _____    ________    _____    ____

    115.03    -397.13    1430.4     0
```

Use IRMonteCarlo Pricer and LinearGaussian2F Model to Price Cap Instrument

This example shows the workflow to price a `Cap` instrument when using a `LinearGaussian2F` model and an `IRMonteCarlo` pricing method.

Create Cap Instrument Object

Use `fininstrument` to create a `Cap` instrument object.

```
CapOpt = fininstrument("Cap",Maturity=datetime(2022,9,15),Strike=0.01,Reset=2,Name="cap_option")

CapOpt =
  Cap with properties:

                     Strike: 0.0100
                   Maturity: 15-Sep-2022
                ResetOffset: 0
                      Reset: 2
                      Basis: 0
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "cap_option"
```

Create LinearGaussian2F Model Object

Use `finmodel` to create a `LinearGaussian2F` model object.

```
LinearGaussian2FModel = finmodel("LinearGaussian2F",Alpha1=0.07,Sigma1=0.01,Alpha2=0.5,Sigma2=0.

LinearGaussian2FModel =
  LinearGaussian2F with properties:

         Alpha1: 0.0700
```

```
         Sigma1: 0.0100
         Alpha2: 0.5000
         Sigma2: 0.0060
    Correlation: -0.7000
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",Model=LinearGaussian2FModel,DiscountCurve=myRC,SimulationDa
```

```
outPricer =
  G2PPMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jan-2021    ...    ]
              Model: [1x1 finmodel.LinearGaussian2F]
```

Price Cap Instrument

Use `price` to compute the price and sensitivities for the `Cap` instrument.

```
[Price,outPR] = price(outPricer,CapOpt,["all"])
```

```
Price = 1.2156

outPR =
  priceresult with properties:
```

```
       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta     Gamma          Vega
    ______    ______    _____    _______________

    1.2156    131.37    11048    126.5    -157.38
```

Use IRMonteCarlo Pricer and BraceGatarekMusiela Model to Price Floor Instrument

This example shows the workflow to price a `Floor` instrument when using a `BraceGatarekMusiela` model and an `IRMonteCarlo` pricing method.

Create Floor Instrument Object

Use `fininstrument` to create a `Floor` instrument object.

```
FloorOpt = fininstrument("Floor",Maturity=datetime(2022,9,15),Strike=0.05,Reset=1,Name="floor_op
```

```
FloorOpt =
  Floor with properties:

                     Strike: 0.0500
                   Maturity: 15-Sep-2022
                ResetOffset: 0
                      Reset: 1
                      Basis: 0
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "floor_option"
```

Create BraceGatarekMusiela Model Object

Use `finmodel` to create a `BraceGatarekMusiela` model object.

```
BGMVolFunc = @(a,t) (a(1)*t + a(2)).*exp(-a(3)*t) + a(4);
BGMVolParams = [.3 -.02 .7 .14];
numRates = 20;
VolFunc(1:numRates-1) = {@(t) BGMVolFunc(BGMVolParams,t)};
Beta = .08;
CorrFunc = @(i,j,Beta) exp(-Beta*abs(i-j));
Correlation = CorrFunc(meshgrid(1:numRates-1)',meshgrid(1:numRates-1),Beta);
BGM = finmodel("BraceGatarekMusiela",Volatility=VolFunc,Correlation=Correlation,Period=1);
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [9x1 datetime]
               Rates: [9x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",Model=BGM,DiscountCurve=myRC,SimulationDates=ZeroDates)
```

```
outPricer =
  BGMMonteCarlo with properties:

         NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [01-Jul-2019    01-Jan-2020    01-Jan-2021    ...    ]
              Model: [1x1 finmodel.BraceGatarekMusiela]
```

Price Floor Instrument

Use `price` to compute the price and sensitivities for the `Floor` instrument.

```
[Price,outPR] = price(outPricer,FloorOpt,["all"])
```

```
Price = 14.7975

outPR =
  priceresult with properties:

       Results: [1x3 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×3 table
    Price      Delta     Gamma
    ______    _______    ______
```

```
    14.797    -398.43    1399.5
```

Use IRMonteCarlo Pricer and Black-Karasinski Model to Price Fixed Bond Instrument

This example shows the workflow to price a `FixedBond` instrument when using a `BlackKarasinski` model and an `IRMonteCarlo` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
FixB = fininstrument("FixedBond",Maturity=datetime(2022,9,15),CouponRate=0.05,Name="fixed_bond")
```

```
FixB =
  FixedBond with properties:

                 CouponRate: 0.0500
                     Period: 2
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2022
                        Name: "fixed_bond"
```

Create BlackKarasinski Model Object

Use `finmodel` to create a `BlackKarasinski` model object.

```
BlackKarasinskiModel = finmodel("BlackKarasinski",'Alpha',0.02,'Sigma',0.34)
```

```
BlackKarasinskiModel =
  BlackKarasinski with properties:

    Alpha: 0.0200
    Sigma: 0.3400
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,1,1);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
```

```
myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 01-Jan-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create IRMonteCarlo Pricer Object

Use `finpricer` to create an `IRMonteCarlo` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("IRMonteCarlo",Model=BlackKarasinskiModel,DiscountCurve=myRC,SimulationDate
```

```
outPricer =
  BKMonteCarlo with properties:

          NumTrials: 1000
      RandomNumbers: []
      DiscountCurve: [1x1 ratecurve]
    SimulationDates: [15-Mar-2019    15-Sep-2019    15-Mar-2020    ...    ]
              Model: [1x1 finmodel.BlackKarasinski]
```

Price FixedBond Instrument

Use `price` to compute the price for the `FixedBond` instrument.

```
[Price,outPR] = price(outPricer,FixB,["all"])
```

```
Price = 113.3046
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta     Gamma      Vega
    _____    _______    ______    ______
```

```
    113.3    -310.39    -26741    2.0294
```

Version History

Introduced in R2021b

Serial date numbers not recommended

Not recommended starting in R2022b

Although `IRMonteCarlo` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

Support for Black-Karasinski model

Behavior changed in R2022b

The name-value argument option for `Model` supports a `"BlackKarasinski"` binomial tree model.

See Also

Functions

`fininstrument` | `finmodel`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

IRTree

Create `IRTree` pricer object for `Cap`, `Floor`, `Swap`, `Swaption`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument

Description

Create and price a `Cap`, `Floor`, `Swap`, `Swaption`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object with a `HullWhite` or `BlackKarasinski` model and an `IRTree` pricing method using this workflow:

1. Use `fininstrument` to create a `Cap`, `Floor`, `Swaption`, `Swap`, `FloatBond`, `FixedBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.
2. Use `finmodel` to specify a `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` model for the `Cap`, `Floor`, `Swaption`, `Swap`, `FixedBond`, `FloatBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.
3. Use `finpricer` to specify an `IRTree` pricer object for a BK, BDT, or HW trinomial tree model for the `Cap`, `Floor`, `Swaption`, `Swap`, `FixedBond`, `FloatBond`, `FixedBondOption`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Cap`, `Floor`, `Swaption`, `Swap`, `FixedBond`, `FloatBond`, `FloatBondOption`, `OptionEmbeddedFixedBond`, or `OptionEmbeddedFloatBond` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
IRTreePricerObj = finpricer(PricerType,'Model',model_type,'
DiscountCurve',ratecurve_obj,'TreeDates',tree_dates)
```

Description

`IRTreePricerObj = finpricer(PricerType,'Model',model_type,'DiscountCurve',ratecurve_obj,'TreeDates',tree_dates)` creates a `IRTree` pricer object by specifying `PricerType` and the required name-value pair arguments for `Model`, `DiscountCurve`, and `TreeDates` to set properties on page 11-3289 using name-value pair arguments. For example, `IRTreePricerObj =`

`finpricer("IRTree",'Model',HullWhite,'DiscountCurve',ratecure_obj,'TreeDates',['jan-30-2018';'jan-30-2019'])` creates an `IRTree` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"IRTree"` | character vector with value `'IRTree'`

Pricer type, specified as a string with the value of `"IRTree"` or a character vector with the value of `'IRTree'`.

Data Types: `char` | `string`

`IRTree` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `IRTreePricerObj = finpricer("IRTree",'Model',HullWhite,'DiscountCurve',ratecure_obj,'TreeDates',['jan-30-2018';'jan-30-2019'])`

`Model` — Model type

`HullWhite` object | `BlackKarasinski` object | `BlackDermanToy` object

Model type, specified as the comma-separated pair consisting of `'Model'` and the name of a previously created `HullWhite`, `BlackKarasinski`, or `BlackDermanToy` model object. Create the model object using `finmodel`.

Note When you use a `HullWhite` model, the `IRTree` pricer uses the HW2000 algorithm [1].

Data Types: `object`

`DiscountCurve` — `ratecurve` object for creating `IRTree` and discounting cash flows

`ratecurve` object

This property is read-only.

`ratecurve` object for creating `IRTree` and discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a `ratecurve` object.

Data Types: `object`

`TreeDates` — Dates marking the cash flow dates of the tree

vector

Dates marking the cash flow dates of the tree, specified as the comma-separated pair consisting of `'TreeDates'` and an `NLEVELS`-by-`1` vector of dates. Cash flows with these dates will fall on tree nodes. The `TreeDates` argument determines the number of levels, or depth, of the tree. List dates in increasing order.

Data Types: `double` | `cell` | `datetime`

Properties

`Tree` — HW or BK trinomial tree

struct for HW or BK tree

HW or BK trinomial tree, returned as a struct with the following properties:

- `tObs` contains the time factor of each level of the tree.
- `dObs` contains the date of each level of the tree.
- `Probs` contains a cell array of 3-by-`N` numeric arrays with the up, mid, down probabilities of each node of the tree except for the last level. The cells in the cell array are ordered from root node. The arrays are 3-by-`N` with the first row corresponding to an up move, the mid row to a mid-move, and so on. Each column of the array represents a node starting from the top node of a given level.
- `CFlowT` is a cell array with as many elements as levels of the tree. Each cell array element contains the time factors (`tObs`) corresponding to its level of the tree and those levels ahead of it.
- `Probs` contains the probability arrays. Each element of the cell array contains the up, middle, and down transition probabilities for each node of the level.
- `Connect` contains a cell array with connectivity information for each node of the tree. The arrangement is similar to `Probs`, with the exception that it has only one row in each cell. The number represents the node in the next level to which the middle branch connects to. The top branch connects to the value above (minus one) and the lower branch connects to the value below (plus one).
- `FwdTree` contains the forward spot rate from one node to the next. The forward spot rate is defined as the inverse of the discount factor.
- `RateTree` contains the interest rate from one node to the next.

Data Types: `struct`

`TreeDates` — Tree dates

datetime

Tree dates, returned as a scalar datetime or datetime array.

Data Types: `datetime`

`Model` — Model type

`HullWhite` object | `BlackKarasinski` object | `BlackDermanToy` object

Model type, returned as an object.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for creating `IRTree` and discounting cash flows

`ratecurve` object

This property is read-only.

`ratecurve` object for creating the `IRTree` object and discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

Object Functions

price Compute price for interest-rate instrument with IRTree pricer

Examples

Use Hull-White Tree Pricer and Hull-White Model to Price FixedBondOption Instrument

This example shows the workflow to price a `FixedBondOption` instrument when you use a `HullWhite` model and an `IRTree` pricing method.

Create `FixedBond` Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",'Maturity',datetime(2029,9,15),'CouponRate',0.025,'Period',
```

```
BondInst =
  FixedBond with properties:

                 CouponRate: 0.0250
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2029
                        Name: "fixed_bond_instrument"
```

Create `FixedBondOption` Instrument Object

Use `fininstrument` to create a `FixedBondOption` instrument object.

```
FixedBOption = fininstrument("FixedBondOption",'ExerciseDate',datetime(2025,9,15),'Strike',98,'Bo
```

```
FixedBOption =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2025
           Strike: 98
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option_instrument"
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calyears([1:10])]';
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
HullWhiteModel = finmodel("HullWhite",'Alpha',0.01,'Sigma',0.05)
```

```
HullWhiteModel =
  HullWhite with properties:

    Alpha: 0.0100
    Sigma: 0.0500
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object with the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("irtree",'Model',HullWhiteModel,'DiscountCurve',myRC,'TreeDates',ZeroDa
```

```
HWTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [10x1 datetime]
           Model: [1x1 finmodel.HullWhite]
   DiscountCurve: [1x1 ratecurve]
```

```
HWTreePricer.Tree
```

```
ans = struct with fields:
        tObs: [0 1 1.9973 2.9945 3.9918 4.9918 5.9891 6.9863 7.9836 8.9836]
        dObs: [15-Sep-2019    15-Sep-2020    15-Sep-2021    ...    ]
      CFlowT: {1x10 cell}
       Probs: {1x9 cell}
     Connect: {1x9 cell}
```

```
     FwdTree: {1x10 cell}
    RateTree: {1x10 cell}
```

Price FixedBondOption Instrument

Use `price` to compute the price and sensitivities for the `FixedBondOption` instrument.

```
[Price, outPR] = price(HWTreePricer,FixedBOption,["all"])
```

```
Price = 11.1739

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma      Vega
    ______    _______    ______    ______

    11.174    -272.19    3667.6    243.09
```

Use Black-Karasinski Tree Pricer and Black-Karasinski Model to Price FixedBondOption Instrument

This example shows the workflow to price a `FixedBondOption` instrument when you use a `BlackKarasinski` model and an `IRTree` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",'Maturity',datetime(2029,9,15),'CouponRate',0.025,'Period',1
```

```
BondInst =
  FixedBond with properties:

                 CouponRate: 0.0250
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                  Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2029
```

```
                    Name: "fixed_bond_instrument"
```

Create FixedBondOption Instrument Object

Use `fininstrument` to create a `FixedBondOption` instrument object.

```
FixedBOption = fininstrument("FixedBondOption",'ExerciseDate',datetime(2025,9,15),'Strike',100,'
```

```
FixedBOption =
  FixedBondOption with properties:

       OptionType: "call"
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2025
           Strike: 100
             Bond: [1x1 fininstrument.FixedBond]
             Name: "fixed_bond_option"
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calyears([1:10])]';
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 0
             Dates: [10x1 datetime]
             Rates: [10x1 double]
            Settle: 15-Sep-2019
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackKarasinski Model Object

Use `finmodel` to create a `BlackKarasinski` model object.

```
BlackKarasinskiModel = finmodel("BlackKarasinski",'Alpha',0.02,'Sigma',0.34)
```

```
BlackKarasinskiModel =
  BlackKarasinski with properties:

    Alpha: 0.0200
    Sigma: 0.3400
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
BKTreePricer = finpricer("IRTree",'Model',BlackKarasinskiModel,'DiscountCurve',myRC,'TreeDates',
```

```
BKTreePricer =
  HWBKTree with properties:

             Tree: [1x1 struct]
        TreeDates: [10x1 datetime]
            Model: [1x1 finmodel.BlackKarasinski]
    DiscountCurve: [1x1 ratecurve]
```

```
BKTreePricer.Tree
```

```
ans = struct with fields:
        tObs: [0 1 1.9973 2.9945 3.9918 4.9918 5.9891 6.9863 7.9836 8.9836]
        dObs: [15-Sep-2019    15-Sep-2020    15-Sep-2021    ...    ]
      CFlowT: {1x10 cell}
       Probs: {1x9 cell}
     Connect: {1x9 cell}
     FwdTree: {1x10 cell}
    RateTree: {1x10 cell}
```

Price FixedBondOption Instrument

Use `price` to compute the price and sensitivities for the `FixedBondOption` instrument.

```
[Price, outPR] = price(BKTreePricer,FixedBOption,["all"])
```

```
Price = 0.5814
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
     Price      Delta     Gamma     Vega
    _______    _______    ______    _____

    0.58143    -15.842    45.702    2.793
```

Use Hull-White Tree Pricer and Hull-White Model to Price Vanilla FixedBond Instrument

This example shows the workflow to price a vanilla `FixedBond` instrument when you use a `HullWhite` model and an `IRTree` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object.

```
Maturity = datetime(2024,1,1);
Period = 1;
VBond = fininstrument("FixedBond",'Maturity', Maturity,'CouponRate', 0.025,'Period',Period)
```

```
VBond =
  FixedBond with properties:

                 CouponRate: 0.0250
                     Period: 1
                      Basis: 0
                EndMonthRule: 1
                   Principal: 100
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                    Holidays: NaT
                   IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 01-Jan-2024
                        Name: ""
```

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding);
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;

HWModel = finmodel("HullWhite",'alpha',AlphaCurve,'sigma',VolCurve);
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

            Tree: [1x1 struct]
       TreeDates: [10x1 datetime]
```

```
            Model: [1x1 finmodel.HullWhite]
    DiscountCurve: [1x1 ratecurve]
```

Price FixedBond Instrument

Use `price` to compute the price and sensitivities for the vanilla `FixedBond` instrument.

```
[Price, outPR] = price(HWTreePricer, VBond,["all"])
```

```
Price = 107.7023
```

```
outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price     Delta     Gamma     Vega
    _____    _______    ______    ____

    107.7    -602.56    4086.4     0
```

Use Hull-White Tree Pricer and Hull-White Model to Price Vanilla FloatBond Instrument

This example shows the workflow to price a vanilla `FloatBond` instrument when you use a `HullWhite` model and an `IRTree` pricing method.

Create ratecurve Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
ZeroTimes = calyears(1:10)';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
Compounding = 1;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates, "Compounding",Compounding);
```

Create FloatBond Instrument Object

Use `fininstrument` to create a vanilla `FloatBond` instrument object.

```
Spread = 0.03;
Reset = 1;
Maturity = datetime(2024,1,1);
Period = 1;
Float = fininstrument("FloatBond",'Maturity',Maturity,'Spread',Spread,'Reset',Reset,'ProjectionCu
```

```
Float =
  FloatBond with properties:
```

```
                    Spread: 0.0300
              ProjectionCurve: [1x1 ratecurve]
                  ResetOffset: 0
                        Reset: 1
                        Basis: 0
                 EndMonthRule: 1
                    Principal: 100
      DaycountAdjustedCashFlow: 0
         BusinessDayConvention: "actual"
            LatestFloatingRate: NaN
                     Holidays: NaT
                    IssueDate: NaT
              FirstCouponDate: NaT
               LastCouponDate: NaT
                    StartDate: NaT
                     Maturity: 01-Jan-2024
                         Name: ""
```

Create HullWhite Model Object

Use `finmodel` to create a `HullWhite` model object.

```
VolCurve = 0.01;
AlphaCurve = 0.1;

HWModel = finmodel("HullWhite",'alpha',AlphaCurve,'sigma',VolCurve);
```

Create IRTree Pricer Object

Use `finpricer` to create an `IRTree` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
HWTreePricer = finpricer("IRTree",'Model',HWModel,'DiscountCurve',ZeroCurve,'TreeDates',ZeroDates
```

```
HWTreePricer =
  HWBKTree with properties:

              Tree: [1x1 struct]
         TreeDates: [10x1 datetime]
             Model: [1x1 finmodel.HullWhite]
     DiscountCurve: [1x1 ratecurve]
```

Price FloatBond Instrument

Use `price` to compute the price and sensitivities for the vanilla `FloatBond` instrument.

```
[Price, outPR] = price(HWTreePricer,Float,["all"])
```

```
Price = 117.4686
```

```
outPR =
  priceresult with properties:

        Results: [1x4 table]
     PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×4 table
    Price      Delta     Gamma     Vega
    _____    ________    _____    ____

    117.47    -60.007    315.09     0
```

Use BlackDermanToy Tree Pricer and Black-Derman-Toy Model to Price FixedBondOption Instrument

This example shows the workflow to price a `FixedBondOption` instrument when you use a `BlackDermanToy` model and an `IRTree` pricing method.

Create FixedBond Instrument Object

Use `fininstrument` to create a `FixedBond` instrument object as the underlying bond.

```
BondInst = fininstrument("FixedBond",Maturity=datetime(2029,9,15),CouponRate=.024,Principal=100,I
```

```
BondInst =
  FixedBond with properties:

                 CouponRate: 0.0240
                     Period: 1
                      Basis: 1
                EndMonthRule: 1
                   Principal: 100
   DaycountAdjustedCashFlow: 0
      BusinessDayConvention: "actual"
                   Holidays: NaT
                  IssueDate: NaT
             FirstCouponDate: NaT
              LastCouponDate: NaT
                   StartDate: NaT
                    Maturity: 15-Sep-2029
                        Name: "fixed_bond"
```

Create FixedBondOption Instrument Object

Use `fininstrument` to create a `FixedBondOption` instrument object.

```
FixedBOption = fininstrument("FixedBondOption",ExerciseDate=datetime(2025,9,15),Strike=800,Bond=F
```

```
FixedBOption =
  FixedBondOption with properties:

      OptionType: "put"
   ExerciseStyle: "american"
    ExerciseDate: 15-Sep-2025
          Strike: 800
            Bond: [1x1 fininstrument.FixedBond]
            Name: "fixed_bond_option"
```

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2019,9,15);
Type = 'zero';
ZeroTimes = [calyears(1:10)]';
ZeroRates = [0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307 0.0310]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates,Basis=5)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 5
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2019
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create BlackDermanToy Model Object

Use finmodel to create a BlackDermanToy model object.

```
BlackDermanToyModel = finmodel("BlackDermanToy",Sigma=0.14)
```

```
BlackDermanToyModel =
  BlackDermanToy with properties:

    Sigma: 0.1400
```

Create IRTree Pricer Object

Use finpricer to create an IRTree pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
BKTreePricer = finpricer("IRTree",Model=BlackDermanToyModel,DiscountCurve=myRC,TreeDates=ZeroDate
```

```
BKTreePricer =
  BDTTree with properties:

            Tree: [1x1 struct]
       TreeDates: [10x1 datetime]
           Model: [1x1 finmodel.BlackDermanToy]
    DiscountCurve: [1x1 ratecurve]
```

```
BKTreePricer.Tree
```

```
ans = struct with fields:
        tObs: [0 1 2 3 4 5 6 7 8 9]
        dObs: [15-Sep-2019    15-Sep-2020    15-Sep-2021    ...    ]
```

```
     FwdTree: {1x10 cell}
    RateTree: {1x10 cell}
```

Price FixedBondOption Instrument

Use `price` to compute the price and sensitivities for the `FixedBondOption` instrument.

```
[Price, outPR] = price(BKTreePricer,FixedBOption,"all")

Price = 705.2729

outPR =
  priceresult with properties:

       Results: [1x4 table]
    PricerData: [1x1 struct]


outPR.Results

ans=1×4 table
    Price     Delta      Gamma        Vega
    ______    ______    _______    __________

    705.27    844.75    -8084.8    7.0486e-08
```

Version History

Introduced in R2020a

Support for Black-Derman-Toy model

Behavior changed in R2022b

The name-value argument option for `Model` supports a `"BlackDermanToy"` binomial tree model.

References

[1] Hull, John, and Alan White. “The General Hull–White Model and Supercalibration.” *Financial Analysts Journal*, vol. 57, no. 6, Nov. 2001, pp. 34–43.

See Also

Functions

`fininstrument` | `finmodel`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

AssetTree

Create `AssetTree` pricer object for `Vanilla`, `Barrier`, `Asian`, or `Lookback` instrument

Description

Create and price a `Vanilla`, `Barrier`, `Asian`, or `Lookback` instrument object with a `BlackScholes` model and an `AssetTree` pricing method using this workflow:

1. Use `fininstrument` to create a `Vanilla`, `Lookback`, `Barrier`, or `Asian` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Vanilla`, `Barrier`, `Asian`, or `Lookback` instrument object.
3. Use `finpricer` to specify an `AssetTree` pricer object for a Cox-Ross-Rubinstein (CRR), equal-probability (EQP), Leisen-Reimer (LR), or Standard Trinomial (ST) lattice tree model for the `Vanilla`, `Barrier`, `Asian`, or `Lookback` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Vanilla`, `Barrier`, `Asian`, or `Lookback` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
AssetTreePricerObj = finpricer(PricerType,'Model',model_type,'
DiscountCurve',ratecurve_obj,'SpotPrice',spot_price)
AssetTreePricerObj = finpricer(___,Name,Value)
```

Description

`AssetTreePricerObj = finpricer(PricerType,'Model',model_type,'DiscountCurve',ratecurve_obj,'SpotPrice',spot_price)` creates an `AssetTree` pricer object by specifying `PricerType` and the required name-value pair arguments for `Model`, `DiscountCurve`, and `SpotPrice`.

`AssetTreePricerObj = finpricer(___,Name,Value)` sets optional properties on page 11-3304 using additional name-value pair arguments in addition to the required arguments in the previous syntax. For example, `AssetTreePricerObj = finpricer("AssetTree",'Model',BlackScholes,'DiscountCurve',ratecure_obj,'SpotPrice',1000)` creates an `AssetTree` pricer object.

Input Arguments

`PricerType` — Pricer type
string with value `"AssetTree"` | character vector with value `'AssetTree'`

Pricer type, specified as a string with the value of "AssetTree" or a character vector with the value of 'AssetTree'.

Data Types: char | string

AssetTree Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: AssetTreePricerObj = finpricer("AssetTree",'Model',BlackScholes,'DiscountCurve',ratecure_obj,'Spot Price',1000)

Model — Model
BlackScholes object

Model, specified as the comma-separated pair consisting of 'Model' and the name of the previously created BlackScholes model object using finmodel.

Data Types: object

DiscountCurve — ratecurve object for creating AssetTree object and discounting cash flows
ratecurve object

This property is read-only.

ratecurve object for creating the AssetTree object and discounting cash flows, specified as the comma-separated pair consisting of 'DiscountCurve' and the name of a ratecurve object.

Data Types: object

SpotPrice — Underlying spot price
scalar numeric

Underlying spot price, specified as the comma-separated pair consisting of 'SpotPrice' and a scalar numeric.

Data Types: double

Optional AssetTree Name-Value Pair Arguments

PricingMethod — Asset pricing method
"CoxRossRubinstein" (default) | string with value "CoxRossRubinstein", "EqualProbability", "LeisenReimer", or "StandardTrinomial" | character vector with value 'CoxRossRubinstein', 'EqualProbability', 'LeisenReimer', or "StandardTrinomial"

Asset pricing method, specified as the comma-separated pair consisting of 'PricingMethod' and a string or character vector.

Data Types: char | string

Maturity — Maturity date
datetime scalar | string scalar | date character vector

Maturity date, specified as the comma-separated pair consisting of `'Maturity'` and a scalar datetime, string, or date character vector.

To support existing code, `AssetTree` also accepts serial date numbers as inputs, but they are not recommended.

If you use a date character vector or string, the format must be recognizable by `datetime` because the `Maturity` property is stored as a datetime.

`NumPeriods` — Number of levels or time steps of the tree
`10` (default) | numeric

Number of levels or time steps of the tree, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar.

Data Types: `double`

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"cash"` or `"continuous"` | character vector with value `'cash'` or `'continuous'`

Stock dividend type, specified as the comma-separated pair consisting of `'DividendType'` and a string or character vector. `DividendType` must be `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `char` | `string`

`DividendValue` — Dividend amount or dividend schedule for underlying stock
`0` (default) | scalar numeric | timetable

Dividend amount or dividend schedule for the underlying stock, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar numeric for a dividend amount or a timetable for a dividend schedule.

Note `DividendValue` must be a scalar for a `"continuous"` `DividendType` or a timetable for `"cash"` `DividendType`.

Data Types: `double` | `timetable`

`Strike` — Option strike used with Leisen-Reimer pricing method
`SpotPrice` (default) | nonnegative numeric

Option strike used with the Leisen-Reimer pricing method, specified as the comma-separated pair consisting of `'Strike'` and a scalar nonnegative numeric.

Data Types: `double`

`InversionMethod` — Inversion method for Leisen-Reimer pricing method
`'PP1'` (default) | string with value `"PP1"` or `"PP2"` | character vector with value `'PP1'` or `'PP2'`

Inversion method for the Leisen-Reimer pricing method, specified as the comma-separated pair consisting of `'InversionMethod'` and a string or character vector.

- `'PP1'` — Peizer-Pratt method 1 inversion

- `'PP2'` — Peizer-Pratt method 2 inversion

Data Types: `string` | `char`

Properties

`InversionMethod` — Inversion method for Leisen-Reimer pricing method

`'PP1'` (default) | string with value `"PP1"` or `"PP2"`

Inversion method for the Leisen-Reimer pricing method, returned as a string.

Data Types: `string`

`Strike` — Option strike used with Leisen-Reimer pricing method

`SpotPrice` (default) | nonnegative numeric

Option strike used with the Leisen-Reimer pricing method, returned as a nonnegative numeric.

Data Types: `double`

`Tree` — CRR, EQP, LR binomial tree or STT trinomial tree

structure

CRR, EQP, LR binomial tree or STT trinomial tree, returned as a structure with the following properties:

- `Probs` contains a 2-by-`NumLevels` numeric array with the up and down probabilities that apply to each level of the tree except for the last one. All nodes in a given level share the same up and down probabilities. The columns of the `Probs` array are ordered from the root node. The first row of the array corresponds to the probability of an up move, while the second row corresponds to a down move.
- `ATree` contains the price tree for the underlying asset.
- `dObs` contains the date of each level of the tree.
- `tObs` contains the time factor of each level of the tree.

Data Types: `struct`

`NumPeriods` — Number of levels or time steps of the tree

numeric

Number of levels or time steps of the tree, returned as a numeric.

Data Types: `datetime`

`Model` — Model type

object

Model type, returned as an object.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for creating `AssetTree` object and discounting cash flows

`ratecurve` object

This property is read-only.

`ratecurve` object for creating the `AssetTree` object and discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"cash"` or `"continuous"`

This property is read-only.

Stock dividend type, returned as a string. `DividendType` is either `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `string`

`DividendValue` — Dividend amount or dividend schedule for underlying stock
`0` (default) | scalar nonnegative numeric | timetable

Dividend amount or dividend schedule for the underlying stock, returned as a scalar nonnegative numeric for a dividend yield or a timetable for a dividend schedule.

Data Types: `double` | `timetable`

`TreeDates` — Tree dates
datetime

Tree dates, returned as a scalar datetime or datetime array.

Data Types: `datetime`

Object Functions

price Compute price for equity instrument with AssetTree pricer

Examples

Use Leisen-Reimer Tree Pricer and Black-Scholes Model to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2019,5,1),'Strike',29,'OptionType','
```

```
VanillaOpt =
  Vanilla with properties:
```

```
       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 01-May-2019
           Strike: 29
             Name: "vanilla_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2500
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2020,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2020
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object for an LR equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
LRPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',30,'
```

```
LRPricer =
  LRTree with properties:

    InversionMethod: PP1
             Strike: 30
               Tree: [1x1 struct]
        NumPeriods: 15
              Model: [1x1 finmodel.BlackScholes]
      DiscountCurve: [1x1 ratecurve]
```

```
          SpotPrice: 30
       DividendType: "continuous"
      DividendValue: 0
          TreeDates: [02-Feb-2018 08:00:00    06-Mar-2018 16:00:00    ...    ]
```

Price `Vanilla` Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(LRPricer,VanillaOpt,"all")
```

```
Price = 2.2542

outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma       Vega     Lambda      Rho       Theta
    ______    ________    ________    ______    ______    _______    ________

    2.2542    -0.33628    0.044039    12.724    -4.469    -16.433    -0.76073
```

Use Standard Trinomial Tree Pricer and Black-Scholes Model to Price Vanilla Instrument

This example shows the workflow to price a `Vanilla` instrument when you use a `BlackScholes` model and an `AssetTree` pricing method for a Standard Trinomial (STT) tree.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2019,5,1),'Strike',29,'OptionType','
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 01-May-2019
           Strike: 29
             Name: "vanilla_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.25)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.2500
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2020,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 1
               Dates: 01-Jan-2020
               Rates: 0.0350
              Settle: 01-Jan-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create AssetTree Pricer Object

Use `finpricer` to create an `AssetTree` pricer object for an Standard Trinomial equity tree and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
STTPricer = finpricer("AssetTree",'DiscountCurve',myRC,'Model',BlackScholesModel,'SpotPrice',30,
```

```
STTPricer =
  STTree with properties:

            Tree: [1x1 struct]
       NumPeriods: 15
            Model: [1x1 finmodel.BlackScholes]
    DiscountCurve: [1x1 ratecurve]
        SpotPrice: 30
     DividendType: "continuous"
    DividendValue: 0
        TreeDates: [02-Feb-2018 08:00:00    06-Mar-2018 16:00:00    ...    ]
```

Price Vanilla Instrument

Use `price` to compute the price and sensitivities for the `Vanilla` instrument.

```
[Price, outPR] = price(STTPricer,VanillaOpt,"all")
```

```
Price = 2.2826
```

```
outPR =
  priceresult with properties:
```

```
       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Vega     Lambda       Rho       Theta
    _____    ________    _________    _____    _______    _______    _________

    2.2826    -0.2592    0.030949    12.51    -3.8981    -16.516    -0.73845
```

Version History

Introduced in R2021a

Support for Standard Trinomial Tree model

The name-value argument option for `PricingMethod` supports a `"StandardTrinomial"` lattice tree model.

Serial date numbers not recommended

Not recommended starting in R2022b

Although `AssetTree` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Hull, John, and Alan White. “The General Hull–White Model and Supercalibration.” *Financial Analysts Journal*, 57, no. 6, (November 2001): 34–43.

See Also

Functions

`fininstrument` | `finmodel`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

KemnaVorst

Create `KemnaVorst` pricer object for `Asian` instrument using `BlackScholes` model

Description

Create and price a `Asian` instrument object with a `BlackScholes` model and a `KemnaVorst` pricing method using this workflow:

1. Use `fininstrument` to create an `Asian` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Asian` instrument object.
3. Use `finpricer` to specify a `KemnaVorst` pricer object for the `Asian` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for an `Asian` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
KemnaVorstPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model,'SpotPrice',spotprice_value)
KemnaVorstPricerObj = finpricer(___,Name,Value)
```

Description

`KemnaVorstPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model,'SpotPrice',spotprice_value)` creates a `KemnaVorst` pricer object by specifying `PricerType` and sets the properties on page 11-3313 for the required name-value pair arguments `DiscountCurve`, `Model`, and `SpotPrice`.

`KemnaVorstPricerObj = finpricer(___,Name,Value)` to set optional properties on page 11-3313 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `KemnaVorstPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',100,'PricingMethod',"KemnaVorst")` creates a `KemnaVorst` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: char | string

KemnaVorst Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: KemnaVorstPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',100,'PricingMethod',"KemnaVorst")

Required KemnaVorst Name-Value Pair Arguments

DiscountCurve — ratecurve object for discounting cash flows
ratecurve object

ratecurve object for discounting cash flows, specified as the comma-separated pair consisting of 'DiscountCurve' and the name of the previously created ratecurve object.

Note Specify a flat ratecurve object for DiscountCurve. If you use a nonflat ratecurve object, the software uses the rate in the ratecurve object at Maturity and assumes that the value is constant for the life of the equity option.

Data Types: object

Model — Model
BlackScholes model object

Model, specified as the comma-separated pair consisting of 'Model' and the name of a previously created BlackScholes model object using finmodel.

Data Types: object

SpotPrice — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of 'SpotPrice' and a scalar nonnegative numeric.

Data Types: double

Optional KemnaVorst Name-Value Pair Arguments

DividendType — Stock dividend type
"continuous" (default) | string with value "cash" or "continuous" | character vector with value 'cash' or 'continuous'

Stock dividend type, specified as the comma-separated pair consisting of 'DividendType' and a string or character vector. DividendType must be "cash" for actual dollar dividends or "continuous" for a continuous dividend yield.

Data Types: char | string

`DividendValue` — Dividend amount or dividend schedule for underlying stock
`0` (default) | scalar numeric | timetable

Dividend amount for the underlying stock, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar numeric for a dividend amount or a timetable for a dividend schedule.

Note `DividendValue` must be a scalar for a `"continuous"` `DividendType` or a timetable for `"cash"` `DividendType`.

Data Types: `double` | `timetable`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"KemnaVorst"` | character vector with value `'KemnaVorst'`

Analytic pricing method, specified as the comma-separated pair consisting of `'PricingMethod'` and a string or character vector.

Note The default pricing method for a `BlackScholes` model is a `BlackScholes` pricer.

Data Types: `double`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
object

This property is read-only.

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendType` — Stock dividend type
`"continuous"` (default) | string with value `"cash"` or `"continuous"`

This property is read-only.

Stock dividend type, returned as a string. `DividendType` is either `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `string`

`DividendValue` — Dividend amount or dividend schedule for underlying stock
`0` (default) | scalar nonnegative numeric | timetable

Dividend amount or dividend schedule for the underlying stock, returned as a scalar numeric for a dividend yield or a timetable for a dividend schedule.

Data Types: `double` | `timetable`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"KemnaVorst"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Kemna-Vorst Pricer and Black-Scholes Model to Price Asian Instrument

This example shows the workflow to price an `Asian` instrument when you use a `BlackScholes` model and a `KemnaVorst` pricing method.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',105,'OptionType',"pu
```

```
AsianOpt =
  Asian with properties:

         OptionType: "put"
             Strike: 105
        AverageType: "geometric"
       AveragePrice: 0
   AverageStartDate: NaT
      ExerciseStyle: "european"
       ExerciseDate: 15-Sep-2022
               Name: "asian_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.32)
```

```
BlackScholesModel =
  BlackScholes with properties:
```

```
     Volatility: 0.3200
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create KemnaVorst Pricer Object

Use `finpricer` to create a `KemnaVorst` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',100,
```

```
outPricer =
  KemnaVorst with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0.0500
     DividendType: "continuous"
```

Price Asian Instrument

Use `price` to compute the price and sensitivities for the `Asian` instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 18.1186
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta        Gamma       Lambda     Vega       Rho        Theta
    ______    ________    _________    ______    ______    _______    _______

    18.119    -0.44689    0.0087391    -3.025    64.582    -251.23    -1.5738
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `timetable` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Kirk

Create `Kirk` pricer object for `Spread` instrument using `BlackScholes` model

Description

Create and price a `Spread` instrument object with a `BlackScholes` model and a `Kirk` pricing method using this workflow:

1. Use `fininstrument` to create a `Spread` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Spread` instrument object.
3. Use `finpricer` to specify a `Kirk` pricer object for the `Spread` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Spread` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
KirkPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model,'SpotPrice',spotprice_value)
KirkPricerObj = finpricer(___,Name,Value)
```

Description

`KirkPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model,'SpotPrice',spotprice_value)` creates a `Kirk` pricer object by specifying `PricerType` and sets the properties on page 11-3319 for the required name-value pair arguments `DiscountCurve`, `Model`, and `SpotPrice`.

`KirkPricerObj = finpricer(___,Name,Value)` to set optional properties on page 11-3319 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `KirkPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',[100;105],'DividendValue',[2.5,2.8],'PricingMethod',"Kirk")` creates a `Kirk` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: char | string

Kirk Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: KirkPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',[100;105],'DividendValue',[2.5,2.8],'PricingMethod',"Kirk")

Required Kirk Name-Value Pair Arguments

DiscountCurve — ratecurve object for discounting cash flows
ratecurve object

ratecurve object for discounting cash flows, specified as the comma-separated pair consisting of 'DiscountCurve' and the name of a previously created ratecurve object.

Note Specify a flat ratecurve object for DiscountCurve. If you use a nonflat ratecurve object, the software uses the rate in the ratecurve object at Maturity and assumes that the value is constant for the life of the equity option.

Data Types: object

Model — Model
BlackScholes model object

Model object, specified as the comma-separated pair consisting of 'Model' and the name of a previously created BlackScholes model object using finmodel.

Data Types: object

SpotPrice — Current price of underlying asset
scalar nonnegative numeric | cell array of nonnegative numerics

Current price of the underlying asset, specified as the comma-separated pair consisting of 'SpotPrice' and a scalar or cell array of nonnegative numerics.

Data Types: double

Optional Kirk Name-Value Pair Arguments

DividendType — Dividend type
"continuous" (default) | string with value "continuous" | character vector with value 'continuous'

Dividend type, specified as the comma-separated pair consisting of 'DividendType' and a string or character vector for a continuous dividend yield.

Data Types: char | string

`DividendValue` — Dividend yield for underlying asset
`[0,0]` (default) | scalar nonnegative numeric | vector of nonnegative numerics

Dividend yield for the underlying asset, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar or a vector of nonnegative numerics. Use a vector of nonnegative values for `DividendValue` when pricing a `Spread` instrument.

Data Types: `double`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"Kirk"` | character vector with value `'Kirk'`

Analytic pricing method, specified as the comma-separated pair consisting of `'PricingMethod'` and a string or character vector.

Note The default pricing method for a `BlackScholes` model is a `BlackScholes` pricer.

Data Types: `double`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

This property is read-only.

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
required input (default) | scalar nonnegative numeric | vector of nonnegative numerics

Current price of the underlying asset, returned as a scalar or vector of nonnegative numeric values.

Data Types: `double`

`DividendType` — Dividend type
`"continuous"` (default) | string with value `"continuous"`

This property is read-only.

Dividend type, returned as a string.

Data Types: `string`

`DividendValue` — Dividend yield for underlying stock
`[0,0]` (default) | scalar nonnegative numeric | vector of nonnegative numerics

Dividend yield for the underlying stock, returned as a scalar or vector of nonnegative numerics.

Data Types: `double`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"Kirk"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Kirk Pricer and Black-Scholes Model to Price Spread Instrument

This example shows the workflow to price a `Spread` instrument when you use a `BlackScholes` model and a `Kirk` pricing method.

Create `Spread` Instrument Object

Use `fininstrument` to create a `Spread` instrument object.

```
SpreadOpt = fininstrument("Spread",'Strike',5,'ExerciseDate',datetime(2021,9,15),'OptionType',"pu
```

```
SpreadOpt =
  Spread with properties:

       OptionType: "put"
           Strike: 5
    ExerciseStyle: "european"
     ExerciseDate: 15-Sep-2021
             Name: "spread_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',[0.2 , 0.1])
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: [0.2000 0.1000]
    Correlation: [2x2 double]
```

Create `ratecurve` Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
```

```
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Kirk Pricer Object

Use `finpricer` to create a `Kirk` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',[103
```

```
outPricer =
  Kirk with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: [103 105]
    DividendValue: [0.0250 0.0280]
     DividendType: "continuous"
```

Price Spread Instrument

Use `price` to compute the price and sensitivities for the `Spread` instrument.

```
[Price, outPR] = price(outPricer,SpreadOpt,["all"])
```

```
Price = 17.8614
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price            Delta                   Gamma                   Lambda                Vega
    _____    ____________________    ____________________    ________________    ____________
```

```
    17.861    -0.44663     0.58229    0.0093493     0.008195    -2.5756     3.3578    59.518    2
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `timetable` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Levy

Create `Levy` pricer object for `Asian` instrument using `BlackScholes` model

Description

Create and price a `Asian` instrument object with a `BlackScholes` model and a `Levy` pricing method using this workflow:

1. Use `fininstrument` to create an `Asian` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Asian` instrument object.
3. Use `finpricer` to specify a `Levy` pricer object for the `Asian` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for an `Asian` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
LevyPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model,'SpotPrice',spotprice_value)
LevyPricerObj = finpricer(___,Name,Value)
```

Description

`LevyPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model,'SpotPrice',spotprice_value)` creates a `Levy` pricer object by specifying `PricerType` and sets the properties on page 11-3325 for the required name-value pair arguments `DiscountCurve`, `Model`, and `SpotPrice`.

`LevyPricerObj = finpricer(___,Name,Value)` to set optional properties on page 11-3325 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `LevyPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',100,'PricingMethod',"Levy")` creates a `Levy` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: char | string

Levy Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: LevyPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',100,'PricingMethod',"Levy")

Required Levy Name-Value Pair Arguments

DiscountCurve — ratecurve object for discounting cash flows
ratecurve object

ratecurve object for discounting cash flows, specified as the comma-separated pair consisting of 'DiscountCurve' and the name of a previously created ratecurve object.

Note Specify a flat ratecurve object for DiscountCurve. If you use a nonflat ratecurve object, the software uses the rate in the ratecurve object at Maturity and assumes that the value is constant for the life of the equity option.

Data Types: object

Model — Model
BlackScholes model object

Model, specified as the comma-separated pair consisting of 'Model' and the name of a previously created BlackScholes model object using finmodel.

Data Types: object

SpotPrice — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of 'SpotPrice' and a scalar nonnegative numeric.

Data Types: double

Optional Levy Name-Value Pair Arguments

DividendType — Dividend type
"continuous" (default) | string with value of "continuous" | character vector with value of 'continuous'

Dividend type, specified as the comma-separated pair consisting of 'DividendType' and a string or character vector for a continuous dividend yield.

Data Types: char | string

`DividendValue` — Dividend yield or dividend schedule for underlying stock
`0` (default) | scalar numeric | timetable

Dividend yield for the underlying stock, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar numeric for a dividend yield or a timetable for a dividend schedule.

Note Specify a scalar if `DividendType` is `"continuous"` and a timetable if `DividendType` is `"cash"`.

Data Types: `double` | `timetable`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"Levy"` | character vector with value `'Levy'`

Analytic pricing method, specified as the comma-separated pair consisting of `'PricingMethod'` and a character vector or string.

Note The default pricing method for a `BlackScholes` model is a `BlackScholes` pricer.

Data Types: `double`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendType` — Dividend type
`"continuous"` (default) | string with value of `"continuous"`

This property is read-only.

Dividend type, returned as a string.

Data Types: `string`

`DividendValue` — Dividend yield or dividend schedule for underlying stock
`0` (default) | scalar nonnegative numeric | timetable

Dividend yield or dividend schedule for the underlying stock, returned as a scalar numeric for a dividend yield or a timetable for a dividend schedule.

Data Types: `double` | `timetable`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"Levy"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Levy Pricer and Black-Scholes Model to Price Asian Instrument

This example shows the workflow to price an `Asian` instrument when you use a `BlackScholes` model and a `Levy` pricing method.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',105,'OptionType',"pu
```

```
AsianOpt =
  Asian with properties:

          OptionType: "put"
              Strike: 105
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.32)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3200
    Correlation: 1
```

Create ratecurve Object

Create a flat ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Levy Pricer Object

Use finpricer to create a Levy pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',100,
```

```
outPricer =
  Levy with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0
     DividendType: "continuous"
```

Price Asian Instrument

Use price to compute the price and sensitivities for the Asian instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 13.0014
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta       Gamma      Lambda      Vega      Theta        Rho
```

```
    ______    _______    _________    _______    ______    ________    _______

    13.001    -0.3749    0.0094403    -2.8836    44.586    -0.71607    -121.97
```

Use Levy Pricer and Black-Scholes Model to Price Asian Instrument for Foreign Exchange

This example shows the workflow to price an `Asian` instrument for an arithmetic average currency option when you use a `BlackScholes` model and a `Levy` pricing method. Assume that the current exchange rate is $0.52 and has a volatility of 12% per annum. The annualized continuously compounded foreign risk-free rate is 8% per annum.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',.50,'OptionType',"pu
```

```
AsianOpt =
  Asian with properties:

          OptionType: "put"
              Strike: 0.5000
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_fx_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
Sigma = .12;
BlackScholesModel = finmodel("BlackScholes",'Volatility',Sigma)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.1200
    Correlation: 1
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Levy Pricer Object

Use `finpricer` to create a `Levy` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument. When you price currencies using a Asian instrument for an arithmetic average currency option, `DividendType` must be `"continuous"` and `DividendValue` is the annualized risk-free interest rate in the foreign country.

```
ForeignRate = 0.08;
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',.52,
```

```
outPricer =
  Levy with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 0.5200
    DividendValue: 0.0800
     DividendType: "continuous"
```

Price Asian FX Instrument

Use `price` to compute the price and sensitivities for the `Asian` FX instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 0.0535
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
     Price       Delta       Gamma      Lambda       Vega        Theta         Rho
    ________    ________    _______    ________    ________    _________    ________
```

```
    0.053516    -0.62792    3.8371    -6.1014    0.15613    -0.010917    -0.82694
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `timetable` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

Normal

Create `Normal` pricer object for `Cap`, `Floor`, or `Swaption` instrument using `Normal` model

Description

Create and price a `Cap`, `Floor`, or `Swaption` instrument object with a `Normal` model and a `Normal` pricing method using this workflow:

1. Use `fininstrument` to create a `Cap`, `Floor`, or `Swaption` instrument object.
2. Use `finmodel` to specify a `Normal` model for the `Cap`, `Floor`, or `Swaption` instrument object.
3. Use `finpricer` to specify a `Normal` pricer object for the `Cap`, `Floor`, or `Swaption` instrument object.

> **Note** If you do not specify `ProjectionCurve` when you create a `Cap`, `Floor`, or `Swaption` instrument with the `HullWhite` pricer, the `ProjectionCurve` value defaults to the `DiscountCurve` value.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Cap`, `Floor`, or `Swaption` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
NormalPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model)
```

Description

`NormalPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model)` creates a `Normal` pricer object by specifying `PricerType` and the required name-value pair arguments `DiscountCurve` and `Model` to set properties on page 11-3332 using name-value pair arguments. For example, `NormalPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',NormModel)` creates a `Normal` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: `char` | `string`

`Normal` Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `NormalPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',NormModel)`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a previously created `ratecurve` object.

Data Types: `object`

`Model` — Model
`Normal` model object

Model, specified as the comma-separated pair consisting of `'Model'` and the name of a previously created `Normal` model object using `finmodel`.

Data Types: `object`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`Normal` model object

Model, returned as a `Normal` model object.

Data Types: `object`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Normal Pricer and Normal Model to Price Cap Instrument

This example shows the workflow to price a `Cap` instrument when you use a `Normal` model and a `Normal` pricing method.

Create Cap Instrument Object

Use `fininstrument` to create a `Cap` instrument object.

```
CapOpt = fininstrument("Cap",'Strike',0.02,'Maturity',datetime(2019,6,25),'Reset',4,'Principal',1
```

```
CapOpt =
  Cap with properties:

                     Strike: 0.0200
                   Maturity: 25-Jun-2019
                ResetOffset: 0
                      Reset: 4
                      Basis: 8
                  Principal: 100
             ProjectionCurve: [0x0 ratecurve]
    DaycountAdjustedCashFlow: 0
       BusinessDayConvention: "actual"
                   Holidays: NaT
                       Name: "cap_option"
```

Create `Normal` Model Object

Use `finmodel` to create a `Normal` model object.

```
NormalModel = finmodel("Normal",'Volatility',0.063)
```

```
NormalModel =
  Normal with properties:

    Volatility: 0.0630
```

Create `ratecurve` Object

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
   ShortExtrapMethod: "next"
    LongExtrapMethod: "previous"
```

Create Normal Pricer Object

Use finpricer to create a Normal pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("analytic",'Model',NormalModel,'DiscountCurve',myRC)
```

```
outPricer =
  Normal with properties:

    DiscountCurve: [1x1 ratecurve]
            Shift: 0
            Model: [1x1 finmodel.Normal]
```

Price Cap Instrument

Use price to compute the price for the Cap instrument.

```
Price = price(outPricer,CapOpt)
```

```
Price = 0.4828
```

Version History

Introduced in R2020a

See Also

Functions

fininstrument | finmodel | ratecurve

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53
“Work with Negative Interest Rates Using Objects” on page 2-22

RollGeskeWhaley

Create `RollGeskeWhaley` pricer object for American exercise `Vanilla` instrument using `BlackScholes` model

Description

Create and price a `Vanilla` instrument object with a `BlackScholes` model and a `RollGeskeWhaley` pricing method using this workflow:

1. Use `fininstrument` to create a `Vanilla` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Vanilla` instrument object.
3. Use `finpricer` to specify a `RollGeskeWhaley` pricer object for the `Vanilla` instrument object (American exercise).

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a `Vanilla` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
RollGeskeWhaleyPricerObj = finpricer(PricerType,'Model',model,'
DiscountCurve',ratecurve_obj,'SpotPrice',spotrate_value,'
DividendType',dividendtype,'DividendValue',dividendvalue)
RollGeskeWhaleyPricerObj = finpricer(___,Name,Value)
```

Description

`RollGeskeWhaleyPricerObj = finpricer(PricerType,'Model',model,'DiscountCurve',ratecurve_obj,'SpotPrice',spotrate_value,'DividendType',dividendtype,'DividendValue',dividendvalue)` creates a `RollGeskeWhaley` pricer object by specifying `PricerType` and sets the properties on page 11-3337 for the required name-value pair arguments `Model`, `DiscountCurve`, and `SpotPrice`.

`RollGeskeWhaleyPricerObj = finpricer(___,Name,Value)` to set optional properties on page 11-3337 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `RollGeskeWhaleyPricerObj = finpricer("Analytic",'Model',BSModel,'DiscountCurve',ratecurve_obj,'SpotPrice',1000,'DividendValue',timetable(datetime(2021,6,15),2.5),'DividendType',"cash",'PricingMethod',"RollGeskeWhaley")` creates a `RollGeskeWhaley` pricer object. You can specify multiple name-value pair arguments.

Input Arguments

PricerType — Pricer type

string with value "Analytic" | character vector with value 'Analytic'

Pricer type, specified as a string with the value of "Analytic" or a character vector with the value of 'Analytic'.

Data Types: char | string

RollGeskeWhaley Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: RollGeskeWhaleyPricerObj = finpricer("Analytic",'Model',BSModel,'DiscountCurve',ratecurve_obj,'SpotPrice',1000,'DividendValue',timetable(datetime(2021,6,15),2.5),'DividendType',"cash",'PricingMethod',"RollGeskeWhaley")

Required RollGeskeWhaley Name-Value Pair Arguments

Model — Model

BlackScholes model object

Model, specified as the comma-separated pair consisting of 'Model' and the name of a previously created BlackScholes model object using finmodel.

Data Types: object

DiscountCurve — ratecurve object for discounting cash flows

ratecurve object

ratecurve object for discounting cash flows, specified as the comma-separated pair consisting of 'DiscountCurve' and the name of a previously created ratecurve object.

Note Specify a flat ratecurve object for DiscountCurve. If you use a nonflat ratecurve object, the software uses the rate in the ratecurve object at Maturity and assumes that the value is constant for the life of the equity option.

Data Types: object

SpotPrice — Current price of underlying asset

nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of 'SpotPrice' and a scalar nonnegative numeric.

Data Types: double

DividendValue — Cash dividend for underlying stock

timetable

Cash dividend for the underlying stock, specified as the comma-separated pair consisting of `'DividendValue'` and a timetable.

Data Types: `timetable`

Optional RollGeskeWhaley Name-Value Pair Arguments

`DividendType` — Stock dividend type
`"cash"` (default) | string with value `"cash"` | character vector with value `'cash'`

Stock dividend type, specified as the comma-separated pair consisting of `'DividendType'` and a string or character vector.

Data Types: `char` | `string`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"RollGeskeWhaley"` | character vector with value `'RollGeskeWhaley'`

Analytic pricing method, specified as the comma-separated pair consisting of `'PricingMethod'` and a string or character vector.

Note The default pricing method for a `BlackScholes` model is a `BlackScholes` pricer.

Data Types: `double`

Properties

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendValue` — Cash dividend for underlying stock
timetable

Cash dividend for the underlying stock, returned as a timetable.

Data Types: `timetable`

`DividendType` — Stock dividend type
`"cash"` (default) | string with value `"cash"`

Stock dividend type, returned as a string.

Data Types: `char` | `string`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"RollGeskeWhaley"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use a Roll-Geske-Whaley Pricer and Black-Scholes Model to Price Vanilla Instrument

This example shows the workflow to price an American exercise `Vanilla` instrument when you use a `BlackScholes` model and a `RollGeskeWhaley` pricing method.

Create `Vanilla` Instrument Object

Use `fininstrument` to create a `Vanilla` instrument object.

```
VanillaOpt = fininstrument("Vanilla",'ExerciseDate',datetime(2022,9,15),'Strike',105,'ExerciseSty
```

```
VanillaOpt =
  Vanilla with properties:

       OptionType: "call"
    ExerciseStyle: "american"
     ExerciseDate: 15-Sep-2022
           Strike: 105
             Name: "vanilla_american_instrument"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.07)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.0700
    Correlation: 1
```

Create ratecurve Object

Create a flat ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
myRC = ratecurve("zero",Settle,datetime(2023,9,15),.05,'Basis',12);
```

Create RollGeskeWhaley Pricer Object

Use finpricer to create a RollGeskeWhaley pricer object and use the ratecurve object for the 'DiscountCurve' name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',100,'
```

```
outPricer =
  RollGeskeWhaley with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: [1x1 timetable]
     DividendType: "cash"
```

Price Vanilla Instrument

Use price to compute the price and sensitivities for the Vanilla instrument.

```
[Price, outPR] = price(outPricer,VanillaOpt,["all"])
```

```
Price = 14.9582
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Lambda      Vega      Theta      Rho
    _____    ________    ________    ______    ______    ________    _____

    14.958    0.87494    0.01471    5.8493    41.184    -3.9873    290.2
```

Version History

Introduced in R2020a

See Also

Functions

fininstrument | finmodel

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

SABR

Create SABR pricer object for Swaption instrument using SABR model

Description

Create and price a Swaption instrument object with a SABR model and a SABR pricing method using this workflow:

1. Use `fininstrument` to create a Swaption instrument object.
2. Use `finmodel` to specify a SABR model for the Swaption instrument object.
3. Use `finpricer` to specify a SABR pricer object for the Swaption instrument object.

> **Note** If you do not specify `ProjectionCurve` when you create a Swaption instrument with the SABR pricer, the `ProjectionCurve` value defaults to the `DiscountCurve` value.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for a Swaption instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
SABRPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'
Model',model)
```

Description

`SABRPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model)` creates a SABR pricer object by specifying `PricerType` and the required name-value pair argument `Model` to set properties on page 11-3342 using name-value pairs. For example, `SABRPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',SABRModel)` creates a SABR pricer object.

Input Arguments

`PricerType` — Pricer type
string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: `char` | `string`

SABR Name-Value Pair Arguments

Specify required pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `SABRPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',SABRModel)`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of a previously created `ratecurve` object.

Data Types: `object`

`Model` — Model
SABR model object

Model, specified as the comma-separated pair consisting of `'Model'` and the name of a previously created SABR model object using `finmodel`.

Data Types: `object`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object

Data Types: `object`

`Model` — Model object
SABR model object

Model, returned as a SABR model object.

Data Types: `object`

Object Functions

price	Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer
volatilities	Compute implied volatilities when using SABR pricer

Examples

Use SABR Pricer and SABR Model to Price Swaption Instrument

This example shows the workflow to price a `Swaption` instrument when you use a SABR model and a SABR pricing method.

Create ratecurve Object

Create a ratecurve object using ratecurve.

```
Settle = datetime(2018,9,15);
Type = 'zero';
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])]';
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;

myRC = ratecurve('zero',Settle,ZeroDates,ZeroRates)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 0
               Dates: [10x1 datetime]
               Rates: [10x1 double]
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create Swap Instrument Object

Use fininstrument to create the underlying Swap instrument object.

```
Swap = fininstrument("Swap",'Maturity',datetime(2023,1,30),'LegRate',[0.018 0.24],'LegType',["fix
```

```
Swap =
  Swap with properties:

                   LegRate: [0.0180 0.2400]
                   LegType: ["fixed"    "float"]
                     Reset: [2 2]
                     Basis: [1 1]
                  Notional: 100
         LatestFloatingRate: [NaN NaN]
                ResetOffset: [0 0]
   DaycountAdjustedCashFlow: [0 0]
           ProjectionCurve: [1x2 ratecurve]
      BusinessDayConvention: ["actual"    "actual"]
                  Holidays: NaT
              EndMonthRule: [1 1]
                 StartDate: 30-Mar-2020
                  Maturity: 30-Jan-2023
                      Name: "swap_instrument"
```

Create Swaption Instrument Object

Use fininstrument to create a Swaption instrument object.

```
Swaption = fininstrument("Swaption",'Strike',0.275,'ExerciseDate',datetime(2021,7,30),'Swap',Swap
```

```
Swaption =
  Swaption with properties:
```

```
       OptionType: "put"
    ExerciseStyle: "european"
     ExerciseDate: 30-Jul-2021
           Strike: 0.2750
             Swap: [1x1 fininstrument.Swap]
             Name: "swaption_instrument"
```

Create SABR Model Object

Use `finmodel` to create a SABR model object.

```
SABRModel = finmodel("SABR",'Alpha',0.032,'Beta',0.04,'Rho',.08,'Nu',0.49,'Shift',0.002)
```

```
SABRModel =
  SABR with properties:

            Alpha: 0.0320
             Beta: 0.0400
              Rho: 0.0800
               Nu: 0.4900
            Shift: 0.0020
    VolatilityType: "black"
```

Create SABR Pricer Object

Use `finpricer` to create a SABR pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',SABRModel,'DiscountCurve',myRC)
```

```
outPricer =
  SABR with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.SABR]
```

Price `Swaption` Instrument

Use `price` to compute the price for the `Swaption` instrument.

```
Price = price(outPricer,Swaption)
```

```
Price = 10.3771
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel`

Topics

FiniteDifference

Create FiniteDifference pricer object for Barrier, DoubleBarrier, or Vanilla instrument using a BlackScholes, Heston, Merton, or Bates model

Description

Create and price a Vanilla, Barrier, or DoubleBarrier instrument object with a BlackScholes, Heston, Bates, Merton, or Dupire model and a FiniteDifference pricing method using this workflow:

1. Use fininstrument to create the Barrier, DoubleBarrier, or Vanilla instrument object.
2. Use finmodel to specify the BlackScholes model for a Barrier or DoubleBarrier instrument or a Heston, Bates, Dupire, or Merton model for the Vanilla instrument object.
3. Use finpricer to specify the FiniteDifference pricer object for the Barrier, DoubleBarrier, or Vanilla instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available pricing methods for a Vanilla, Barrier, or DoubleBarrier instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
FiniteDifferencePricerObj = finpricer(PricerType,'Model',model,'
DiscountCurve',ratecurve_obj,'SpotPrice',spotprice_value)
FiniteDifferencePricerObj = finpricer(___,Name,Value)
```

Description

`FiniteDifferencePricerObj = finpricer(PricerType,'Model',model,'DiscountCurve',ratecurve_obj,'SpotPrice',spotprice_value)` creates a FiniteDifference pricer object by specifying PricerType and sets the properties on page 11-3349 for the required name-value pair arguments Model, DiscountCurve, and SpotPrice.

`FiniteDifferencePricerObj = finpricer(___,Name,Value)` sets optional properties on page 11-3349 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `FiniteDifferencePricerObj = finpricer("FiniteDifference",'Model',BSModel,'DiscountCurve',ratecurve_obj,'SpotPrice',100,'DividendValue',.025,'DividendType',"cash")` creates a FiniteDifference pricer object. You can specify multiple name-value pair arguments.

Input Arguments

PricerType — Pricer type
string with value "FiniteDifference" | character vector with value 'FiniteDifference'

Pricer type, specified as a string with the value of `"FiniteDifference"` or the character vector with a value of `'FiniteDifference'`.

Data Types: `char` | `string`

`FiniteDifference` Name-Value Pair Arguments

Specify required and optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `FiniteDifferencePricerObj = finpricer("FiniteDifference",'Model',BSModel,'DiscountCurve',ratecurve_obj,'SpotPrice',100,'DividendValue',.025,'DividendType',"cash")`

Required FiniteDifference Name-Value Pair Arguments

`Model` — Model object
object

Model object, specified as the comma-separated pair consisting of `'Model'` and the name of the previously created `Merton`, `Bates`, or `Heston` model object using `finmodel`.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

This property is read-only.

`ratecurve` object for discounting cash flows, specified as the comma-separated pair consisting of `'DiscountCurve'` and the name of the `ratecurve` object.

Note Specify a flat `ratecurve` object for `DiscountCurve`. If you use a nonflat `ratecurve` object, the software uses the rate in the `ratecurve` object at `Maturity` and assumes that the value is constant for the life of the equity option.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of `'SpotPrice'` and a scalar nonnegative numeric.

Data Types: `double`

Optional FiniteDifference Name-Value Pair Arguments

`DividendValue` — Dividend yield or dividend schedule
`0` (default) | scalar | timetable

Dividend yield or dividend schedule, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar for a dividend yield or a timetable for a dividend schedule.

Data Types: double | timetable

DividendType — Dividend type
"continuous" (default) | string with value "cash" or "continuous" | character vector with value 'cash' or 'continuous'

Dividend type, specified as the comma-separated pair consisting of 'DividendType' and a character vector or string. DividendType must be "cash" for actual dollar dividends or "continuous" for a continuous dividend yield.

Note When you use a Barrier instrument, you must set DividendType to "cash".

Data Types: char | string

SpotGridSize — Size of spot grid for finite difference grid
400 (default) | scalar numeric

Size of the spot grid for the finite difference grid, specified as the comma-separated pair consisting of 'SpotGridSize' and a scalar numeric.

Data Types: double

SpotPriceMax — Maximum price for price grid boundary
if unspecified, SpotPrice values are calculated using asset distributions at maturity (default) | positive scalar

Maximum price for the price grid boundary, specified as the comma-separated pair consisting of 'SpotPriceMax' and a positive scalar.

Data Types: single | double

VarianceGridSize — Number of nodes of variance grid for finite difference grid
200 (default) | scalar numeric

Number of nodes of the variance grid for the finite difference grid, specified as the comma-separated pair consisting of 'VarianceGridSize' and a scalar numeric.

Note VarianceGridSize is supported only when you use a Heston or Bates model.

Data Types: double

VarianceMax — Maximum variance for variance grid boundary
1.0 (default) | scalar numeric

Maximum variance for the variance grid boundary, specified as the comma-separated pair consisting of 'VarianceMax' as a scalar numeric.

Note VarianceMax is supported only when you use a Heston or Bates model.

Data Types: double

`TimeGridSize` — Number of nodes of time grid for finite difference grid
`100` (default) | positive numeric scalar

Number of nodes of the time grid for the finite difference grid, specified as the comma-separated pair consisting of `'TimeGridSize'` and a positive numeric scalar.

Data Types: `double`

`InterpMethod` — Method of interpolation for estimating the implied volatility surface from `ImpliedVolData`
`'linear'` (default) | string with value `"linear"`, `"makima"`, `"spline"`, or `"tpaps"` | character vector with value `'linear'`, `'makima'`, `'spline'`, or `'tpaps'`

Method of interpolation for estimating the implied volatility surface from `ImpliedVolData` for use only with a `Dupire` model, specified as the comma-separated pair consisting of `'InterpMethod'` and a string or character vector with one of the following values:

- `'linear'` — Linear interpolation
- `'makima'` — Modified Akima cubic Hermite interpolation
- `'spline'` — Cubic spline interpolation
- `'tpaps'` — Thin-plate smoothing spline interpolation

Note The `'tpaps'` method uses the thin-plate smoothing spline functionality from Curve Fitting Toolbox.

The `'makima'` and `'spline'` methods work only for gridded data. For scattered data, use the `'linear'` or `'tpaps'` methods.

For more information on gridded or scattered data and details on interpolation methods, see “Gridded and Scattered Sample Data” and “Interpolating Gridded Data”.

Data Types: `char` | `string`

Properties

`Model` — Model
object

Model, returned as a model object.

Data Types: `object`

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

This property is read-only.

`ratecurve` object for discounting cash flows, returned as the `ratecurve` object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendValue` — Dividend yield or dividend schedule
`0` (default) | scalar | timetable

Dividend yield or dividend schedule, returned as a scalar for a dividend yield or a timetable for a dividend schedule.

Data Types: `double` | `timetable`

`DividendType` — Dividend type
`"continuous"` (default) | string with value `"cash"` or `"continuous"`

This property is read-only.

Dividend type, returned as a string. The `DividendType` is either `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `string`

`GridProperties` — Grid properties
`400` (default) | scalar numeric

Grid properties, returned as a struct.

For a `Dupire` model, `GridProperties` contains the following fields:

- `SpotGridSize` — Size of the spot grid for the finite difference grid, returned as a scalar numeric.
- `SpotPriceMax` — Maximum price for price grid boundary, returned as a positive scalar.
- `TimeGridSize` — Number of nodes of the time grid for the finite difference grid, returned as a positive numeric scalar.
- `InterpMethod` — Method of interpolation for estimating the implied volatility surface, returned as a string.

For a `Heston` or `Bates` model, `GridProperties` contains the following fields:

- `VarianceGridSize` — Size of the variance grid for the finite difference grid, returned as a scalar numeric.
- `VarianceMax` — Maximum variance for the variance grid boundary, returned as a scalar numeric.

Data Types: `struct`

Object Functions

price Compute price for equity instrument with FiniteDifference pricer

Examples

Use Finite Difference Pricer and Black-Scholes Model to Price Barrier Instrument

This example shows the workflow to price a `Barrier` instrument when you use a `BlackScholes` model and a `FiniteDifference` pricing method.

Create Barrier Instrument Object

Use `fininstrument` to create a `Barrier` instrument object.

```
BarrierOpt = fininstrument("Barrier",'Strike',105,'ExerciseDate',datetime(2019,1,1),'OptionType'
```

```
BarrierOpt =
  Barrier with properties:

       OptionType: "call"
           Strike: 105
      BarrierType: "do"
     BarrierValue: 40
           Rebate: 0
    ExerciseStyle: "american"
     ExerciseDate: 01-Jan-2019
             Name: "barrier_option"
```

Create BlackScholes Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.30)
```

```
BlackScholesModel =
  BlackScholes with properties:

     Volatility: 0.3000
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,1,1);
Maturity = datetime(2023,1,1);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',1)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
       Compounding: -1
             Basis: 1
             Dates: 01-Jan-2023
             Rates: 0.0350
            Settle: 01-Jan-2018
      InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create FiniteDifference Pricer Object

Use `finpricer` to create a `FiniteDifference` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("FiniteDifference",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPric
```

```
outPricer =
  FiniteDifference with properties:

     DiscountCurve: [1x1 ratecurve]
             Model: [1x1 finmodel.BlackScholes]
         SpotPrice: 100
    GridProperties: [1x1 struct]
      DividendType: "continuous"
     DividendValue: 0
```

Price `Barrier` Instrument

Use `price` to compute the price and sensitivities for the `Barrier` instrument.

```
[Price, outPR] = price(outPricer,BarrierOpt,["all"])
```

```
Price = 11.3230
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: [1x1 struct]
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta      Gamma      Lambda      Theta       Rho        Vega
    _____    ________    _______    _______    ________    _______    _______

    11.323    0.54126    0.0132     4.7802     -7.4408     42.766     39.627
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `timetable`

Topics

“Price European Vanilla Call Options Using Black-Scholes Model and Different Equity Pricers” on page 1-97
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

TurnbullWakeman

Create `TurnbullWakeman` pricer object for `Asian` instrument using `BlackScholes` model

Description

Create and price a `Asian` instrument object with a `BlackScholes` model and a `TurnbullWakeman` pricing method using this workflow:

1. Use `fininstrument` to create an `Asian` instrument object.
2. Use `finmodel` to specify a `BlackScholes` model for the `Asian` instrument object.
3. Use `finpricer` to specify a `TurnbullWakeman` pricer object for the `Asian` instrument object.

For more information on this workflow, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22.

For more information on the available instruments, models, and pricing methods for an `Asian` instrument, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
TurnbullWakemanPricerObj = finpricer(PricerType,'
DiscountCurve',ratecurve_obj,'Model',model,'SpotPrice',spotprice_value)
TurnbullWakemanPricerObj = finpricer( ___ ,Name,Value)
```

Description

`TurnbullWakemanPricerObj = finpricer(PricerType,'DiscountCurve',ratecurve_obj,'Model',model,'SpotPrice',spotprice_value)` creates a `TurnbullWakeman` pricer object by specifying `PricerType` and sets the properties on page 11-3355 for the required name-value pair arguments `DiscountCurve`, `Model`, and `SpotPrice`.

`TurnbullWakemanPricerObj = finpricer( ___ ,Name,Value)` to set optional properties on page 11-3355 using additional name-value pairs in addition to the required arguments in the previous syntax. For example, `TurnbullWakemanPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',100)` creates a `TurnbullWakeman` pricer object.

Input Arguments

`PricerType` — Pricer type

string with value `"Analytic"` | character vector with value `'Analytic'`

Pricer type, specified as a string with the value of `"Analytic"` or a character vector with the value of `'Analytic'`.

Data Types: char | string

TurnbullWakeman Name-Value Pair Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: TurnbullWakemanPricerObj = finpricer("Analytic",'DiscountCurve',ratecurve_obj,'Model',BSModel,'SpotPrice',1000,'DividendType',"continuous",'DividendValue',100,'PricingMethod','TurnbullWakeman')

Required TurnbullWakeman Name-Value Pair Arguments

DiscountCurve — ratecurve object for discounting cash flows
ratecurve object

ratecurve object for discounting cash flows, specified as the comma-separated pair consisting of 'DiscountCurve' and the name of a previously created ratecurve object.

Specify a flat ratecurve object for DiscountCurve. If you use a nonflat ratecurve object, the software uses the rate in the ratecurve object at Maturity and assumes that the value is constant for the life of the equity option.

Data Types: object

Model — Model
BlackScholes model object

Model, specified as the comma-separated pair consisting of 'Model' and the name of a previously created BlackScholes model object using finmodel.

Data Types: object

SpotPrice — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, specified as the comma-separated pair consisting of 'SpotPrice' and a scalar nonnegative numeric.

Data Types: double

Optional TurnbullWakeman Name-Value Pair Arguments

DividendType — Stock dividend type
"continuous" (default) | string with value "cash" or "continuous" | character vector with value 'cash' or 'continuous'

Stock dividend type, specified as the comma-separated pair consisting of 'DividendType' and a character vector or string. DividendType must be "cash" for actual dollar dividends or "continuous" for a continuous dividend yield.

Data Types: char | string

`DividendValue` — Dividend yield or dividend schedule for underlying stock
`0` (default) | scalar numeric | timetable

Dividend yield for the underlying stock, specified as the comma-separated pair consisting of `'DividendValue'` and a scalar numeric for a dividend yield or a timetable for a dividend schedule.

Note `DividendValue` must be a scalar for a `"continuous"` `DividendType` or a timetable for a `"cash"` `DividendType`.

Data Types: `double` | `timetable`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"TurnbullWakeman"` | character vector with value `'TurnbullWakeman'`

Analytic pricing method, specified as the comma-separated pair consisting of `'PricingMethod'` and a string or character vector.

Note The default pricing method for a `BlackScholes` model is a `BlackScholes` pricer.

Data Types: `double`

Properties

`DiscountCurve` — `ratecurve` object for discounting cash flows
`ratecurve` object

`ratecurve` object for discounting cash flows, returned as a `ratecurve` object.

Data Types: `object`

`Model` — Model
`BlackScholes` model object

Model, returned as a `BlackScholes` model object.

Data Types: `object`

`SpotPrice` — Current price of underlying asset
nonnegative numeric

Current price of the underlying asset, returned as a scalar nonnegative numeric.

Data Types: `double`

`DividendType` — Stock dividend type
`'continuous'` (default) | string with value `"cash"` or `"continuous"`

This property is read-only.

Stock dividend type, returned as a string. `DividendType` is either `"cash"` for actual dollar dividends or `"continuous"` for a continuous dividend yield.

Data Types: `string`

`DividendValue` — Dividend yield or dividend schedule for underlying stock
`0` (default) | scalar nonnegative numeric | timetable

Dividend yield or dividend schedule for the underlying stock, returned as a scalar nonnegative numeric for a dividend yield or a timetable for a dividend schedule.

Data Types: `double` | `timetable`

`PricingMethod` — Analytic pricing method
default pricer associated with `BlackScholes` model (default) | string with value `"TurnbullWakeman"`

Analytic pricing method, returned as a string.

Data Types: `string`

Object Functions

price Compute price for interest-rate, equity, or credit derivative instrument with Analytic pricer

Examples

Use Turnbull-Wakeman Pricer and Black-Scholes Model to Price Asian Instrument

This example shows the workflow to price an `Asian` instrument when you use a `BlackScholes` model and a `TurnbullWakeman` pricing method.

Create `Asian` Instrument Object

Use `fininstrument` to create an `Asian` instrument object.

```
AsianOpt = fininstrument("Asian",'ExerciseDate',datetime(2022,9,15),'Strike',105,'OptionType',"pu
```

```
AsianOpt =
  Asian with properties:

          OptionType: "put"
              Strike: 105
         AverageType: "arithmetic"
        AveragePrice: 0
    AverageStartDate: NaT
       ExerciseStyle: "european"
        ExerciseDate: 15-Sep-2022
                Name: "asian_option"
```

Create `BlackScholes` Model Object

Use `finmodel` to create a `BlackScholes` model object.

```
BlackScholesModel = finmodel("BlackScholes",'Volatility',0.35)
```

```
BlackScholesModel =
  BlackScholes with properties:
```

```
     Volatility: 0.3500
    Correlation: 1
```

Create ratecurve Object

Create a flat `ratecurve` object using `ratecurve`.

```
Settle = datetime(2018,9,15);
Maturity = datetime(2023,9,15);
Rate = 0.035;
myRC = ratecurve('zero',Settle,Maturity,Rate,'Basis',12)
```

```
myRC =
  ratecurve with properties:

                Type: "zero"
         Compounding: -1
               Basis: 12
               Dates: 15-Sep-2023
               Rates: 0.0350
              Settle: 15-Sep-2018
        InterpMethod: "linear"
    ShortExtrapMethod: "next"
     LongExtrapMethod: "previous"
```

Create TurnbullWakeman Pricer Object

Use `finpricer` to create a `TurnbulllWakeman` pricer object and use the `ratecurve` object for the `'DiscountCurve'` name-value pair argument.

```
outPricer = finpricer("analytic",'Model',BlackScholesModel,'DiscountCurve',myRC,'SpotPrice',100,
```

```
outPricer =
  TurnbullWakeman with properties:

    DiscountCurve: [1x1 ratecurve]
            Model: [1x1 finmodel.BlackScholes]
        SpotPrice: 100
    DividendValue: 0
     DividendType: "continuous"
```

Price Asian Instrument

Use `price` to compute the price and sensitivities for the `Asian` instrument.

```
[Price, outPR] = price(outPricer,AsianOpt,["all"])
```

```
Price = 14.3431
```

```
outPR =
  priceresult with properties:

       Results: [1x7 table]
    PricerData: []
```

```
outPR.Results
```

```
ans=1×7 table
    Price      Delta        Gamma       Lambda      Vega      Theta        Rho
    ______    ________    _________    _______    ______    ________    _______

    14.343    -0.37004    0.0085706    -2.5799    44.864    -0.86257    -125.73
```

Version History

Introduced in R2020a

See Also

Functions

`fininstrument` | `finmodel` | `timetable` | `ratecurve`

Topics

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

defprobcurve

Create `defprobcurve` object for credit instrument

Description

Create a `defprobcurve` object for a credit instrument.

After creating a `defprobcurve` object, you can use the associated functions `survprobs`, `hazardrates`, and `defprobstrip`.

To price a `CDS` instrument, you must create a `defprobcurve` object and then create a `Credit` pricer object.

For more information on the available instruments, models, and pricing methods, see “Choose Instruments, Models, and Pricers” on page 1-53.

Creation

Syntax

```
DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities)
DefaultProbCurve = defprobcurve( ___ ,Name,Value)
```

Description

`DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities)` creates a `defprobcurve` object.

`DefaultProbCurve = defprobcurve( ___ ,Name,Value)` sets properties on page 11-3361 using name-value pairs and any of the arguments in the previous syntax. For example, `DefaultProbCurve = defprobcurve(datetime(2017,1,30),[datetime(2018,1,30);datetime(2019,1,30)], [0.005 0.007],'Basis',2)` creates a default probability curve object. You can specify multiple name-value pair arguments.

Input Arguments

`Settle` — Settle date for curve

datetime scalar | string scalar | date character vector

Settle date for curve, specified as a scalar datetime, string, or date character vector.

To support existing code, `defprobcurve` also accepts serial date numbers as inputs, but they are not recommended.

If you use a date character vector or string, the format must be recognizable by `datetime` because the `Settle` property is stored as a datetime.

`ProbDates` — Dates corresponding to `DefaultProbabilities`

datetime array | string array | date character vector

Dates corresponding to `DefaultProbabilities`, specified as an `NPOINTS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `defprobcurve` also accepts serial date numbers as inputs, but they are not recommended.

If you use a date character vector or string, the format must be recognizable by `datetime` because the `ProbDates` property is stored as a datetime.

`DefaultProbabilities` — Default probability data for the curve
vector

Default probability data for the curve, specified as a numeric vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `defprobcurve = defprobcurve(datetime(2017,1,30), [datetime(2018,1,30);datetime(2019,1,30)],[0.005 0.007],'Basis',2)`

`Basis` — Day count basis
`2` (actual/360) (default) | integer from `0` to `13`

Day count basis, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer.

- 0 — actual/actual
- 1 — 30/360 (SIA)
- 2 — actual/360
- 3 — actual/365
- 4 — 30/360 (PSA)
- 5 — 30/360 (ISDA)
- 6 — 30/360 (European)
- 7 — actual/365 (Japanese)
- 8 — actual/actual (ICMA)
- 9 — actual/360 (ICMA)
- 10 — actual/365 (ICMA)
- 11 — 30/360E (ICMA)
- 12 — actual/365 (ISDA)
- 13 — BUS/252

For more information, see “Basis” on page 2-228.

Data Types: `double`

Properties

`Settle` — Settlement date

datetime

Settlement date, returned as a datetime.

Data Types: `datetime`

`Basis` — Day count basis

2 (actual/360) (default) | integer from `0` to `13`

This property is read-only.

Day count basis of the instrument, returned as a scalar integer.

Data Types: `double`

`Dates` — Dates corresponding to rate data

datetime

Dates corresponding to the rate data, returned as a datetime.

Data Types: `datetime`

`DefaultProbabilities` — Default probabilities for the curve

vector

Default probabilities for the curve, returned as a vector.

Data Types: `double`

Object Functions

survprobs	Compute survival probability based on default probability curve
hazardrates	Compute hazard rates based on default probability curve
defprobstrip	Bootstrap defprobcurve object from market CDS instruments

Examples

Create `defprobcurve` Object

Create a `defprobcurve` object using `defprobcurve`.

```
Settle = datetime(2017,9,20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;

DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities,'Basis',2)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                 Settle: 20-Sep-2017
                  Basis: 2
```

```
                Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

Version History

Introduced in R2020a

Serial date numbers not recommended

Not recommended starting in R2022b

Although `defprobcurve` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

Functions

`ratecurve`

Topics

“Bootstrapping a Default Probability Curve from Credit Default Swaps” on page 8-42
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Workflow for Creating and Analyzing a defprobcurve” on page 1-51
“Choose Instruments, Models, and Pricers” on page 1-53

survprobs

Compute survival probability based on default probability curve

Syntax

```
outSurvProbs = survprobs(obj,inDates)
```

Description

`outSurvProbs = survprobs(obj,inDates)` computes the survival probability based on the default probability curve object.

Examples

Calculate Survival Probability Based on Default Probability Curve

Create a `defprobcurve` object using `defprobcurve` and then use `survprobs` to calculate the survival probability.

```
Settle = datetime(2017,9,20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
DefProbDates = Settle + DefProbTimes;

DefaultProbCurve = defprobcurve(Settle,DefProbDates,DefaultProbabilities,'Basis',5)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                  Settle: 20-Sep-2017
                   Basis: 5
                   Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

```
SurvProbTimes = [calmonths([6 12 18])];
SurvProbDates = Settle + SurvProbTimes;
outSurvProb = survprobs(DefaultProbCurve, SurvProbDates)
```

```
outSurvProb = 3×1

    0.9950
    0.9930
    0.9915
```

Input Arguments

`obj` — `defprobcurve` object
`defprobcurve` object

`defprobcurve` object, specified as a previously created `defprobcurve` object.

Data Types: `object`

`inDates` — Survival probability dates
datetime array | string array | date character vector

Survival probability dates, specified as a vector using a datetime array, string array, or date character vectors.

To support existing code, `survprobs` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`outSurvProbs` — Survival probability
numeric

Survival probability, returned as a numeric.

Version History

Introduced in R2020a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `survprobs` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`hazardrates` | `defprobstrip`

Topics
“Bootstrapping a Default Probability Curve from Credit Default Swaps” on page 8-42
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

hazardrates

Compute hazard rates based on default probability curve

Syntax

```
OutRates = hazardrates(obj)
```

Description

`OutRates = hazardrates(obj)` computes hazard rates based on a `defprobcurve` object.

Examples

Calculate Hazard Rates Based on Default Probability Curve

Create a `defprobcurve` object using `defprobcurve` and then use `hazardrates` to calculate the hazard rates.

```
Settle = datetime(2017,9,20);
DefProbTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
DefaultProbabilities = [0.005 0.007 0.01 0.015 0.026 0.04 0.077 0.093 0.15 0.20]';
ProbDates = Settle + DefProbTimes;

DefaultProbCurve = defprobcurve(Settle,ProbDates,DefaultProbabilities,'Basis',5)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                  Settle: 20-Sep-2017
                   Basis: 5
                   Dates: [10x1 datetime]
    DefaultProbabilities: [10x1 double]
```

```
hazardrates(DefaultProbCurve)
```

```
ans = 10×1

    0.0100
    0.0040
    0.0030
    0.0051
    0.0112
    0.0145
    0.0197
    0.0058
    0.0065
    0.0061
```

Input Arguments

obj — defprobcurve object
defprobcurve object

defprobcurve object, specified as a previously created defprobcurve object.

Data Types: object

Output Arguments

OutRates — Hazard rates
numeric

Hazard rates, returned as a numeric.

Version History

Introduced in R2020a

See Also

survprobs | defprobstrip

Topics

“Bootstrapping a Default Probability Curve from Credit Default Swaps” on page 8-42
“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

defprobstrip

Bootstrap `defprobcurve` object from market CDS instruments

Syntax

```
OutCurve = defprobstrip(ZeroCurve,MarketInstruments,MarketQuotes)
OutCurve = defprobstrip( ___ ,Name,Value)
```

Description

`OutCurve = defprobstrip(ZeroCurve,MarketInstruments,MarketQuotes)` bootstraps a `defprobcurve` object from market CDS instruments.

`OutCurve = defprobstrip( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any of the input argument combinations in the previous syntax.

Examples

Bootstrap Default Probability Curve from Market CDS Instruments

This example shows how to use `defprobstrip` to bootstrap a `defprobcurve` object based on market CDS instruments.

Create `ratecurve` Object for Zero-Rate Curve

Create a `ratecurve` object using `ratecurve`.

```
Settle = datetime(2017,9,15);
ZeroTimes = [calmonths(6) calyears([1 2 3 4 5 7 10 20 30])];
ZeroRates = [0.0052 0.0055 0.0061 0.0073 0.0094 0.0119 0.0168 0.0222 0.0293 0.0307]';
ZeroDates = Settle + ZeroTimes;
ZeroCurve = ratecurve("zero",Settle,ZeroDates,ZeroRates);
```

Market CDS Spreads and Vector of Market CDS Instruments

Define the market CDS spreads and use `fininstrument` to create a vector of market CDS instrument objects.

```
SpreadTimes = [1 2 3 4 5 7 10 20 30]';
Spread = [140 175 210 265 310 360 410 460 490]';
MarketDates = datemnth(Settle,12*SpreadTimes);

NumMarketInst = length(MarketDates);
ContractSpreadBP = zeros(NumMarketInst,1);

MarketCDSInstruments(NumMarketInst,1) = fininstrument("cds", ...
      'ContractSpread', ContractSpreadBP(end), 'Maturity', MarketDates(end));
  for k = 1:NumMarketInst
      MarketCDSInstruments(k,1) = fininstrument("cds", ...
          'ContractSpread', ContractSpreadBP(k), 'Maturity', MarketDates(k));
  end
```

Use `defprobstrip` to create a `defprobcurve` object.

```
DefaultProbCurve = defprobstrip(ZeroCurve,MarketCDSInstruments, Spread)
```

```
DefaultProbCurve =
  defprobcurve with properties:

                  Settle: 15-Sep-2017
                   Basis: 2
                   Dates: [9x1 datetime]
    DefaultProbabilities: [9x1 double]
```

Input Arguments

`ZeroCurve` — Zero-rate curve

`ratecurve` object

Zero-rate curve, specified by a previously created `ratecurve`.

Data Types: `object`

`MarketInstruments` — Market CDS instruments

vector

Market CDS instruments, specified as an `NINST`-by-`1` vector.

Data Types: `double`

`MarketQuotes` — Market spread

vector

Market quotes, specified as an `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `DefaultProbCurve = defprobstrip(ZeroCurve, MarketInstruments, MarketQuotes,'QuoteType',"upfront")`

`QuoteType` — Market quote type

`"fairspread"` (default) | string with value of `"fairspread"` or `"upfront"` | character vector with value of `'fairspread'` or `'upfront'`

Frequency of payments per year, specified as the comma-separated pair consisting of `'QuoteType'` and a scalar character vector or string.

- `"fairspread"` — CDS break-even spread for zero upfront price
- `"upfront"` — CDS upfront price for a given contractual spread

Data Types: `char` | `string`

`ProbDates` — Dates for probability data
`NINST`-by-`1` vector of maturity dates in `MarketInstruments` input (default) | datetime array | string array | date character vector

Dates for probability data, specified as the comma-separated pair consisting of `'ProbDates'` and a `P`-by-`1` vector of dates for the output `defprobcurve` object, given as a datetime array, string array, or date character vectors.

To support existing code, `defprobstrip` also accepts serial date numbers as inputs, but they are not recommended.

Output Arguments

`OutCurve` — Default probability curve
`defprobcurve` object

Default probability curve, returned as a `defprobcurve` object with the following properties:

- `Settle`
- `Basis`
- `Dates`
- `DefaultProbabilities`

Version History

Introduced in R2020a

Serial date numbers not recommended
Not recommended starting in R2022b

Although `defprobstrip` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`survprobs` | `hazardrates`

Topics
“Bootstrapping a Default Probability Curve from Credit Default Swaps” on page 8-42

“Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” on page 1-22
“Choose Instruments, Models, and Pricers” on page 1-53

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` on page A-2, `Warnings` on page A-2, `ConstRate` on page A-3, and `BarrierMethod` on page A-3.

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 have 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. So 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. So, setting `ConstRate 'off'` dramatically increases the memory and processor demands on the computer.

BarrierMethod Field

When using binomial trees to price barrier options, this may require many 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`, `barrierbyitt`, `crrprice`, `eqpprice`, `ittprice`, `crrsens`, `eqpsens`, or `ittsens`. `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, except for 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'`.

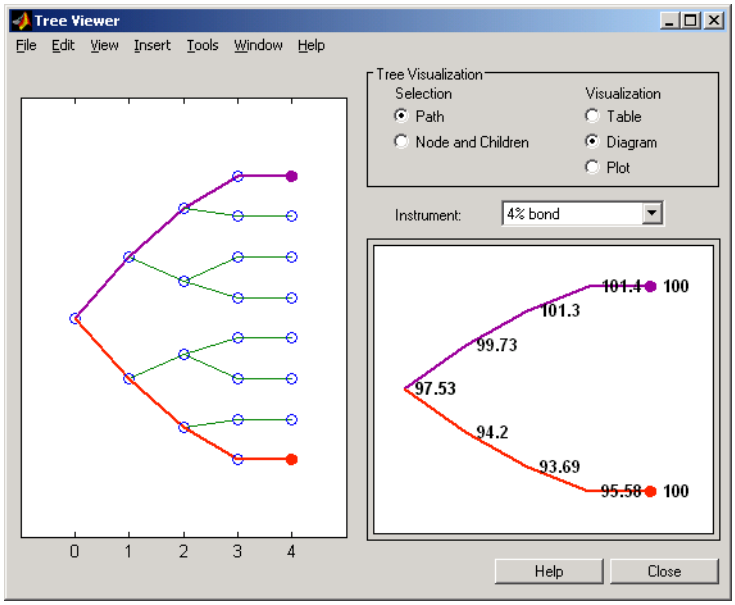
```
OptionsNoCR = derivset('ConstR', 'off')

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.

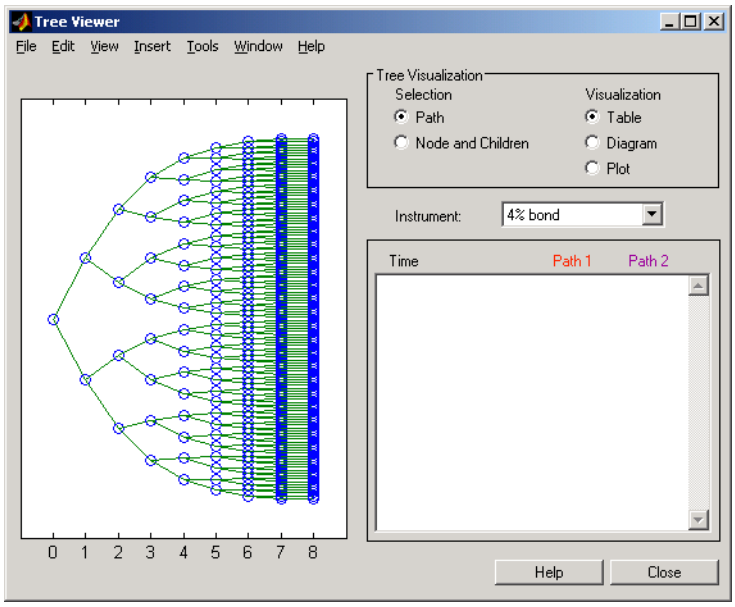
```
size(BondPriceTree.PBush{end})

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
```

See Also

`instasian` | `instbarrier` | `instcompound` | `instlookback` | `instoptstock`

Related Examples

- “Pricing Equity Derivatives Using Trees” on page 3-64
- “Pricing Options Structure” on page A-2
- “Pricing European Call Options Using Different Equity Models” on page 3-88
- “Pricing Using the Black-Scholes Model” on page 3-82
- “Compute Option Prices on a Forward” on page 11-1597
- “Compute Forward Option Prices and Delta Sensitivities” on page 11-1662
- “Compute the Option Price on a Future” on page 11-1598
- “Pricing Asian Options” on page 3-110

More About

- “Supported Interest-Rate Instrument Functions” on page 2-3
- “Supported Equity Derivative Functions” on page 3-19
- “Supported Energy Derivative Functions” on page 3-34
- “Mapping Financial Instruments Toolbox Functions for Interest-Rate Instrument Objects” on page 1-73
- “Mapping Financial Instruments Toolbox Functions for Equity, Commodity, FX Instrument Objects” on page 1-84

Bibliography

Bibliography

Black-Derman-Toy (BDT) Modeling

A description of the Black-Derman-Toy interest-rate model can be found in:

[1] 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 Instruments Toolbox software, can be found in:

[2] 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:

[3] 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:

[4] Hull, J., and A. White. "Numerical Procedures for Implementing Term Structure Models I: Single-Factor Models." *Journal of Derivatives.* 1994.

[5] 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:

[6] 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:

[7] Chriss, Neil A., E. Derman, and I. Kani. “Implied trinomial trees of the volatility smile.” *Journal of Derivatives.* 1996.

Leisen-Reimer Tree (LR) Modeling

To learn about the Leisen-Reimer model, see:

[8] Leisen D.P., M. Reimer. “Binomial Models for Option Valuation – Examining and Improving Convergence.” *Applied Mathematical Finance.* Number 3, 1996, pp. 319–346.

Equal Probabilities Tree (EQP) Modeling

To learn about the Equal Probabilities model, see:

[9] 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:

[10] Bjerksund, P. and G. Stensland. “Closed-Form Approximation of American Options.” *Scandinavian Journal of Management.* Vol. 9, 1993, Suppl., pp. S88–S99.

[11] Bjerksund, P. and G. Stensland. *“Closed Form Valuation of American Options.”*, Discussion paper, 2002.

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 Instruments Toolbox software are:

[12] Chance, Don. M. *An Introduction to Derivatives.* The Dryden Press, 1998, ISBN 0-030-024483-8.

[13] Fabozzi, Frank J. *Treasury Securities and Derivatives.* Frank J. Fabozzi Associates, 1998, ISBN 1-883249-23-6.

[14] Wilmott, Paul. *Derivatives: The Theory and Practice of Financial Engineering.* John Wiley and Sons, 1998, ISBN 0-471-983-89-6.

Fitting Interest-Rate Curve Functions

[15] Nelson, C.R., Siegel, A.F. "Parsimonious modelling of yield curves." *Journal of Business.* Number 60, 1987, pp 473–89.

[16] Svensson, L.E.O. "*Estimating and interpreting forward interest rates: Sweden 1992-4*." International Monetary Fund, IMF Working Paper, 1994, p. 114.

[17] Fisher, M., Nychka, D., Zervos, D. "*Fitting the term structure of interest rates with smoothing splines.*" Board of Governors of the Federal Reserve System, Federal Reserve Board Working Paper, 1995.

[18] Anderson, N., Sleath, J. "New estimates of the UK real and nominal yield curves." *Bank of England Quarterly Bulletin.* November, 1999, pp 384–92.

[19] Waggoner, D. "Spline Methods for Extracting Interest Rate Curves from Coupon Bond Prices," Federal Reserve Board Working Paper, 1997, p. 10.

[20] "Zero-coupon yield curves: technical documentation." *BIS Papers*, Bank for International Settlements, Number 25, October, 2005.

[21] Bolder, D.J., Gusba,S. "Exponentials, Polynomials, and Fourier Series: More Yield Curve Modelling at the Bank of Canada." *Working Papers.* Bank of Canada, 2002, p. 29.

[22] Bolder, D.J., Streliski, D. "Yield Curve Modelling at the Bank of Canada." *Technical Reports.* Number 84, 1999, Bank of Canada.

Interest-Rate Modeling Using Monte Carlo Simulation

[23] Brigo, D. and F. Mercurio. *Interest Rate Models - Theory and Practice with Smile, Inflation and Credit.* Springer Finance, 2006.

[24] Andersen, L. and V. Piterbarg. *Interest Rate Modeling.* Atlantic Financial Press. 2010.

[25] Hull, J, *Options, Futures, and Other Derivatives.* Springer Finance, 2003.

[26] Glasserman, P. *Monte Carlo Methods in Financial Engineering.* Prentice Hall, 2008.

[27] Rebonato, R., K. McKay, and R. White. *The Sabr/Libor Market Model: Pricing, Calibration and Hedging for Complex Interest-Rate Derivatives.* John Wiley & Sons, 2010.

Bootstrapping a Swap Curve

[28] Hagan, P., West, G. "Interpolation Methods for Curve Construction." *Applied Mathematical Finance.* Vol. 13, Number 2, 2006.

[29] Ron, Uri. "A Practical Guide to Swap Curve Construction." *Working Papers.* Bank of Canada, 2000, p. 17.

Bond Futures

[30] Burghardt, G., T. Belton, M. Lane, and J. Papa. *The Treasury Bond Basis.* McGraw-Hill, 2005.

[31] Krgin, Dragomir. *Handbook of Global Fixed Income Calculations.* John Wiley & Sons, 2002.

Credit Derivatives

[32] Beumee, J., D. Brigo, D. Schiemert, and G. Stoyle. “Charting a Course Through the CDS Big Bang.” *Fitch Solutions, Quantitative Research.* Global Special Report. April 7, 2009.

[33] Hull, J., and A. White. “Valuing Credit Default Swaps I: No Counterparty Default Risk.” *Journal of Derivatives.* Vol. 8, pp. 29–40.

[34] O'Kane, D. and S. Turnbull. “Valuation of Credit Default Swaps.” *Lehman Brothers, Fixed Income Quantitative Credit Research.* April, 2003.

[35] O'Kane, D. *Modelling Single-name and Multi-name Credit Derivatives.* Wiley Finance, 2008, pp. 156–169.

Convertible Bonds

[36] Tsiveriotis, K., and C. Fernandes. “Valuing Convertible Bonds with Credit Risk.” *Journal of Fixed Income.* Vol. 8, 1998, pp. 95–102.

[37] Hull, J. *Options, Futures and Other Derivatives.* Fourth Edition. Prentice Hall, 2000, pp. 646–649.