Trading Toolbox™
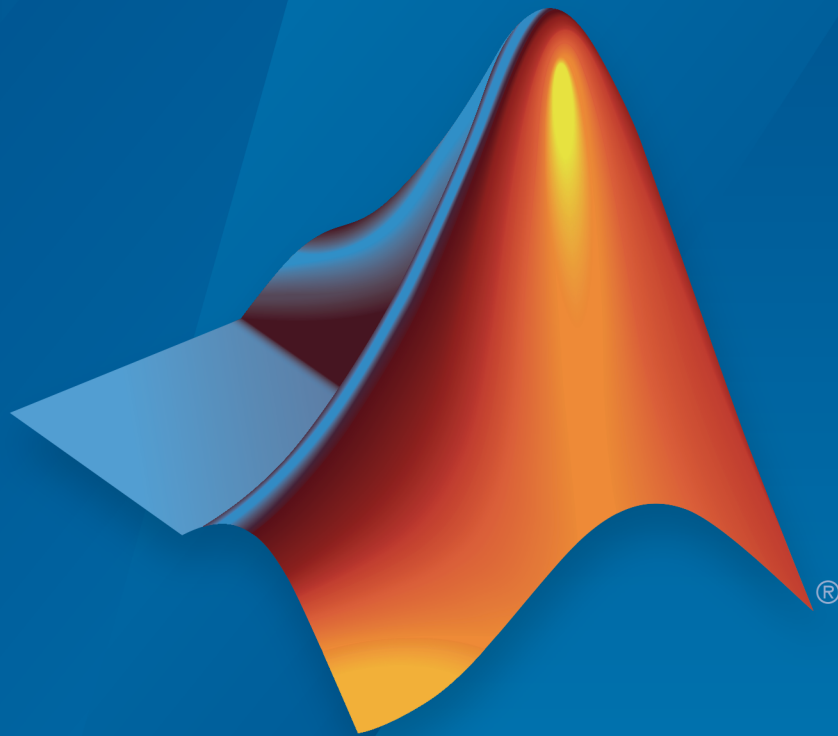
User's Guide

MATLAB®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

March 2013	Online only	New for Version 1.0 (Release 2013a)
September 2013	Online only	Revised for Version 2.0 (Release 2013b)
March 2014	Online only	Revised for Version 2.1 (Release 2014a)
October 2014	Online only	Revised for Version 2.1.1 (Release 2014b)
March 2015	Online only	Revised for Version 2.2 (Release 2015a)
September 2015	Online only	Revised for Version 2.2.1 (Release 2015b)
March 2016	Online only	Revised for Version 3.0 (Release 2016a)
September 2016	Online only	Revised for Version 3.1 (Release 2016b)
March 2017	Online only	Revised for Version 3.2 (Release 2017a)

Contents

Workflow Models

2

Transaction Cost Analysis

3

4

Sample Code for Workflows

5

Functions — Alphabetical List

Getting Started

Trading Toolbox Product Description

Access prices, analyze transaction costs, and send orders to trading systems

Trading Toolbox provides functions for analyzing transaction costs, accessing trade and quote pricing data, defining order types, and sending orders to financial trading markets. The toolbox lets you integrate streaming and event-based data into MATLAB®, enabling you to develop financial trading strategies and algorithms that analyze and react to the market in real time. You can build algorithmic or automated trading strategies that work across multiple asset classes, instrument types, and trading markets while integrating with industry-standard or proprietary trade execution platforms.

With Trading Toolbox you can analyze and estimate transaction costs before placing an order, as well as attribute costs post-trade. You can analyze transaction costs associated with market impact, timing, liquidity, and price appreciation, and use cost curves to minimize transaction costs for single assets or for a portfolio of assets.

Trading Toolbox lets you access real-time streams of tradable instrument data, including quotes, volumes, trades, market depth, and instrument metadata. You can define order types and specify order routing and filling procedures.

Key Features

- Market impact modeling and cost curve generation using Kissell Research Group models
- Trading cost, sensitivity, and post-trade execution analysis
- Access to current, intraday, event-based, and real-time tradable instrument data
- Data filtering by instrument and exchange
- Definable order types and execution instructions
- Access to FIX-compliant trading systems using FIX Flyer™ Engine
- Support for Bloomberg® EMSX, Trading Technologies® X_TRADER®, CQG® Integrated Client, and Interactive Brokers® TWS

Installation

In this section...

Bloomberg

To install Bloomberg EMSX from Bloomberg L.P., find the latest installation files at http://www.bloomberg.com. You need a Bloomberg license to install and run Bloomberg EMSX.

CQG

To install CQG, find the latest installation files at http://www.cqg.com. You need a CQG license to install and run CQG.

The Trading Toolbox no longer supports connection using a 32-bit version of MATLAB. To configure CQG to work with a 64-bit version of MATLAB, see http://www.mathworks.com/matlabcentral/answers/223461-how-can-i-set-up-a-cqg-connection-using-the-trading-toolbox-with-64-bit-version-of-matlab.

FIX Flyer

1. Install FIX Flyer. Find the latest installation files at the FIX Flyer Download Portal.
2. Download the zip file that contains the installation JAR files. Unzip the file.
3. Search the folders for the JAR file `fix-flyer.jar` and the folder named `core`. The JAR file is located in the folder where FIX Flyer is installed. The JAR file points to the folder `core` that contains the other required JAR files.
4. Add the JAR file `fix-flyer.jar` to the static Java® class path. Edit the `javaclasspath.txt` file and enter the path to the file. For example, `..\FIXFlyer\fix-flyer-5.0.1\devkit\lib\fix-flyer.jar`. This file path assumes an installation of FIX Flyer version 5.0.1.

If you are running Linux® or Mac, the JAR file path has a different format. For example, `/FIXFlyer/fix-flyer-5.0.1/devkit/lib/fix-flyer.jar`.

For details about modifying the static Java class path, see "Java Class Path" (MATLAB).

You need a FIX Flyer license to install and run FIX Flyer.

Interactive Brokers

1 Download and install the IB Trader Workstation[SM] Desktop Trading Client. Find the latest installation files at https://www.interactivebrokers.com/en/index.php?f=552.

2 Download and install the Interactive Brokers API software. Find the latest installation files at http://interactivebrokers.github.io/.

3 Configure IB Trader Workstation to enable connections. Follow these steps in IB Trader Workstation:

  a Select **File** > **Global Configuration** under **Application Settings**.
  b Select **API** > **Settings** on the left side.
  c Select **Enable ActiveX and Socket Clients** on the right side.
  d Click **Apply**, then **OK**.
  e Restart MATLAB and connect to IB Trader Workstation.

You need an Interactive Brokers license to install and run Interactive Brokers.

Trading Technologies

To install Trading Technologies, find the latest installation files at http://www.tradingtechnologies.com. You need a Trading Technologies license to install and run Trading Technologies.

See Also
`cqg` | `emsx` | `fixflyer` | `ibtws` | `xtrdr`

Related Examples
- "Create an Order Using Bloomberg EMSX" on page 1-14

- "Create an Order Using CQG" on page 1-12
- "Create an Order Using FIX Flyer" on page 1-20
- "Create an Order Using IB Trader Workstation" on page 1-8
- "Create an Order Using X_TRADER" on page 1-17

Trading System Providers

In this section...
"Supported Providers" on page 1-6
"Connection Requirements" on page 1-6
"Platform Requirements" on page 1-7

Supported Providers

This toolbox supports connections to financial trading systems provided by the following corporations:

- Bloomberg EMSX from Bloomberg L.P. (http://www.bloomberg.com)

  Note: Only the Bloomberg Desktop API is supported.

- CQG (http://www.cqg.com)
- FIX Flyer (http://www.fixflyer.com/)
- IB Trader Workstation from Interactive Brokers (http://www.interactivebrokers.com)

  Note: IB Trader Workstation versions 9.69 and 9.7 and later are supported.

- X_TRADER from Trading Technologies (http://www.tradingtechnologies.com)

See the MathWorks® website for the system requirements for connecting to these trading systems.

Connection Requirements

To connect to these trading systems, additional requirements apply. The following data service providers require you to install proprietary software on your PC:

- Bloomberg EMSX

  Note: You need the Bloomberg Desktop software license for the host on which Trading Toolbox and MATLAB software are running.

- CQG
- FIX Flyer
- Interactive Brokers IB Trader Workstation
- Trading Technologies X_TRADER

You must have a valid license for required client software on your machine.

For more information about how to obtain required software, contact your trading system sales representative.

Platform Requirements

The Trading Toolbox supports 64-bit Windows® only.

These data service providers work only with the Windows platform:

- Bloomberg EMSX
- CQG
- Interactive Brokers
- Trading Technologies X_TRADER

Create an Order Using IB Trader Workstation

Create a connection to the IB Trader Workstation℠ and create a market order based on historical and current data for a security. You can also create orders for a different instrument, such as a futures contract.

Before creating the connection, you must enter your credentials and run the IB Trader Workstation℠ application.

To run this example, you must have the Financial Toolbox™ installed.

Run IB Trader Workstation℠ Application

Ensure the IB Trader Workstation℠ application is running, and that API connections are enabled. Follow these steps in IB Trader Workstation℠.

1 To open the Trader Workstation Configuration (Simulated Trading) dialog box, select **File** > **Global Configuration**.

2 Select **API** > **Settings**.

3 Ensure that the **Enable ActiveX and Socket Clients** check box is selected.

Connect to IB Trader Workstation℠

Connect to the IB Trader Workstation℠ and create connection `ib` using the local host and default port number `7496`.

```
ib = ibtws('',7496);
```

When the `Accept incoming connection attempt` message appears in the IB Trader Workstation℠, click **Yes**.

Retrieve Historical and Current Data

Create the IB Trader Workstation℠ IContract object `ibContract`. This object specifies the security. Retrieve data for Microsoft® stock. Specifying `SMART` as the exchange lets Interactive Brokers® determine which venue to use for data retrieval. To clarify any ambiguity, set the primary exchange for the destination to `NASDAQ`. To retrieve dollar-denominated stock, set the currency type to `USD`. Setting currency type is useful when stocks are dual-listed or multi-listed across different jurisdictions.

```
ibContract = ib.Handle.createContract;
```

```matlab
ibContract.symbol = 'MSFT';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'NASDAQ';
ibContract.currency = 'USD';
```

Define the date range for the last 20 business days, excluding today. To calculate the appropriate start and end dates, this code uses the `daysadd` function from Financial Toolbox™.

```matlab
bizDayConvention = 13; % i.e. BUS/252
currentdate = today;
startDate = daysadd(currentdate,-20,bizDayConvention);
endDate = daysadd(currentdate,-1,bizDayConvention);
```

Retrieve historical data for the last 20 business days.

```matlab
histTradeData = history(ib,ibContract,startDate,endDate);
```

The `history` function accepts additional parameters that let you obtain other historical data such as option-implied volatility, historical volatility, bid prices, ask prices, or midpoints. If you do not specify anything, last traded prices return by default.

Retrieve current price data from the contract.

```matlab
currentData = getdata(ib,ibContract)


currentData =

  struct with fields:

    LAST_PRICE: 62.8500
     LAST_SIZE: 1
        VOLUME: 41273
     BID_PRICE: 62.8400
      BID_SIZE: 17
     ASK_PRICE: 62.8600
      ASK_SIZE: 12
```

Create Trade Market Order

The IB Trader Workstation℠ supports various order types, including basic types such as limit orders, stop orders, and market orders.

Create the IB Trader Workstation℠ `Iorder` object `ibMktOrder`. This object specifies the order. To buy shares, specify the action `BUY`. To specify buying 100 shares, set `totalQuantity` to 100. To create a market order, specify the order type as `MKT`.

```
ibMktOrder = ib.Handle.createOrder;
ibMktOrder.action = 'BUY';
ibMktOrder.totalQuantity = 100;
ibMktOrder.orderType = 'MKT';
```

Set a unique order identifier and send the order to Interactive Brokers®.

```
id = orderid(ib);

result = createOrder(ib,ibContract,ibMktOrder,id)


result =

  struct with fields:

             STATUS: 'Submitted'
             FILLED: 0
          REMAINING: 100
     AVG_FILL_PRICE: 0
            PERM_ID: '1621177315'
          PARENT_ID: 0
    LAST_FILL_PRICE: 0
          CLIENT_ID: 0
           WHY_HELD: ''
```

Specify Different Instrument

You can trade various instruments using the IB Trader Workstation℠ API, including equities, futures, options, futures options, and foreign currencies.

`ibFutures` is the E-mini Standard and Poor's 500 futures contract on the CME Globex with a December 2013 expiry. Specify the symbol as `ES`, the security type as a futures contract `FUT`, the expiry as a `YYYYMM` date format, the exchange as `GLOBEX`, and the currency as `USD`.

```
ibFutures = ib.Handle.createContract;
ibFutures.symbol = 'ES';
ibFutures.secType = 'FUT';
```

```
ibFutures.expiry = '201312'; % Dec 2013
ibFutures.exchange = 'GLOBEX';
ibFutures.currency = 'USD';
```

Retrieve futures data and send orders using the `getdata` and `createOrder` functions.

Close IB Trader Workstation℠ **Connection**

```
close(ib)
```

See Also
close | createOrder | getdata | history | ibtws

Related Examples
- "Create Interactive Brokers Combination Order" on page 4-39
- "Create and Manage an Interactive Brokers Order" on page 4-26
- "Request Interactive Brokers Historical Data" on page 4-32
- "Request Interactive Brokers Real-Time Data" on page 4-35

More About
- "Workflow for Interactive Brokers" on page 2-6

External Websites
- http://www.interactivebrokers.com/en/software/api/api.htm

Create an Order Using CQG

This example shows how to connect to CQG and create a market order.

Connect to CQG

```
c = cqg;
```

Establish Event Handlers

Start the CQG session. Set up event handlers for instrument subscription, orders, and associated events.

```
startUp(c)

streamEventNames = {'InstrumentSubscribed', ...
    'InstrumentChanged','IncorrectSymbol'};

for i = 1:length(streamEventNames)
  registerevent(c.Handle,{streamEventNames{i}, ...
    @(varargin)cqgrealtimeeventhandler(varargin{:})})
end

orderEventNames = {'AccountChanged','OrderChanged','AllOrdersCanceled'};

for i = 1:length(orderEventNames)
  registerevent(c.Handle,{orderEventNames{i}, ...
    @(varargin)cqgordereventhandler(varargin{:})})
end
```

Subscribe to Instrument

Subscribe to a security tied to the EURIBOR.

```
realtime(c,'F.US.IE')
pause(2)
```

Create **CQGInstrument** Object

To use the instrument for creating an order, import the instrument name cqgInstrumentName into the current MATLAB workspace. Then, create the CQGInstrument object cqgInst.

```
cqgInstrumentName = evalin('base','cqgInstrument');
```

```matlab
cqgInst = c.Handle.Instruments.Item(cqgInstrumentName);
```

Set Up Account Credentials

Set the CQG flags to enable account information retrieval.

```matlab
c.Handle.set('AccountSubscriptionLevel','aslNone');
c.Handle.set('AccountSubscriptionLevel','aslAccountUpdatesAndOrders');
pause(2)
accountHandle = c.Handle.Accounts.ItemByIndex(0);
```

Create Market Order

Create a market order that buys one share of the subscribed security `cqgInst` using the account credentials `accountHandle`.

```matlab
orderType = 1;  % Market order flag
quantity = 1;   % Positive quantity is Buy, negative is Sell
oMarket = createOrder(c,cqgInst,orderType,accountHandle,quantity);
oMarket.Place
```

Close CQG Connection

```matlab
close(c)
```

See Also
close | cqg | createOrder | realtime | startUp

Related Examples
- "Create CQG Orders" on page 4-45
- "Request CQG Historical Data" on page 4-51
- "Request CQG Intraday Tick Data" on page 4-54
- "Request CQG Real-Time Data" on page 4-58

More About
- "Workflow for CQG" on page 2-8

External Websites
- CQG API Reference Guide

Create an Order Using Bloomberg EMSX

This example shows how to connect to Bloomberg EMSX and create and route a market order.

For details about connecting to Bloomberg EMSX and creating orders, see the *EMSX API Programmer's Guide*.

Connect to Bloomberg EMSX

1   If you are using `emsx` for the first time, install a Java archive file from Bloomberg for `emsx` and other Bloomberg commands to work correctly.

If you already have `blpapi3.jar` downloaded from Bloomberg, you can find it in your Bloomberg folders at `..\blp\api\APIv3\JavaAPI\lib\blpapi3.jar` or `..\blp\api\APIv3\JavaAPI\v3.x\lib\blpapi3.jar`. If you have `blpapi3.jar`, go to step 3.

If `blpapi3.jar` is not downloaded from Bloomberg, then download it as follows:

a   In your Bloomberg terminal, type `WAPI {GO}` to open the API Developer's Help Site screen.

b   Click API Download Center, then click Desktop API.

c   After downloading `blpapi3.jar` on your system, add it to the MATLAB Java class path using the `javaaddpath` function.

Execute the `javaaddpath` function for every session of MATLAB. To avoid executing the `javaaddpath` function at every session, add `javaaddpath` to your `startup.m` file or add the full path for `blpapi3.jar` to your `javaclasspath.txt` file. For details about `javaclasspath.txt`, see "Java Class Path" (MATLAB). For details about editing your `startup.m` file, see "Startup Options in MATLAB Startup File" (MATLAB).

2   Connect to the Bloomberg EMSX test service.

```
c = emsx('//blp/emapisvc_beta')

c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
```

```
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

Create Market Order Request

Create an order request structure `order` for a buy market order of 400 shares of IBM[®]. Specify the broker as `EFIX`, use any hand instruction, and set the time in force to `DAY`.

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_SIDE = 'BUY';
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(400);
order.EMSX_BROKER = 'EFIX';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_TIF = 'DAY';
```

Create and Route Market Order

Create and route the market order using the Bloomberg EMSX connection `c` and order request structure `order`.

```
events = createOrderAndRoute(c,order);

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier

- Bloomberg EMSX message

Close Bloomberg EMSX Connection

```
close(c)
```

See Also
`close` | `createOrderAndRoute` | `emsx`

Related Examples
- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

More About
- "Workflow for Bloomberg EMSX" on page 2-2

Create an Order Using X_TRADER

This example shows how to connect to Trading Technologies X_TRADER and create a market order.

Connect to Trading Technologies X_TRADER

```
c = xtrdr;
```

Create Instrument for Contract

Create an instrument for a contract of CAISO NP15 EZ Gen Hub 5 MW Peak Calendar-Day Real-Time LMP Futures with an expiration date of August 2014 on the Chicago Mercantile Exchange.

```
createInstrument(c,'Exchange','CME','Product','2F',...
                 'ProdType','Future','Contract','Aug14',...
                 'Alias','SubmitOrderInstrument3')
```

Register Event Handler for Order Server

Register an event handler to check the order server status.

```
sExchange = c.Instrument.Exchange;
c.Gate.registerevent({'OnExchangeStateUpdate',...
                     @(varargin)ttorderserverstatus(varargin{:},sExchange)})
```

Create Order Set and Set Order Properties

Create an empty order set. Then, set order set properties. Setting the first property to true (1) enables the X_TRADER API to send order rejection notifications. Setting the second property to true (1) enables the X_TRADER API to add order pairs for all order updates to the order tracker list in this order set. Setting the third property to ORD_NOTIFY_NORMAL sets the X_TRADER API notification mode for order status events to normal.

```
createOrderSet(c)

c.OrderSet(1).EnableOrderRejectData = 1;
c.OrderSet(1).EnableOrderUpdateData = 1;
c.OrderSet(1).OrderStatusNotifyMode = 'ORD_NOTIFY_NORMAL';
```

Set Position Limit Checks

```
c.OrderSet(1).Set('NetLimits',false)
```

Register Event Handlers for Order Status

Register event handlers to track events associated with the order status.

```
registerevent(c.OrderSet(1),{'OnOrderFilled',...
                             @(varargin)ttorderevent(varargin{:},c)})
registerevent(c.OrderSet(1),{'OnOrderRejected',...
                             @(varargin)ttorderevent(varargin{:},c)})
registerevent(c.OrderSet(1),{'OnOrderSubmitted',...
                             @(varargin)ttorderevent(varargin{:},c)})
registerevent(c.OrderSet(1),{'OnOrderDeleted',...
                             @(varargin)ttorderevent(varargin{:},c)})
```

Enable Order Submission

Open the instrument for trading and enable the X_TRADER API to retrieve market depth information when opening the instrument.

```
c.OrderSet(1).Open(1)
```

Build Order Profile with Existing Instrument

```
orderProfile = createOrderProfile(c,'Instrument',c.Instrument(1));
```

Set Customer Default Property

Assign the customer defaults for trading an instrument.

```
orderProfile.Customer = '<Default>';
```

Set Up Order Profile as Market Order

Set up the order profile as a market order for buying 225 shares.

```
orderProfile.Set('BuySell','Buy')
orderProfile.Set('Qty','225')
orderProfile.Set('OrderType','M')
```

Check Order Server Status

```
nCounter = 1;
while ~exist('bServerUp','var') && nCounter < 20
  % bServerUp is created by ttorderserverstatus
  pause(1)
  nCounter = nCounter + 1;
end
```

Verify Order Server Availability and Submit Order

```matlab
if exist('bServerUp','var') && bServerUp
  % Submit the order
  submittedQuantity = c.OrderSet(1).SendOrder(orderProfile);
  disp(['Quantity Sent: '  num2str(submittedQuantity)])
else
  disp('Order server is down.  Unable to submit order.')
end
```

The X_TRADER API submits the order to the exchange and returns the number of contracts sent for lot-based contracts or the flow quantity sent for flow-based contracts in the output argument `submittedQuantity`.

Close Trading Technologies X_TRADER Connection

```matlab
close(c)
```

See Also

close | createInstrument | createOrderProfile | createOrderSet | xtrdr

Related Examples

- "Listen for X_TRADER Price Updates" on page 4-2
- "Listen for X_TRADER Price Market Depth Updates" on page 4-4
- "Submit X_TRADER Orders" on page 4-8

More About

- "Workflows for Trading Technologies X_TRADER" on page 2-4

External Websites

- https://developer.tradingtechnologies.com/x_trader-api

Create an Order Using FIX Flyer

This example shows how to create a FIX Flyer connection, process event data for sending FIX messages, and submit various orders using FIX messages.

FIX is a financial industry protocol that facilitates low latency trading. For details about the FIX protocol, see FIX Trading Community.

To access the example code, enter `edit FixFlyerExample.m` at the command line.

Connect to FIX Flyer

Import the FIX Flyer Java libraries.

```
import flyer.apps.*;
import flyer.apps.FlyerApplicationManagerFactory.*;
import flyer.core.session.*;
```

Create the FIX Flyer Engine connection `c` using these arguments:

- User name `username`
- Password `password`
- IP address `ipaddress`
- Port number `port`
- Order information port number `orderport`

```
username = 'guest';
password = 'guest';
ipaddress = 'example.fixcomputeserver.com';
port = 12001;
orderport = 13001;

c = fixflyer(username,password,ipaddress,port,orderport);
```

Add Listener and Subscribe to FIX Sessions

Add the FIX Flyer event listener to the FIX Flyer Engine connection. Listen for and display the FIX Flyer Engine event data in the Workspace browser by using the sample event handling listener `fixExampleListener`.

To access the code for the listener, enter `edit fixExampleListener.m`. Or, to process the event data in another way, you can write a custom event handling listener function. For details, see "Create Functions in Files" (MATLAB).

Process the FIX Flyer Engine events `e` using the sample event handling listener `fixExampleListener`. Specify `e` as any letter. `fixExampleListener` returns a handle to the listener `lh`.

```
lh = addListener(c,@(~,e)fixExampleListener(e,c));
```

Subscribe to FIX sessions and set up the FIX Flyer Application Manager. Register with the FIX Flyer session. Connect the FIX Flyer Application Manager to the FIX Flyer Engine and start the internal receiving thread.

```
c.SessionID = flyer.core.session.SessionID('Alpha',...
                                       'Beta','FIX.4.4');
c.FlyerApplicationManager.setLoadDefaultDataDictionary(false);
c.FlyerApplicationManager.registerFIXSession(...
                                   flyer.apps.FixSessionSubscription(...
                                   c.SessionID,true,0));
c.FlyerApplicationManager.connect;
c.FlyerApplicationManager.start;
```

Create FIX Messages

Create two FIX messages using a structure array `order`. Each structure in the array represents one FIX message. Both messages denote a sell side transaction for 1000 IBM shares. The order type is a previously quoted order. The order handling instruction is a private automated execution. The order transaction time is the current moment. The FIX protocol version is 4.4.

Set the `MsgType` to `'D'` to denote a new order.

```
order.BeginString{1,1} = 'FIX.4.4';
order.CLOrdId{1,1} = '338';
order.Side{1,1} = '2';
order.TransactTime{1,1} = datestr(now);
order.OrdType{1,1} = 'D';
order.Symbol{1,1} = 'IBM';
order.HandlInst{1,1} = '1';
order.MsgType{1,1} = 'D';
order.OrderQty{1,1} = '1000';
order.HeaderFields{1,1} = {'OnBehalfOfCompID','TRADER'};
order.BodyFields{1,1} = {'NoPartyIDs','3'; ...
                         'PartyID','1'; ...
                         'PartyRole','BBVA'; ...
                         'PartyID','1'; ...
                         'PartyRole','CVGX'; ...
                         'PartyID','1'; ...
                         'PartyRole','GSAM'};
```

```
order.BeginString{2,1} = 'FIX.4.4';
order.CLOrdId{2,1} = '339';
order.Side{2,1} = '2';
order.TransactTime{2,1} = datestr(now);
order.OrdType{2,1} = 'D';
order.Symbol{2,1} = 'IBM';
order.HandlInst{2,1} = '1';
order.MsgType{2,1} = 'D';
order.OrderQty{2,1} = '1000';
order.HeaderFields{2,1} = {'OnBehalfOfCompID','TRADER'};
order.BodyFields{2,1} = {'NoPartyIDs','3'; ...
                         'PartyID','1'; ...
                         'PartyRole','BBVA'; ...
                         'PartyID','1'; ...
                         'PartyRole','CVGX'; ...
                         'PartyID','1'; ...
                         'PartyRole','GSAM'};
```

Send FIX Messages

Use the FIX Flyer Engine connection to send the FIX messages. `status` contains a logical zero for a successful message delivery.

```
status = sendMessage(c,order);
```

Return Order Information

Return and display the order information `o` for all orders. The Variables editor displays the contents of `o`.

```
o = orderInfo(c);
openvar('o')
```

Replace an order. Create a FIX message `replace` with an updated quantity of 3378 shares. Set the field `MsgType` to `'G'` to specify a replace order.

```
replace.BeginString{1,1} = 'FIX.4.4';
replace.CLOrdId{1,1} = '338_REPLACE';
replace.origClOrdId{1,1} = '338';
replace.Symbol{1,1} = 'IBM';
replace.OnBehalfOfCompID{1,1} = 'TRADER';
replace.OrdType{1,1} = 'D';
replace.OrderQty{1,1} = '3378';
replace.MsgType{1,1} = 'G';
replace.Text{1,1} = 'REST API REPLACE';
```

Send the FIX message. To see the replaced order, retrieve and display the order information. The Variables editor displays the contents of o.

```
status = sendMessage(c,replace);

o = orderInfo(c);
openvar('o')
```

Now, cancel the order. Create a FIX message cancel with order number 338. Set the field MsgType to 'F' to specify a cancel order.

```
cancel.BeginString{1,1} = 'FIX.4.4';
cancel.CLOrdId{1,1} = '338_CANCEL';
cancel.origClOrdId{1,1} = '338_REPLACE';
cancel.Symbol{1,1} = 'IBM';
cancel.OnBehalfOfCompID{1,1} = 'TRADER';
cancel.OrdType{1,1} = 'D';
cancel.MsgType{1,1} = 'F';
cancel.Text{1,1} = 'REST API CANCEL';
```

Send the FIX message. Then retrieve and display the canceled order information. The Variables editor displays the contents of o.

```
status = sendMessage(c,cancel);

o = orderInfo(c);
openvar('o')
```

Receive FIX Message

Use the sample event handling listener fixExampleListener to listen for FIX messages from the FIX Flyer Engine. The listener fixExampleListener returns the raw FIX message in the table fixResponse. Display the first three columns of the table. The column names of fixResponse contain FIX tag names from the returned raw FIX message. The data in the columns contain the values of the returned raw FIX message.

```
fixResponse(:,1:3)

ans =

    BeginString     BodyLength     MsgType
    _____     _____     _____

    'FIX.4.4'       '219'          '8'
```

Close FIX Flyer Connection

```
close(c)
```

See Also
addListener | close | fixflyer | orderInfo | sendMessage

External Websites
·   FIX Trading Community

Writing and Running Custom Event Handler Functions with Bloomberg EMSX

In this section...
"Write a Custom Event Handler Function" on page 1-25
"Run a Custom Event Handler Function" on page 1-25
"Workflow for Custom Event Handler Function" on page 1-26

Write a Custom Event Handler Function

You can process events related to any Bloomberg EMSX orders and routes by writing a custom event handler function to use with Trading Toolbox. For example, you can plot the changes in the number of shares routed. Follow these tasks to write a custom event handler.

1   Choose the events that you want to process, monitor, or evaluate.

2   Decide how the custom event handler function processes these events.

3   Determine the input and output arguments for the custom event handler function.

4   Write the code for the custom event handler function.

For details, see "Create Functions in Files" (MATLAB). For a code example of an event handler function, see the function `processEventToBlotter` in the `emsxOrderBlotter.m` file.

Run a Custom Event Handler Function

You can run the custom event handler function by using `timer`. Specify the custom event handler function name as a function handle and pass this function handle as an input argument to `timer`. For details about function handles, see "Create Function Handle" (MATLAB). For example, suppose you want to create an order using `createOrderAndRoute` with the custom event handler function named `eventhandler`. This code assumes a Bloomberg EMSX connection `c`, Bloomberg EMSX order `order`, and timer object `t`.

1   Run `timer` to execute `eventhandler`. The name-value pair argument `TimerFcn` specifies the event handler function. The name-value pair argument `Period`

specifies a 1-second delay between executions of the event handler function. When the name-value pair argument `ExecutionMode` is set to `fixedRate`, the event handler function executes immediately after it is added to the MATLAB execution queue.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate');
```

2 Start the timer to initiate and execute `eventhandler` immediately.

```
start(t)
```

3 Run `createOrderAndRoute` using the custom event handler by setting `useDefaultEventHandler` to `false`.

```
createOrderAndRoute(c,order,'useDefaultEventHandler',false)
```

4 Stop the timer to stop data updates.

```
stop(t)
```

If you want to resume data updates, run `start`.

5 Delete the timer once you are done with processing data updates for the Bloomberg EMSX connection.

```
delete(t)
```

Workflow for Custom Event Handler Function

This workflow summarizes the tasks to work with a custom event handler function using Bloomberg EMSX.

1 Write a custom event handler function and save it to a file.

2 Create a connection using `emsx`.

3 Subscribe to Bloomberg EMSX fields using `orders` and `routes`. You can also write custom event handler functions to process subscription events.

4 Run the custom event handler function using timer. Use a function handle to specify the custom event handler function name to run `timer`.

5 Start the timer to execute the custom event handler function immediately using `start`.

6 Stop data updates using `stop`.

7 Unsubscribe from Bloomberg EMSX fields by using the API syntax.

8   Delete the timer using `delete`.

9   Close the connection using `close`.

See Also

timer | `close` | `createOrderAndRoute` | `delete` | `emsx` | `orders` | `routes` | `start` | `stop`

Related Examples

- "Create Functions in Files" (MATLAB)

More About

- "Create Function Handle" (MATLAB)

Writing and Running Custom Event Handler Functions with Interactive Brokers

In this section...
"Write a Custom Event Handler Function" on page 1-28
"Run a Custom Event Handler Function" on page 1-28
"Workflow for Custom Event Handler Function" on page 1-29

Write a Custom Event Handler Function

You can process events related to any Interactive Brokers data updates by writing a custom event handler function to use with Trading Toolbox. For example, you can request data about all open orders or retrieve account information. Follow these tasks to write a custom event handler.

1 Choose the events that you want to process, monitor, or evaluate.
2 Decide how the custom event handler function processes these events.
3 Determine the input and output arguments for the custom event handler function.
4 Write the code for the custom event handler function.

For details, see "Create Functions in Files" (MATLAB). For a code example of an Interactive Brokers event handler function, see `ibExampleEventHandler.m`.

Run a Custom Event Handler Function

You can run the custom event handler function by passing the function name as an input argument into an existing function. Specify the custom event handler function name as a character vector or function handle. For details about function handles, see "Create Function Handle" (MATLAB).

For example, suppose you want to retrieve real-time data from Interactive Brokers using `realtime` with the custom event handler function named `eventhandler`. You can use either of these syntaxes to run `eventhandler`. This code assumes a IB Trader Workstation connection `ib`, IB Trader Workstation `IContract` object `ibContract`, and Interactive Brokers fields `f`.

Use a character vector.

```
tickerid = realtime(ib,ibContract,f,'eventhandler');
```

Or, use a function handle.

```
tickerid = realtime(ib,ibContract,f,@eventhandler);
```

Workflow for Custom Event Handler Function

This workflow summarizes the tasks to work with a custom event handler function using Interactive Brokers.

1 Write a custom event handler function and save it to a file.

2 Create a connection to the IB Trader Workstation using `ibtws`.

3 Run an existing function to receive data updates. Use the custom event handler function as an input argument.

Caution: To run default event handler and sample event handler functions, you must run one event handler function at a time. After you run one event handler, close the IB Trader Workstation connection. Then, create another connection to run a different event handler with the same existing function. Otherwise, MATLAB assigns multiple existing functions to events and errors occur.

---

4 Close the connection to the IB Trader Workstation using `close`.

See Also
`close` | `ibtws` | `realtime`

More About
· "Create Functions in Files" (MATLAB)
· "Create Function Handle" (MATLAB)

2

Workflow Models

Workflow for Bloomberg EMSX

The workflow for Bloomberg EMSX is versatile with many options for alternate flows to create, route, and manage the status of an open order until it is filled.

1   Connect to Bloomberg EMSX using `emsx`.

2   Set up a subscription for orders and routes to obtain events on subsequent requests using `orders` and `routes`.

3   Create a Bloomberg EMSX order. Options in the flow of creating an order are:

- Create an order using `createOrder`.
- Route an order using `routeOrder`.
- Route an order with a strategy using `routeOrderWithStrat`.
- Route multiple orders with a strategy using `groupRouteOrderWithStrat`.
- Create an order and route using `createOrderAndRoute`.
- Create an order and route with a strategy using `createOrderAndRouteWithStrat`.

4   Modify an order or route using these functions:

- Modify an order using `modifyOrder`.
- Modify a route using `modifyRoute`.
- Modify a route with a strategy using `modifyRouteWithStrat`.

5   Delete an order or route using these functions:

- Delete an order using `deleteOrder`.
- Delete a route using `deleteRoute`.

6   Obtain information from Bloomberg EMSX using these functions:

- Obtain broker information using `getBrokerInfo`.
- Obtain Bloomberg EMSX field information using `getAllFieldMetaData`.

7   Explore information about existing orders and routes using these functions:

- View order transactions with a sample order blotter using `emsxOrderBlotter`.
- Process the current contents of the event queue using `processEvent`.

8   Close the Bloomberg EMSX connection using `close`.

Related Examples

Workflows for Trading Technologies X_TRADER

You can use X_TRADER to monitor market price information and submit orders.

To monitor market price information:

1 Connect to Trading Technologies X_TRADER using `xtrdr`.

2 Create an event notifier using `createNotifier`.

3 Create an instrument and attach it to the notifier using `createInstrument`. Optionally, use `getData` to return information on the instrument that you have created.

4 Close the Trading Technologies X_TRADER connection using `close`.

To submit orders to X_TRADER:

1 Connect to Trading Technologies X_TRADER using `xtrdr`.

2 Create an event notifier using `createNotifier`.

3 Create an instrument and attach it to the notifier using `createInstrument`. Optionally, use `getData` to return information on the instrument that you have created.

4 Create an order set using `createOrderSet` to define the level of the order status events and event handlers for orders that will be submitted to X_TRADER.

5 Define the order using `createOrderProfile`. An order profile contains the settings that define an individual order to be submitted.

6 Route the order for execution using the `OrderSet` object created by `createOrderSet` in step 4.

7 Close the Trading Technologies X_TRADER connection using `close`.

To monitor market price information and respond to market changes by automatically submitting orders to X_TRADER:

1 Connect to Trading Technologies X_TRADER using `xtrdr`.

2 Create an event notifier using `createNotifier`.

3 Create an instrument and attach it to the notifier using `createInstrument`. Use `getData` to return information on the instrument that you have created.

4 Define events by assigning callbacks for validating or invalidating an instrument and performing calculations based on the event. Based on some predefined condition

reached when changes in the incoming data satisfy the condition, event callbacks execute the functions in steps 5, 6, and 7.

5   Create an order set using `createOrderSet` to define the level of the order status events and event handlers for orders that will be submitted to X_TRADER.

6   Define the order using `createOrderProfile`. An order profile contains the settings that define an individual order to be submitted.

7   Route the order for execution using the `OrderSet` object created by `createOrderSet` in step 5.

8   Close the Trading Technologies X_TRADER connection using `close`.

Related Examples

- "Create an Order Using X_TRADER" on page 1-17
- "Listen for X_TRADER Price Updates" on page 4-2
- "Listen for X_TRADER Price Market Depth Updates" on page 4-4
- "Submit X_TRADER Orders" on page 4-8

Workflow for Interactive Brokers

In this section...
"Request Interactive Brokers Market Data" on page 2-6
"Create Interactive Brokers Orders" on page 2-7
"Request Interactive Brokers Informational Data" on page 2-7

This diagram shows the functions that you can use with the IB Trader Workstation to monitor market price information and submit orders.
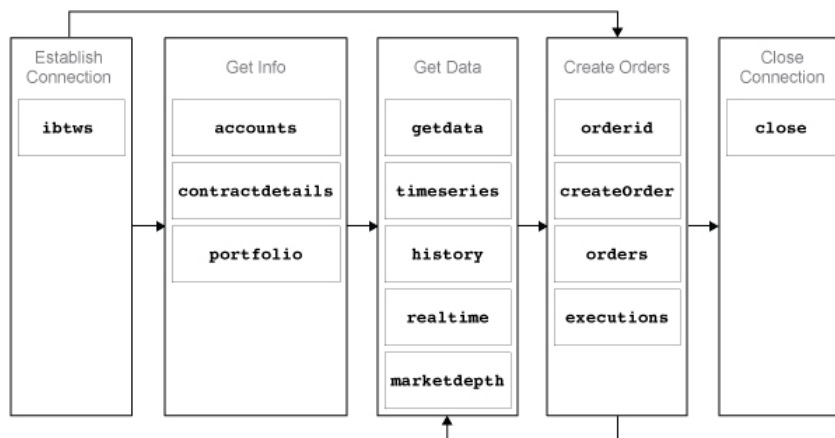
Request Interactive Brokers Market Data

To request current, intraday, real-time, historical, or market depth data:

1   Connect to the IB Trader Workstation using `ibtws`.

2   Create the IB Trader Workstation `IContract` object.

3   Request current data for a security using `getdata`.

4   Request intraday data for a security using `timeseries`.

5   Request real-time data for a security using `realtime`.

6   Request historical data for a security using `history`.

7   Request market depth data for a security using `marketdepth`.

8   Close the IB Trader Workstation connection using `close`.

Create Interactive Brokers Orders

To submit orders to the IB Trader Workstation:

1   Connect to the IB Trader Workstation using `ibtws`.

2   Create the IB Trader Workstation `IContract` object.

3   Create the IB Trader Workstation `IOrder` object.

4   Request a unique order identifier using `orderid`.

5   Create and submit the order using `createOrder`.

6   Request open order data using `orders`.

7   Request executed order data using `executions`.

8   Close the IB Trader Workstation connection using `close`.

Request Interactive Brokers Informational Data

To request information from the IB Trader Workstation:

1   Connect to the IB Trader Workstation using `ibtws`.

2   Create the IB Trader Workstation `IContract` object.

3   Request contract detailed data using `contractdetails`.

4   Request account information using `accounts`.

5   Request portfolio data using `portfolio`.

6   Close the IB Trader Workstation connection using `close`.

Related Examples

Workflow for CQG

This diagram shows the functions you can use with CQG to monitor market price information and submit orders.
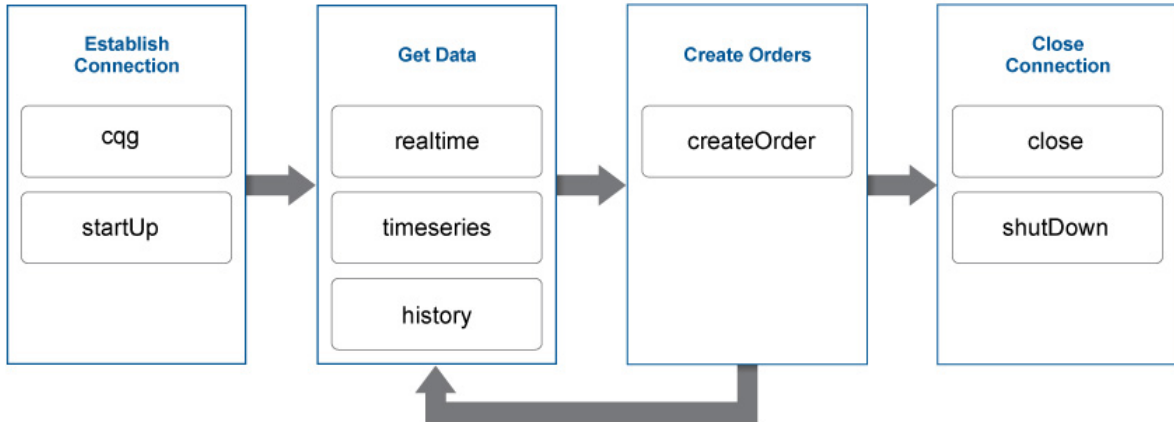
To request current, intraday, or historical data:

1 Create the CQG connection object using `cqg`.

2 Define the CQG event handlers.

3 Connect to CQG using `startUp`.

4 Subscribe to a CQG instrument to request real-time data using `realtime`.

5 Request intraday data for a security using `timeseries`.

6 Request historical data for a security using `history`.

7 Close the CQG connection using `close` or `shutDown`.

To submit orders to CQG:

1 Create the CQG connection object using `cqg`.

2 Define the CQG event handlers.

3 Connect to CQG using `startUp`.

4 Create the CQG account credentials object.

5 Subscribe to a CQG instrument to request real-time data using `realtime`.

6 Create and submit the order using `createOrder`.

7 Close the CQG connection using `close` or `shutDown`.

Related Examples

3

Transaction Cost Analysis

Analyze Trading Execution Results

This example shows how to conduct post-trade analysis using transaction cost analysis from the Kissell Research Group. Post-trade analysis includes implementation shortfall, alpha capture, benchmark costs, broker value add, and Z-Score. For details about these metrics, see "Post-Trade Analysis Metrics Definitions" on page 3-6. You can use post-trade analysis to evaluate portfolio returns and profits. You can measure performance of brokers and algorithms.

To access the example code, enter `edit KRGPostTradeAnalysisExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Transaction Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Load the example data `PostTradeData` from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData.mat PostTradeData
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Determine Implementation Shortfall Costs

Determine the components of the implementation shortfall costs in basis points. The components are:

- Fixed cost `ISFixed`
- Delay cost `ISDelayCost`
- Execution cost `ISExecutionCost`
- Opportunity cost `ISOpportunityCost`

For details about the cost components, see "Post-Trade Analysis Metrics Definitions" on page 3-6.

```
PostTradeData.ISDollars = ...
    PostTradeData.OrderShares .* PostTradeData.ISDecisionPrice;
PostTradeData.ISFixed = ...
    PostTradeData.ISFixedDollars ./ PostTradeData.ISDollars*10000;
PostTradeData.ISDelayCost = ...
    PostTradeData.OrderShares .* ...
    (PostTradeData.ISArrivalPrice-PostTradeData.ISDecisionPrice).* ...
    PostTradeData.SideIndicator ./ PostTradeData.ISDollars*1000;
PostTradeData.ISExecutionCost = ...
    PostTradeData.TradedShares .* ...
    (PostTradeData.AvgExecPrice-PostTradeData.ISArrivalPrice).* ...
    PostTradeData.SideIndicator ./ PostTradeData.ISDollars*1000;
PostTradeData.ISOpportunityCost = ...
    (PostTradeData.OrderShares-PostTradeData.TradedShares).* ...
    (PostTradeData.ISEndPrice-PostTradeData.ISArrivalPrice).* ...
    PostTradeData.SideIndicator ./ PostTradeData.ISDollars*1000;
```

Determine the total implementation shortfall cost `ISCost`.

```
PostTradeData.ISCost = PostTradeData.ISFixed + ...
    PostTradeData.ISDelayCost + PostTradeData.ISExecutionCost + ...
    PostTradeData.ISOpportunityCost;
```

Determine Profit

Determine the alpha capture `Alpha_CapturePct`. Divide realized profit `Alpha_Realized` by potential profit `Alpha_TotalPeriod`.

```
PostTradeData.Alpha_Realized = ...
    (PostTradeData.ISEndPrice-PostTradeData.AvgExecPrice).* ...
    PostTradeData.TradedShares .* PostTradeData.SideIndicator ./ ...
    (PostTradeData.TradedShares .* PostTradeData.ISArrivalPrice)*10000;
PostTradeData.Alpha_TotalPeriod = ...
    (PostTradeData.ISEndPrice-PostTradeData.ISArrivalPrice).* ...
    PostTradeData.TradedShares .* PostTradeData.SideIndicator ./ ...
    (PostTradeData.TradedShares .* PostTradeData.ISArrivalPrice)*10000;
```

```matlab
lenAlpha_Realized = length(PostTradeData.Alpha_Realized);
PostTradeData.Alpha_CapturePct = zeros(lenAlpha_Realized,1);
for ii = 1:lenAlpha_Realized
    if PostTradeData.Alpha_TotalPeriod(ii) > 0
        PostTradeData.Alpha_CapturePct(ii) = ...
        PostTradeData.Alpha_Realized(ii) ./ ...
        PostTradeData.Alpha_TotalPeriod(ii);
    else
        PostTradeData.Alpha_CapturePct(ii) = ...
        -(PostTradeData.Alpha_Realized(ii) - ...
        PostTradeData.Alpha_TotalPeriod(ii)) ./ ...
        PostTradeData.Alpha_TotalPeriod(ii);
    end
end
```

Determine Benchmark and Trading Costs

Determine benchmark costs in basis points. Here, the benchmark prices are:

- Close price of the previous day `PrevClose_Cost`
- Open price `Open_Cost`
- Close price `Close_Cost`
- Arrival cost `Arrival_Cost`
- Period VWAP `PeriodVWAP_Cost`

```matlab
PostTradeData.PrevClose_Cost = ...
    (PostTradeData.AvgExecPrice-PostTradeData.PrevClose).* ...
    PostTradeData.SideIndicator ./ PostTradeData.PrevClose*10000;
PostTradeData.Open_Cost = ...
    (PostTradeData.AvgExecPrice-PostTradeData.Open).* ...
    PostTradeData.SideIndicator ./ PostTradeData.Open*10000;
PostTradeData.Close_Cost = (PostTradeData.AvgExecPrice-PostTradeData.Close).* ...
    PostTradeData.SideIndicator ./ PostTradeData.Close*10000;
PostTradeData.Arrival_Cost = (PostTradeData.AvgExecPrice- ...
    PostTradeData.ArrivalPrice).* ...
    PostTradeData.SideIndicator ./ PostTradeData.ArrivalPrice*10000;
PostTradeData.PeriodVWAP_Cost = (PostTradeData.AvgExecPrice- ...
    PostTradeData.PeriodVWAP).* ...
    PostTradeData.SideIndicator ./ PostTradeData.PeriodVWAP*10000;
```

Estimate market-impact `miCost` and timing risk `tr` costs.

```matlab
PostTradeData.Size = PostTradeData.TradedShares ./ PostTradeData.ADV;
```

```
PostTradeData.Price = PostTradeData.ArrivalPrice;
PostTradeData.miCost = marketImpact(k,PostTradeData);
PostTradeData.tr = timingRisk(k,PostTradeData);
```

Determine Broker Value Add and Z-Score

Determine the broker value add using the arrival cost and market impact.

```
PostTradeData.ValueAdd = (PostTradeData.Arrival_Cost-PostTradeData.miCost) * -1;
```

Determine the Z-Score using the broker value add and timing risk.

```
PostTradeData.zScore = PostTradeData.ValueAdd./PostTradeData.tr;
```

For details about the preceding calculations, contact the Kissell Research Group.

See Also
krg | marketImpact | timingRisk

More About
- "Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22
- "Post-Trade Analysis Metrics Definitions" on page 3-6
- "Kissell Research Group Data Sets" on page 3-9

Post-Trade Analysis Metrics Definitions

In this section...
"Implementation Shortfall" on page 3-6
"Alpha Capture" on page 3-7
"Benchmark Costs" on page 3-7
"Broker Value Add" on page 3-7
"Z-Score" on page 3-7

After executing a transaction, Kissell Research Group provides various metrics for analyzing the results of a transaction. For an example using these metrics, see "Analyze Trading Execution Results" on page 3-2.

For details about these calculations, contact the Kissell Research Group.

Implementation Shortfall

Implementation shortfall (IS) determines the total cost of implementing an investment decision. IS subtracts the actual return from the paper return of a stock or portfolio after including all visible costs including commissions, fees, and taxes. The Kissell Research Group IS cost formula decomposes costs into fixed, delay, execution, and opportunity cost components.

IS Component	Description
Fixed cost	Cost component that is not dependent upon the implementation strategy.
Delay cost	Cost component that represents the loss in investment value between the time the managers make the investment decision and the order releases to the market.
Execution cost	Cost component that is the difference between the execution price and the stock price at the time the order releases to the market.

IS Component	Description
Opportunity cost	Cost component that represents the foregone profit or loss resulting from not being able to execute the order to completion within the allotted time period.

Portfolio managers and traders use IS to understand the trading cost environment.

Alpha Capture

Alpha capture, or profit, is the realized profit divided by the potential profit. Realized profit is based on the difference between end price and average execution price. Potential profit is based on the difference between end price and arrival price. Portfolio managers and traders use alpha capture to measure portfolio performance.

Benchmark Costs

The benchmark cost compares the average execution price to a specific benchmark price. A benchmark price can be any price such as the close price. Traders use benchmark costs to measure strategy and transaction performance.

Broker Value Add

The broker value add metric is a measure of the overall broker performance. A positive value indicates that the broker performed better than expected and a negative value indicates the broker under-performed expectations. This metric is the difference between the estimated trading cost and the actual cost incurred by the investor. You can estimate trading costs using `marketImpact`, `priceAppreciation`, and `timingRisk`. This metric reflects performance given all market conditions on the day and buying and selling behavior from all other participants.

Traders use this metric to measure broker performance.

Z-Score

Z-Score is the broker value add metric divided by timing risk. You can estimate timing risk using `timingRisk`. The Z-Score specifies the number of standard deviations away from the estimated cost. If the Z-Score is greater than or equal to two standard deviations, then the actual cost varies greatly from the estimated cost.

Traders use this metric to measure broker performance.

References

[1] Kissell, Robert. "The Expanded Implementation Shortfall: Understanding Transaction Cost Components." *Journal of Trading*. Vol. 1, Number 3, Summer 2006, pp. 6–16.

Related Examples

- "Analyze Trading Execution Results" on page 3-2

Kissell Research Group Data Sets

The following descriptions define the data sets provided in the file
`KRGExampleData.mat`.

`Basket` Variables

The table `Basket` contains a trade list for a collection of stocks in a portfolio. For
examples of using this data set, see "Rank Broker Performance" on page 3-73.

Real trade lists come from portfolio managers.

Table Variable	Description
Symbols	Stock symbol.
Side	Side (`'B'` or `'S'`).
Size	Size (number of shares divided by average daily volume).
Shares	Number of shares.
Price	Stock price.
ADV	Average daily volume.
Volatility	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
POV	Percentage of volume.

`BrokerNames` Variables

The table `BrokerNames` contains the broker names and their associated market-impact
code. For examples of using this data set, see "Rank Broker Performance" on page
3-73.

Real trade lists come from portfolio managers.

Table Variable	Description
Broker	Broker name.
MICode	Market-impact code (1, 2, 3, and so on).

TradeData Variables

The table TradeData provides example data for a collection of stocks in a transaction. For examples of using this data set, see "Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22 and "Estimate Portfolio Liquidation Costs" on page 3-26.

Real market data comes from a data source such as Bloomberg.

Table Variable	Description
Symbol	Stock symbol.
Side	Side ('Buy' or 'Sell').
SideIndicator	Side indicator. 1 is a buy (add shares to portfolio). -1 is a sell (remove shares from portfolio).
AvgExecPrice	Average execution price.
ArrivalPrice	Arrival price. The price at the time the order enters the market.
PeriodVWAP	Volume weighted average price (VWAP). The VWAP compares the execution price to the interval VWAP price.
CCYRate	Currency rate.
Volatility	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
POV	Percentage of volume.

Table Variable	Description
SectorCategory	Market sector category ('Energy', 'Industrials', 'Materials', and so on).
OrderSizeCategory	Order size category ('Large', 'Medium', or 'Small').
VolatilityCategory	Volatility category ('High', 'Medium', or 'Low').
POVRateCategory	Percentage of volume rate category ('Aggressive', 'Passive', or 'Normal').
MktCapCategory	Market capitalization category ('LC' is large cap, 'MC' is mid cap, 'SM' is small cap).
MomentumCategory	Momentum category ('Favorable', 'Neutral', or 'Adverse').
MktMovementCategory	Market movement category ('Favorable', 'Neutral', or 'Adverse').
ADV	Average daily volume.
Price	Stock price.
Size	Size (number of shares divided by average daily volume).
Alpha_bp	Alpha estimate per day in basis points.
Shares	Number of shares.
Broker	Broker name.

TradeDataCurrent and TradeDataHistorical Variables

The tables TradeDataCurrent and TradeDataHistorical provide example current and historical data, respectively, for a collection of stocks in a transaction. For an example of using this data set, see "Determine Buy-Sell Imbalance Using Cost Index" on page 3-67.

Real market data comes from a data source such as Bloomberg.

Table Variable	Description
Symbol	Stock symbol.
Date	Transaction date.
MICode	Market-impact code (1, 2, 3, and so on).
Open	Stock open price.
VWAP	Volume weighted average price (VWAP).
Last	Stock last price.
Volume	Trade volume.
Volatility	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
ADV	Average daily volume.
Beta	Beta.
IndexOpen	Index open price.
IndexVWAP	Index VWAP.
IndexLast	Index last price.
Price	Stock price.
POV	Percentage of volume.
Shares	Number of shares.

PortfolioData Variables

The table PortfolioData provides example data for a collection of stocks in a portfolio. To use this data set, see portfolioCostCurves.

Real portfolio data comes from a portfolio belonging to a company or portfolio manager.

Table Variable	Description
Symbol	Stock symbol.

Table Variable	Description
Price_Local	Local price of the stock.
Price_Currency	Stock price with a specified base currency if the stock trades outside the United States. If the stock trades in the United States, Price_Currency has the same value as Price_Local.
ADV	Average daily volume.
Volatility	Volatility.
Shares	Number of shares.

PostTradeData Variables

The table PostTradeData provides example data for a collection of stocks in an executed transaction. To use this data set, see "Analyze Trading Execution Results" on page 3-2.

Real market data comes from a data source such as Bloomberg.

Table Variable	Description
Symbol	Stock symbol.
Side	Side ('Buy' or 'Sell').
SideIndicator	Side indicator. 1 is a buy (add shares to portfolio). -1 is a sell (remove shares from portfolio).
Date	Transaction date.
DecisionTime	Decision time. The portfolio manager decides to buy, sell, short, or cover a position at this time. If no other timestamp is available, set this variable to the time when the portfolio manager enters the order into the trading system. If the portfolio manager does not have a timestamp for this decision, investors use the close time of the previous day, open time, or arrival time.

Table Variable	Description
ArrivalTime	Arrival time. The trading system enters the order into the market for execution at this time. You can obtain it from the first trade from the electronic audit trail.
EndTime	End time. The portfolio manager specifies to complete the order at this time. Typically, this time is the end of the day or the time of the last trade.
AvgExecPrice	Average executed price.
OrderShares	Number of shares.
TradedShares	Number of shares executed.
Volatility	Volatility.
ADV	Average daily volume.
POV	Percentage of volume.
CCYRate	Currency rate.
MICategory	Market-impact category (for example, 1).
PrevClose	Close price of the previous day.
Open	Open price.
Close	Close price.
ArrivalPrice	Arrival price. The price at the time the order enters the market.
PeriodVWAP	Volume weighted average price (VWAP). The VWAP compares the execution price to the interval VWAP price.
Broker	Broker name.
Algorithm	Trading algorithm ('Dark Pool', 'TWAP', 'Arrival', and so on).
Manager	Portfolio manager name.
Trader	Trader name.

Table Variable	Description
SectorCategory	Market sector category ('Energy', 'Industrials', 'Materials', and so on).
OrderSizeCategory	Order size category ('Large', 'Medium', or 'Small').
VolatilityCategory	Volatility category ('High', 'Medium', or 'Low').
POVRateCategory	Percentage of volume rate category ('Aggressive', 'Passive', or 'Normal').
MktCapCategory	Market capitalization category ('LC' is large cap, 'MC' is mid cap, 'SM' is small cap).
StockMomentumCategory	Stock momentum category ('Favorable', 'Neutral', or 'Adverse').
MktMovementCategory	Market movement category ('Favorable', 'Neutral', or 'Adverse').
StepOut	Investor field designation. This variable is optional for grouping and summary analysis. This field refers to a process where a broker (broker 1) receives an order from a client. Then this broker gives that order to another broker (broker 2) for its execution. Broker 1 receives credit for the trade but its performance applies to broker 2 who executed the trade.
ISDecisionPrice	Decision price. This variable is the stock price when the portfolio manager decides to buy, sell, short, or cover a position.
ISArrivalPrice	Midpoint of the bid-ask spread at the time an order enters the market.
ISEndPrice	End price. This variable is the stock price at the specified end time of the order.

Table Variable	Description
ISFixedDollars	Fixed fees in dollars that include the commission, taxes, clearing and settlement charges, and so on.

TradeDataBackTest Variables

The table TradeDataBackTest provides example data for a set of stocks and a series of dates. The data contains historical trade information for each stock. To use this data set, see "Conduct Back Test on Portfolio" on page 3-46.

Real market data comes from a data source such as Bloomberg.

Table Variable	Description
Symbol	Stock symbol.
Date	Historical transaction date.
Shares	Number of shares.
Side	Side ('Buy' or 'Sell').
Value	Dollar value of the stock in the portfolio.
Price	Stock price.
Size	Size (number of shares divided by average daily volume).
EstReturn	Estimated return decimal value for the stock in the portfolio.
Volatility	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
ADV	Average daily volume.
MktCap	Market capitalization.
TradeTime	Trade duration time.

Table Variable	Description
POVRate	Percentage of volume rate.
MICode	Market-impact code (1, 2, 3, and so on).
FXRate	Foreign exchange rate.
POV	Percentage of volume.

TradeDataStressTest Variables

The table TradeDataStressTest provides example data for a set of stocks for a date range. The data contains trade information for each stock. To use this data set, see "Conduct Stress Test on Portfolio" on page 3-50.

Real market data comes from a data source such as Bloomberg.

Table Variable	Description
Symbol	Stock symbol.
Date	Historical transaction date.
Shares	Number of shares.
Side	Side ('Buy' or 'Sell').
Value	Dollar value of the stock in the portfolio.
Price	Stock price.
Size	Size (number of shares divided by average daily volume).
EstReturn	Estimated return decimal value for the stock in the portfolio.
Volatility	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
ADV	Average daily volume.
MktCap	Market capitalization.

Table Variable	Description
TradeTime	Trade duration time.
POVRate	Percentage of volume rate.
MICode	Market-impact code (1, 2, 3, and so on).
FXRate	Foreign exchange rate.

TradeDataPortOpt Variables

The table TradeDataPortOpt contains example data for a collection of stocks in a portfolio. This data contains lower and upper bounds for the constraints used in the portfolio optimization. To use this data set, see "Liquidate Dollar Value from Portfolio" on page 3-56.

To see the related covariance data for each stock in the portfolio, see the covariance data table CovarianceData.

Real portfolio data comes from a portfolio belonging to a company or portfolio manager.

Table Variable	Description
Symbol	Stock symbol.
Date	Transaction date.
Shares	Number of shares.
Value	Dollar value of the stock in the portfolio.
Price	Stock price.
Size	Size (number of shares divided by average daily volume).
EstReturn	Estimated return decimal value for the stock in the portfolio.
Volatility	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.

Table Variable	Description
ADV	Average daily volume.
MktCap	Market capitalization.
TradeTime	Trade time.
MICode	Market-impact code (1, 2, 3, and so on).
LB_Wt	Lower bound weight.
UB_Wt	Upper bound weight.
LB_MinShares	Lower bound for the minimum shares.
UB_MaxShares	Upper bound for the maximum shares.
LB_MinPctADV	Lower bound for the minimum percentage of average daily volume.
UB_MaxPctADV	Upper bound for the maximum percentage of average daily volume.
LB_MinValue	Lower bound for the minimum value.
UB_MaxValue	Upper bound for the maximum value.
UB_MaxMI	Upper bound for the maximum market-impact cost.

TradeDataTradeOpt Variables

The table TradeDataTradeOpt provides an example trade list for a collection of stocks in a portfolio. For an example of using this data set, see "Optimize Trade Schedule Trading Strategy for Basket" on page 3-80.

Real trade lists come from portfolio managers.

Table Variable	Description
Date	Transaction date.
Side	Side ('B' or 'S').
Shares	Number of shares.
Price	Stock price.
ADV	Average daily volume.

Table Variable	Description
Volatility	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
PctADV	Percentage of average daily volume.
Value	Transaction value.
Weight	Weight.
SideIndicator	Side indicator. 1 is a buy (add shares to portfolio). -1 is a sell (remove shares from portfolio).
MIRegion	Market-impact region.
Symbol	Stock symbol.
Alpha_bp	Alpha in basis points.
Beta	Beta.
Sector	Market sector, such as Energy.
MktCap	Market capitalization.

CovarianceData Table

The table CovarianceData contains a covariance value for all stocks in the portfolio data table TradeDataPortOpt. Each variable in the table is a different stock. To use this data set in the portfolio optimization, see "Liquidate Dollar Value from Portfolio" on page 3-56.

CovarianceTradeOpt Table

The table CovarianceTradeOpt contains a covariance value for each stock in the portfolio data table TradeDataTradeOpt. Each variable in the table is a different stock. To use this data set in the trade schedule optimization, see "Optimize Trade Schedule Trading Strategy for Basket" on page 3-80.

References

[1] Kissell, Robert. "A Practical Framework for Transaction Cost Analysis." *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. "The Expanded Implementation Shortfall: Understanding Transaction Cost Components." *Journal of Trading*. Vol. 1, Number 3, Summer 2006, pp. 6–16.

[3] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

Related Examples

- "Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22
- "Conduct Back Test on Portfolio" on page 3-46
- "Conduct Stress Test on Portfolio" on page 3-50
- "Estimate Portfolio Liquidation Costs" on page 3-26
- "Liquidate Dollar Value from Portfolio" on page 3-56
- "Analyze Trading Execution Results" on page 3-2

Conduct Sensitivity Analysis to Estimate Trading Costs

This example shows how to evaluate changes in trading costs due to liquidity, volatility, and market sensitivity to order flow and trades. With transaction cost analysis from the Kissell Research Group, you can simulate the trading cost environment for a collection of stocks. Sensitivity analysis enables you to estimate future trading costs for different market conditions to determine the appropriate portfolio contents that meet the needs of the investors.

Here, evaluate changes in trading costs due to decreasing average daily volume by 50% and doubling volatility. The example data uses the percentage of volume (POV) trade strategy.

To access the example code, enter `edit KRGSensitivityAnalysisExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Transaction Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData.mat
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Estimate Initial Trading Costs

Estimate initial trading costs using the example data `TradeData`. The trading costs are:

- Instantaneous trading cost `itc`
- Market-impact cost `mi`
- Timing risk `tr`
- Price appreciation `pa`

Group all four trading costs into a numeric matrix `initTCA`.

```
itc = iStar(k,TradeData);
mi = marketImpact(k,TradeData);
tr = timingRisk(k,TradeData);
pa = priceAppreciation(k,TradeData);
initTCA = [itc mi tr pa];
```

Create Scenario

Set variables to create the scenario. Here, the scenario decreases average daily volume by 50% and doubles volatility. The stock price, volume, estimated alpha, and trade strategy remain unchanged from the example data. You can modify the values of these variables to create different scenarios. The fields are:

- Average daily volume
- Volatility
- Stock price
- Volume
- Alpha estimate
- POV trade strategy
- Trade time trade strategy

```
adjADV = 0.5;
adjVolatility = 2.0;
adjPrice = 1.0;
adjVolume = 1.0;
adjAlpha = 1.0;
adjPOV = 1.0;
adjTradeTime = 1.0;
```

Adjust the example data based on the scenario variables.

```
TradeDataAdj = TradeData;
TradeDataAdj.Size = TradeData.Size .* (1./adjADV);
TradeDataAdj.ADV = TradeData.ADV .* adjADV;
TradeDataAdj.Volatility = TradeData.Volatility .* adjVolatility;
TradeDataAdj.Price = TradeData.Price .* adjPrice;
TradeDataAdj.Alpha_bp = TradeData.Alpha_bp .* adjAlpha;
```

`TradeDataAdj` contains the adjusted data. Size doubles because average daily volume decreases by 50%.

Convert POV trade strategy to the trade time trade strategy.

```
[~,povFlag,timeFlag] = krg.krgDataFlags(TradeData);
if povFlag
    TradeDataAdj.POV = TradeData.POV.*adjPOV;
    TradeDataAdj.TradeTime = TradeDataAdj.Size .* ...
        ((1-TradeDataAdj.POV) ./ TradeDataAdj.POV) .* (1./adjVolume);
elseif timeFlag
    TradeDataAdj.TradeTime = tradedata.TradeTime .* adjTradeTime;
    TradeDataAdj.POV = TradeDataAdj.Size ./ ...
        (TradeDataAdj.Size + TradeDataAdj.TradeTime .* adjVolume);
end
```

Estimate Trading Costs for Scenario

Estimate the trading costs based on the adjusted data. The numeric matrix `newTCA` contains the trading costs for the scenario.

```
itc = iStar(k,TradeDataAdj);
mi = marketImpact(k,TradeDataAdj);
tr = timingRisk(k,TradeDataAdj);
pa = priceAppreciation(k,TradeDataAdj);
newTCA = [itc mi tr pa];
```

Subtract the trading costs from the scenario from the initial trading costs.

```
rawWI = newTCA - initTCA;
wi = table(rawWI(:,1),rawWI(:,2),rawWI(:,3),rawWI(:,4), ...
    'VariableNames',{'ITC','MI','TR','PA'});
```

The table `wi` contains the full impact of this scenario on the trading costs.

Display trading costs for the first three rows in `wi`.

```
wi(1:3,:)
```

```
ans =

    ITC        MI         TR         PA

   _____      _____      _____      _____

    43.05      0.65     290.80      -9.49
   408.29    124.52     443.16       8.47
    80.92     13.79     114.97       0.93
```

The variables in `wi` are:

· Instantaneous trading cost

· Market-impact cost

· Timing risk

· Price appreciation

For details about the preceding calculations, contact the Kissell Research Group.

See Also

`iStar` | `krg` | `marketImpact` | `priceAppreciation` | `timingRisk`

More About

· "Analyze Trading Execution Results" on page 3-2

· "Kissell Research Group Data Sets" on page 3-9

Estimate Portfolio Liquidation Costs

This example shows how to determine the cost of liquidating individual stocks in a portfolio using transaction cost analysis from the Kissell Research Group. Compare the individual stocks in a portfolio using various metrics in a scatter plot.

The example data uses the percentage of volume trade strategy to calculate costs. You can also use the trade time trade strategy to run the analysis by replacing the percentage of volume data with trade time data.

To access the example code, enter `edit KRGPortfolioLiquidityExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Transaction Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Load the example data `TradeData` from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData.mat TradeData
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Estimate Trading Costs

Estimate market-impact costs `mi`.

```
TradeData.mi = marketImpact(k,TradeData);
```

Estimate the timing risk `tr`.

```
TradeData.tr = timingRisk(k,TradeData);
```

Estimate the liquidity factor `lf`.

```
TradeData.lf = liquidityFactor(k,TradeData);
```

For details about the preceding calculations, contact the Kissell Research Group.

Display Portfolio Plots

Create a scatter plot that shows the following:

- Size
- Volatility
- Market impact
- Timing risk
- Liquidity factor

```
figure
axOrder = subplot(2,3,1);
nSymbols = 1:length(TradeData.Size);
scatter(nSymbols,TradeData.Size*100,10,'filled')
grid on
box on
title(' Order Size (%ADV)')
axOrder.YAxis.TickLabelFormat = '%.1f%%';

axVolatility = subplot(2,3,2);
scatter(nSymbols,TradeData.Volatility*100,10,'filled')
grid on
box on
title('Volatility')
axVolatility.YAxis.TickLabelFormat = '%g%%';

axMI = subplot(2,3,4);
scatter(nSymbols,TradeData.mi,10,'filled')
grid on
box on
title('Market Impact (bp)')
axMI.YAxis.TickLabelFormat = '%.1f';
```

```
axTR = subplot(2,3,5);
scatter(nSymbols,TradeData.tr,10,'filled')
grid on
box on
title('Timing Risk (bp)')
axTR.YAxis.TickLabelFormat = '%.1f';

axLF = subplot(2,3,6);
scatter(nSymbols,TradeData.lf*100,10,'filled')
grid on
box on
title('Liquidity Factor')
axLF.YAxis.TickLabelFormat = '%.2f%%';
```
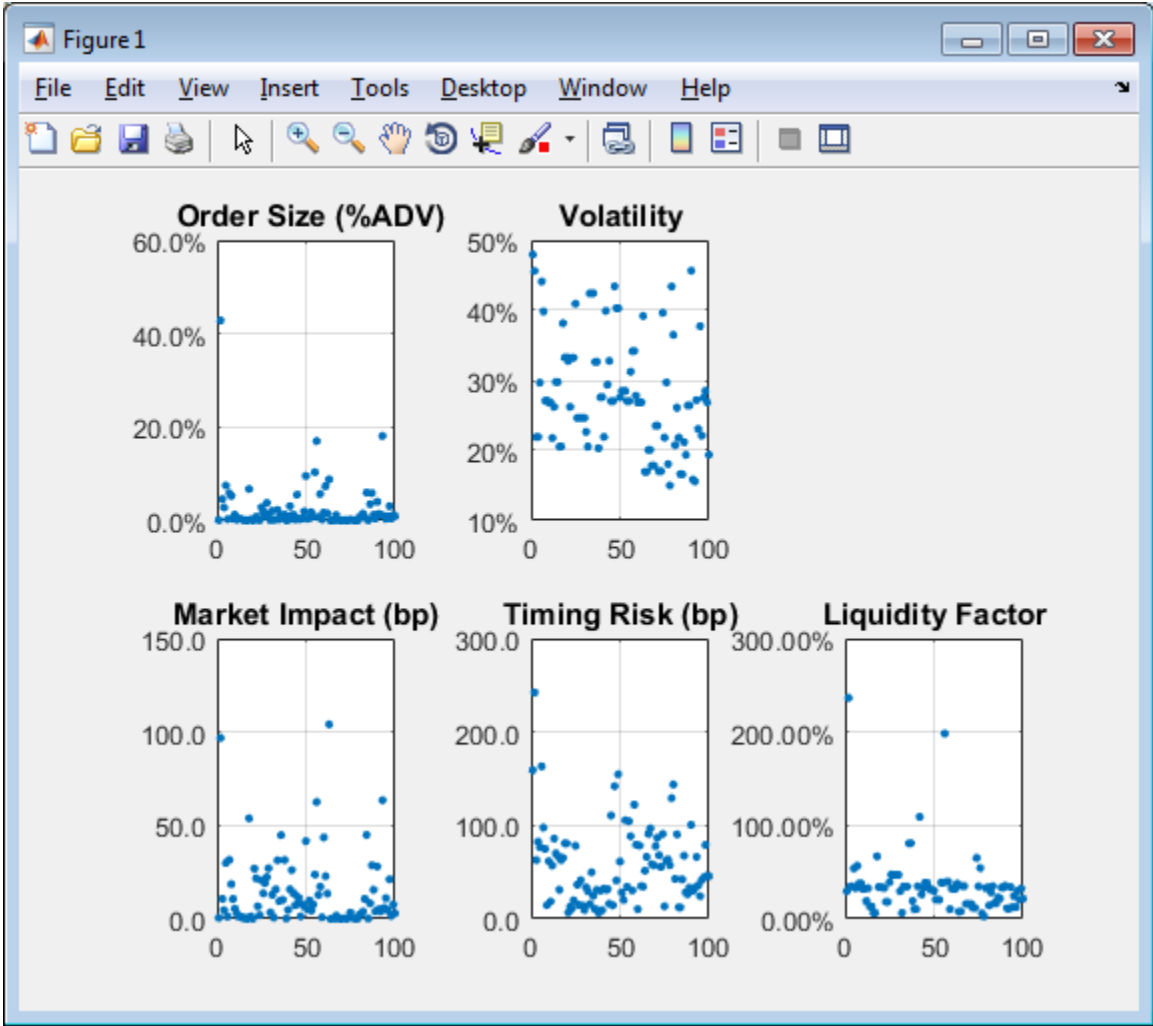
This figure demonstrates a snapshot view into the trading and liquidation costs, volatility, and size of the stocks in the portfolio. You can modify this scatter plot to include other variables from `TradeData`.

See Also

krg | liquidityFactor | marketImpact | timingRisk

More About

- "Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22
- "Kissell Research Group Data Sets" on page 3-9

Optimize Percentage of Volume Trading Strategy

This example shows how to optimize the strategy for a single stock by minimizing trading costs using transaction cost analysis from the Kissell Research Group. The optimization minimizes trading costs associated with the percentage of volume trading strategy and a specified risk aversion parameter *Lambda*. The trading cost minimization is expressed as

$$\min\big[(MI + PA) + Lambda \cdot TR\big],$$

where trading costs are market impact *MI*, price appreciation *PA*, and timing risk *TR*. For details, see `marketImpact`, `priceAppreciation`, and `timingRisk`. This example finds a local minimum for this expression. For details about searching for the global minimum, see "Troubleshooting and Tips" (MATLAB).

Here, you can optimize the percentage of volume trade strategy. To optimize trade time and trade schedule strategies, see "Optimize Trade Time Trading Strategy" on page 3-35 and "Optimize Trade Schedule Trading Strategy" on page 3-39.

To access the example code, enter `edit KRGSingleStockOptimizationExample.m` at the command line.

Retrieve Market-Impact Parameters and Create Example Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Create Single Stock Data

The structure `tradeData` contains data for a single stock. Use a structure or table to define this data. The fields are:

- Number of shares
- Average daily volume
- Volatility
- Stock price
- Initial percentage of volume trade strategy
- Alpha estimate

```
tradeData.Shares = 100000;
tradeData.ADV = 1000000;
tradeData.Volatility = 0.25;
tradeData.Price = 35;
tradeData.POV = 0.5;
tradeData.Alpha_bp = 50;
```

Define Optimization Parameters

Define risk aversion level `Lambda`. Set `Lambda` from 0 to `Inf`.

```
Lambda = 1;
```

Define lower `LB` and upper `UB` bounds of strategy input for optimization.

```
LB = 0;
UB = 1;
```

Define the function handle `fun` for the objective function. To access the code for this function, enter `edit krgSingleStockOptimizer.m`.

```
fun = @(pov)krgSingleStockOptimizer(pov,k,tradeData,Lambda);
```

Minimize Trading Costs for Trade Strategy

Minimize the trading costs for the percentage of volume trade strategy. `fminbnd` finds the optimal value for the percentage of volume trade strategy based on the lower and upper bound values. `fminbnd` finds a local minimum for the trading cost minimization expression.

```
[tradeData.POV,totalcost] = fminbnd(fun,LB,UB);
```

Display the optimized trade strategy `tradeData.POV`.

```
tradeData.POV

ans =

    0.35
```

Estimate Trading Costs for Optimized Strategy

Estimate the trading costs `povCosts` using the optimized trade strategy.

```
mi = marketImpact(k,tradeData);
pa = priceAppreciation(k,tradeData);
tr = timingRisk(k,tradeData);
povCosts = [totalcost mi pa tr];
```

Display trading costs.

```
povCosts

        100.04          56.15          4.63          39.27
```

The trading costs are:

- Total cost
- Market impact
- Price appreciation
- Timing risk

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. "Algorithmic Trading Strategies." Ph.D. Thesis. Fordham University, May 2006.

[2] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[3] Glantz, Morton, and Robert Kissell. *Multi-Asset Risk Modeling*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also
fminbnd | krg | marketImpact | priceAppreciation | timingRisk

Related Examples

- "Optimize Trade Time Trading Strategy" on page 3-35
- "Optimize Trade Schedule Trading Strategy" on page 3-39
- "Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22
- "Estimate Portfolio Liquidation Costs" on page 3-26

Optimize Trade Time Trading Strategy

This example shows how to optimize the strategy for a single stock by minimizing trading costs using transaction cost analysis from the Kissell Research Group. The optimization minimizes trading costs associated with the trade time trading strategy and a specified risk aversion parameter *Lambda*. The trading cost minimization is expressed as

$$\min\big[(MI+PA)+Lambda \cdot TR\big],$$

where trading costs are market impact *MI*, price appreciation *PA*, and timing risk *TR*. For details, see `marketImpact`, `priceAppreciation`, and `timingRisk`. This example finds a local minimum for this expression. For details about searching for the global minimum, see "Troubleshooting and Tips" (MATLAB).

Here, you can optimize the trade time trade strategy. To optimize percentage of volume and trade schedule strategies, see "Optimize Percentage of Volume Trading Strategy" on page 3-31 and "Optimize Trade Schedule Trading Strategy" on page 3-39.

To access the example code, enter `edit KRGSingleStockOptimizationExample.m` at the command line.

Retrieve Market-Impact Parameters and Create Example Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Create Single Stock Data

The structure `tradeData` contains data for a single stock. Use a structure or table to define this data. The fields are:

- Number of shares
- Average daily volume
- Volatility
- Stock price
- Initial trade time trade strategy
- Alpha estimate

```
tradeData.Shares = 100000;
tradeData.ADV = 1000000;
tradeData.Volatility = 0.25;
tradeData.Price = 35;
tradeData.TradeTime = 0.5;
tradeData.Alpha_bp = 50;
```

Define Optimization Parameters

Define risk aversion level `Lambda`. Set `Lambda` from 0 to `Inf`.

```
Lambda = 1;
```

Define lower `LB` and upper `UB` bounds of strategy input for optimization.

```
LB = 0;
UB = 1;
```

Define the function handle `fun` for the objective function. To access the code for this function, enter `edit krgSingleStockOptimizer.m`.

```
fun = @(tradetime)krgSingleStockOptimizer(tradetime,k,tradeData,Lambda);
```

Minimize Trading Costs for Trade Strategy

Minimize the trading costs for the trade time trade strategy. `fminbnd` finds the optimal value for the trade time trade strategy based on the lower and upper bound values. `fminbnd` finds a local minimum for the trading cost minimization expression.

```
[tradeData.TradeTime,totalcost] = fminbnd(fun,LB,UB);
```

Display the optimized trade strategy `tradeData.TradeTime`.

```
tradeData.TradeTime

ans =

    0.19
```

Estimate Trading Costs for Optimized Strategy

Estimate the trading costs `tradeTimeCosts` using the optimized trade strategy.

```
mi = marketImpact(k,tradeData);
tr = timingRisk(k,tradeData);
pa = priceAppreciation(k,tradeData);
tradeTimeCosts = [totalcost mi pa tr];
```

Display trading costs.

```
tradeTimeCosts

tradeTimeCosts =

        100.04          56.15          4.63          39.27
```

The trading costs are:

- Total cost
- Market impact
- Price appreciation
- Timing risk

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. "Algorithmic Trading Strategies." Ph.D. Thesis. Fordham University, May 2006.

[2] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management.* Cambridge, MA: Elsevier/Academic Press, 2013.

[3] Glantz, Morton, and Robert Kissell. *Multi-Asset Risk Modeling.* Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also
fminbnd | krg | marketImpact | priceAppreciation | timingRisk

Related Examples

-   "Optimize Percentage of Volume Trading Strategy" on page 3-31
-   "Optimize Trade Schedule Trading Strategy" on page 3-39
-   "Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22
-   "Estimate Portfolio Liquidation Costs" on page 3-26

Optimize Trade Schedule Trading Strategy

This example shows how to optimize the strategy for a single stock by minimizing trading costs using transaction cost analysis from the Kissell Research Group. The optimization minimizes trading costs associated with the trade schedule trading strategy and a specified risk aversion parameter *Lambda*. The trading cost minimization is expressed as

$$\min\bigl[(MI + PA) + Lambda \cdot TR\bigr],$$

where trading costs are market impact *MI*, price appreciation *PA*, and timing risk *TR*. For details, see `marketImpact`, `priceAppreciation`, and `timingRisk`.

This example requires an Optimization Toolbox™ license. For background information, see "Optimization Theory Overview" (Optimization Toolbox).

Here, you can optimize the trade schedule trade strategy. The optimization finds a local minimum for this expression. For ways to search for the global minimum, see "Local vs. Global Optima" (Optimization Toolbox). To optimize percentage of volume and trade time strategies, see "Optimize Percentage of Volume Trading Strategy" on page 3-31 and "Optimize Trade Time Trading Strategy" on page 3-35.

To access the example code, enter `edit KRGSingleStockOptimizationExample.m` at the command line.

Retrieve Market-Impact Parameters

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Create Single Stock Data

The structure `tradeData` contains data for a single stock. Use a structure or table to define this data. The fields are:

- Number of shares
- Average daily volume
- Volatility
- Stock price
- Alpha estimate

```
tradeData.Shares = 100000;
tradeData.ADV = 1000000;
tradeData.Volatility = 0.25;
tradeData.Price = 35;
tradeData.Alpha_bp = 50;
```

Define the number of trades and the volume per trade for the initial strategy. The fields `VolumeProfile` and `TradeSchedule` define the initial trade schedule trade strategy.

```
numIntervals = 26;
tradeData.VolumeProfile = ones(1,numIntervals) * ...
    tradeData.ADV/numIntervals;
tradeData.TradeSchedule = ones(1,numIntervals) .* ...
    (tradeData.Shares./numIntervals);
```

Define Optimization Parameters

Define risk aversion level `Lambda`. Set `Lambda` from 0 to `Inf`.

```
Lambda = 1;
```

Define lower `LB` and upper `UB` bounds of shares traded per interval for optimization.

```
LB = zeros(1,numIntervals);
UB = ones(1,numIntervals) .* tradeData.Shares;
```

Specify constraints `Aeq` and `Beq` to denote that shares traded in the trade schedule must match the total number of shares.

```
Aeq = ones(1,numIntervals);
Beq = tradeData.Shares;
```

Define the maximum number of function evaluations and iterations for optimization. Set `'MaxFunEvals'` and `'MaxIter'` to large values so that the optimization can iterate many times to find a local minimum.

```
options = optimoptions('fmincon','MaxFunEvals',100000,'MaxIter',100000);
```

Define the function handle `fun` for the objective function. To access the code for this function, enter `edit krgSingleStockOptimizer.m`.

```
fun = @(tradeschedule)krgSingleStockOptimizer(tradeschedule,k, ...
    tradeData,Lambda);
```

Minimize Trading Costs for Trade Strategy

Minimize the trading costs for the trade schedule trade strategy. `fmincon` finds the optimal value for the trade schedule trade strategy based on the lower and upper bound values. It does this by finding a local minimum for the trading cost.

```
[tradeData.TradeSchedule,totalcost,exitflag] = fmincon(fun, ...
    tradeData.TradeSchedule,[],[],Aeq,Beq,LB,UB,[],options);
```

To check whether `fmincon` found a local minimum, display the reason why the function stopped.

```
exitflag
```

```
exitflag =

     1.00
```

`fmincon` returns 1 when it finds a local minimum. For details, see exitflag.

Display the optimized trade strategy `tradeData.TradeSchedule`.

```
tradeData.TradeSchedule
```

```
ans =

  Columns 1 through 5

     35563.33      18220.14      11688.59       8256.81       6057.39
```

```
...
```

Estimate Trading Costs for Optimized Strategy

Estimate trading costs `tradeScheduleCosts` using the optimized trade strategy.

```
mi = marketImpact(k,tradeData);
pa = priceAppreciation(k,tradeData);
tr = timingRisk(k,tradeData);
tradeScheduleCosts = [totalcost mi pa tr];
```

Display trading costs.

```
tradeScheduleCosts
```

```
tradeScheduleCosts =

        97.32          47.66          6.75          42.91
```

The trading costs are:

- Total cost
- Market impact
- Price appreciation
- Timing risk

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. "Algorithmic Trading Strategies." Ph.D. Thesis. Fordham University, May 2006.

[2] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[3] Glantz, Morton, and Robert Kissell. *Multi-Asset Risk Modeling*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`fmincon` | `krg` | `marketImpact` | `optimoptions` | `priceAppreciation` | `timingRisk`

Related Examples

Estimate Trading Costs for Collection of Stocks

This example shows how to estimate four different trading costs for a collection of stocks using Kissell Research Group transaction cost analysis.

Retrieve Market-Impact Parameters and Load Transaction Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Load the example data `TradeData` from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData.mat TradeData
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Estimate Trading Costs

Estimate instantaneous trading cost `itc` using `TradeData`.

```
itc = iStar(k,TradeData);
```

Estimate market-impact cost `mi`.

```
mi = marketImpact(k,TradeData);
```

Estimate timing risk `tr`.

```
tr = timingRisk(k,TradeData);
```

Estimate price appreciation `pa`.

```
pa = priceAppreciation(k,TradeData);
```

See Also
`iStar` | `krg` | `marketImpact` | `priceAppreciation` | `timingRisk`

More About

Conduct Back Test on Portfolio

This example shows how to conduct a back test on a set of stocks using transaction cost analysis from the Kissell Research Group.

- Analyze the implementation of an investment strategy on a specific day or date range.
- Estimate historical market-impact costs and the corresponding dollar values for the specified historical dates.
- Analyze the trading costs of different orders on various dates.

To access the example code, enter `edit KRGBackTestingExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Historical Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data `TradeDataBackTest` from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData TradeDataBackTest
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Prepare Data for Back Testing

Determine the number of stocks `numRecords` in the portfolio.

```
numRecords = length(TradeDataBackTest.Symbol);
```

Preallocate the output data table `o`.

```
o = table(TradeDataBackTest.Symbol,TradeDataBackTest.Side, ...
    TradeDataBackTest.Date,NaN(numRecords,1),NaN(numRecords,1), ...
    'VariableNames',{'Symbol','Side','Date','MI','MIDollar'});
```

Ensure that the number of shares is a positive value using the `abs` function.

```
TradeDataBackTest.Shares = abs(TradeDataBackTest.Shares);
```

Convert trade time trade strategy to the percentage of volume trade strategy.

```
TradeDataBackTest.TradeTime = TradeDataBackTest.TradeTime ...
    .* TradeDataBackTest.ADV;
TradeDataBackTest.POV = krg.tradetime2pov(TradeDataBackTest.TradeTime, ...
    TradeDataBackTest.Shares);
```

Conduct Back Test by Estimating Historical Market-Impact Costs

Estimate the historical market-impact costs for each stock in the portfolio on different dates using `marketImpact`. Convert market-impact cost from decimal into local dollars. Retrieve the resulting data in the output data table `o`.

```
for ii = 1:numRecords

    k.MiDate = TradeDataBackTest.Date(ii);
    k.MiCode = TradeDataBackTest.MICode(ii);

    o.MI(ii) = marketImpact(k,TradeDataBackTest(ii,:));
    MIDollars = (TradeDataBackTest.Shares(ii) * TradeDataBackTest.Price(ii)) ...
        * o.MI(ii)/10000 * TradeDataBackTest.FXRate(ii);

    o.MIDollar(ii) = MIDollars;

end
```

Display the first three rows of output data.

```
o(1:3,:)
```

```
ans =

    Symbol    Side      Date        MI     MIDollar

    _____    ____    _____    ____   _____

    'A'       1.00    '5/1/2015'    1.04     103.91
    'B'       1.00    '5/1/2015'    3.09    3864.44
    'C'       1.00    '5/1/2015'    8.54    5335.03
```

The output data contains these variables:

- Stock symbol
- Side
- Historical trade date
- Historical market-impact cost in basis points
- Historical market-impact value in local dollars

References

[1] Kissell, Robert. "Creating Dynamic Pre-Trade Models: Beyond the Black Box." *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[2] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[3] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Chung, Grace and Robert Kissell. "An Application of Transaction Costs in the Portfolio Optimization Process." *Journal of Trading*. Vol. 11, Number 2, Spring 2016, pp. 11–20.

See Also

krg | marketImpact

More About

- "Conduct Stress Test on Portfolio" on page 3-50
- "Liquidate Dollar Value from Portfolio" on page 3-56

- "Kissell Research Group Data Sets" on page 3-9

Conduct Stress Test on Portfolio

This example shows how to conduct a stress test on a set of stocks using transaction cost analysis from the Kissell Research Group.

- Estimate historical market-impact costs and the corresponding dollar values for the specified date range.
- Use trading costs to screen stocks in a portfolio and estimate the cost to liquidate or purchase a specified number of shares.
- Analyze trading costs during volatile periods of time such as a financial crisis, flash crash, or debt crisis.

To access the example code, enter `edit KRGStressTestingExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Historical Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Load the example data `TradeDataStressTest` from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData TradeDataStressTest
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Create a Kissell Research Group transaction-cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Prepare Data for Stress Testing

Specify the date range from May 1, 2015 through July 31, 2015.

```
startDate = '5/1/2015';
endDate = '7/31/2015';
```

Determine the number of stocks numStocks in the portfolio. Create a date range dateRange from the specified dates. Find the number of days numDates in the date range.

```
numStocks = length(TradeDataStressTest.Symbol);
dateRange = (datenum(startDate):datenum(endDate))';
numDates = length(dateRange);
```

Preallocate the output data table o.

```
outLength = numStocks*numDates;
symbols = TradeDataStressTest.Symbol(:,ones(1,numDates));
sides = TradeDataStressTest.Side(:,ones(1,numDates));
dates = dateRange(:,ones(1,numStocks))';

o = table(symbols(:),sides(:),dates(:),NaN(outLength,1),NaN(outLength,1), ...
    'VariableNames',{'Symbol','Side','Date','MI','MIDollar'});
```

Ensure that the number of shares is a positive value using the abs function.

```
TradeDataStressTest.Shares = abs(TradeDataStressTest.Shares);
```

Convert trade time trade strategy to the percentage of volume trade strategy.

```
TradeDataStressTest.TradeTime = TradeDataStressTest.TradeTime ...
    .* TradeDataStressTest.ADV;
TradeDataStressTest.POV = krg.tradetime2pov(TradeDataStressTest.TradeTime, ...
    TradeDataStressTest.Shares);
```

Conduct Stress Test by Estimating Historical Market-Impact Costs

Estimate the historical market-impact costs for each stock in the portfolio for the date range using marketImpact. Convert market-impact cost from decimal into local dollars. Retrieve the resulting data in the output data table o.

```
kk = 1;
```

```
for ii = dateRange(1):dateRange(end)

    for jj = 1:numStocks

        k.MiCode = TradeDataStressTest.MICode(jj);
        k.MiDate = ii;

        o.MI(kk) = marketImpact(k,TradeDataStressTest(jj,:));
        o.MIDollar(kk) = (TradeDataStressTest.Shares(jj) ...
            * TradeDataStressTest.Price(jj)) ...
            * o.MI(kk) /10000 * TradeDataStressTest.FXRate(jj);

        kk = kk + 1;

    end

end
```

Display the first three rows of output data.

```
o(1:3,:)
```

```
ans =
```

Symbol	Side	Date	MI	MIDollar
'A'	1.00	736085.00	3.84	384.31
'B'	1.00	736085.00	11.43	14292.24
'C'	1.00	736085.00	32.69	20430.65

The output data contains these variables:

- Stock symbol
- Side
- Historical trade date
- Historical market-impact cost in basis points
- Historical market-impact value in local dollars

Retrieve the daily market-impact cost `dailyCost`. Determine the number of days `numDays` in the output data. Loop through the data and sum the market-impact costs for individual stocks for each day.

```
numDays = length(o.Date)/numStocks;

idx = 1;
for i = 1:numDays

  dailyCost.Date(i) = o.Date(idx);
  dailyCost.DailyMiCost(i) = sum(o.MI(idx:idx+(numStocks-1)));
  idx = idx+numStocks;

end
```

Display the daily market-impact cost in the specified date range. This figure demonstrates how market-impact costs change over time.

```
plot(b.Date,b.DailyMiCost)
ylabel({'Daily Cost','(bps)'})
title('Daily Market-Impact Cost Stress Test')
xlabel('Date')
grid on
xData = linspace(b.Date(1),b.Date(92),11);
a = gca;
a.XAxis.TickLabels = datestr(xData,'mm/dd/yyyy');
a.XTickLabelRotation = 45;
```

References

[1] Kissell, Robert. "Creating Dynamic Pre-Trade Models: Beyond the Black Box." *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[2] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[3] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Chung, Grace and Robert Kissell. "An Application of Transaction Costs in the Portfolio Optimization Process." *Journal of Trading*. Vol. 11, Number 2, Spring 2016, pp. 11–20.

See Also

`krg` | `marketImpact`

More About

- "Conduct Back Test on Portfolio" on page 3-46
- "Liquidate Dollar Value from Portfolio" on page 3-56
- "Kissell Research Group Data Sets" on page 3-9

Liquidate Dollar Value from Portfolio

This example shows how to liquidate a dollar value from a portfolio while minimizing market-impact costs using transaction cost analysis from the Kissell Research Group. This example always results in a portfolio that shrinks in size. The market-impact cost minimization is expressed as

$$\arg\min_{x}\left[MI'|x|\right],$$

where *MI* is the market-impact cost for the traded shares and *x* denotes the final weights for each stock.

This example requires an Optimization Toolbox license. For background information, see "Optimization Theory Overview" (Optimization Toolbox).

The optimization finds a local minimum for the market-impact cost of liquidating a dollar value from a portfolio. For ways to search for the global minimum, see "Local vs. Global Optima" (Optimization Toolbox).

To access the example code, enter `edit KRGLiquidityOptimizationExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data `TradeDataPortOpt` and the covariance data `CovarianceData` from the file `KRGExampleData.mat`, which is included with the Trading Toolbox. Limit the data set to the first 10 rows.

```
load KRGExampleData.mat TradeDataPortOpt CovarianceData

n = 10;
TradeDataPortOpt = TradeDataPortOpt(1:n,:);
CovarianceData = CovarianceData(1:n,1:n);
C = table2array(CovarianceData);
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Define Optimization Parameters

Set the portfolio liquidation value to \$100,000,000. Set the portfolio risk boundaries between 90% and 110%. Set the maximum total market-impact cost to 50 basis points. Determine the number of stocks in the portfolio. Retrieve the upper bound constraint for the maximum market-impact cost for liquidating shares in each stock.

```
PortLiquidationValue = 100000000;
PortRiskBounds = [0.9 1.10];
maxTotalMI = 0.005;
numPortStocks = length(TradeDataPortOpt.Symbol);
maxMI = TradeDataPortOpt.UB_MaxMI;
```

Determine the target portfolio value `PortfolioTargetValue` by subtracting the portfolio liquidation value from the total portfolio value.

```
PortfolioValue = sum(TradeDataPortOpt.Value);
absPortValue = abs(TradeDataPortOpt.Value);
PortfolioAbsValue = sum(absPortValue);
PortfolioTargetValue = PortfolioValue-PortLiquidationValue;
```

Determine the current portfolio weight w based on the value of each stock in the portfolio.

```
w = sign(TradeDataPortOpt.Shares).*absPortValue/PortfolioAbsValue;
```

Specify constraints `Aeq` and `beq` to indicate that the weights must sum to one. Initialize the linear inequality constraints `A` and `b`.

```
Aeq = ones(1,numPortStocks);
```

```
beq = 1;

A = [];
b = [];
```

Retrieve the lower and upper bounds for the final portfolio weight in
`TradeDataPortOpt`.

```
LB = TradeDataPortOpt.LB_Wt;
UB = TradeDataPortOpt.UB_Wt;
```

Determine the lower and upper bounds for the number of shares in the final portfolio
using other optional constraints in the example data set.

```
lbShares = max([TradeDataPortOpt.LB_MinShares, ...
    TradeDataPortOpt.LB_MinValue./TradeDataPortOpt.Price, ...
    TradeDataPortOpt.LB_MinPctADV.*TradeDataPortOpt.ADV],[],2);

ubShares = min([TradeDataPortOpt.UB_MaxShares, ...
    TradeDataPortOpt.UB_MaxValue./TradeDataPortOpt.Price, ...
    TradeDataPortOpt.UB_MaxPctADV.*TradeDataPortOpt.ADV],[],2);
```

Specify the initial portfolio weights.

```
x0 = TradeDataPortOpt.Value./sum(TradeDataPortOpt.Value);
x = x0;
```

Define optimization options. Set the optimization algorithm to sequential quadratic
programming. Set the termination tolerance on the function value and on `x`. Set the
tolerance on the constraint violation. Set the termination tolerance on the PCG iteration.
Set the maximum number of function evaluations `'MaxFunEvals'` and iterations
`'MaxIter'`. The options `'MaxFunEvals'` and `'MaxIter'` are set to large values so
that the optimization can iterate many times to find a local minimum. Set the minimum
change in variables for finite differencing.

```
options = optimoptions('fmincon','Algorithm','sqp', ...
    'TolFun',10E-8,'TolX',10E-16,'TolCon',10E-8,'TolPCG',10E-8, ...
    'MaxFunEvals',50000,'MaxIter',50000,'DiffMinChange',10E-8);
```

Minimize Market-Impact Costs for Portfolio Liquidation

Define the function handle `objectivefun` for the sample objective function
`krgLiquidityFunction`. To access the code for this function, enter `edit`
`krgLiquidityFunction.m`. Define the function handle `constraintsfun` for the

sample function `krgLiquidityConstraint` that sets additional constraints. To access the code for this function, enter `edit krgLiquidityConstraint.m`.

```
objectivefun = @(x) krgLiquidityFunction(x,TradeDataPortOpt, ...
    PortfolioTargetValue,k);
```

```
constraintsfun = @(x) krgLiquidityConstraint(x,w,C,TradeDataPortOpt, ...
    PortfolioTargetValue,PortRiskBounds,lbShares,ubShares,maxMI,maxTotalMI,k);
```

Minimize the market-impact costs for the portfolio liquidation. `fmincon` finds the optimal value for the portfolio weight for each stock based on the lower and upper bound values. It does this by finding a local minimum for the market-impact cost.

```
[x,~,exitflag] = fmincon(objectivefun,x0,A,b,Aeq,beq,LB,UB, ...
    constraintsfun,options);
```

To check whether `fmincon` found a local minimum, display the reason why the function stopped.

```
exitflag
```

```
exitflag =

    1.00
```

`fmincon` returns 1 when it finds a local minimum. For details, see exitflag.

Determine the optimized weight value `x1` of each stock in the portfolio in decimal format.

```
x1 = x.*PortfolioTargetValue/PortfolioValue;
```

Determine the optimized portfolio target value `TargetValue` and number of shares `SharesToTrade` for each stock in the portfolio.

```
TargetShares = x*PortfolioTargetValue./TradeDataPortOpt.Price;
SharesToTrade = TradeDataPortOpt.Shares-TargetShares;
TargetValue = x*PortfolioTargetValue;
TradeDataPortOpt.Shares = abs(SharesToTrade);
```

Determine the optimized percentage of volume strategy.

```
TradeDataPortOpt.TradeTime = TradeDataPortOpt.TradeTime ...
    .* TradeDataPortOpt.ADV;
TradeDataPortOpt.POV = krg.tradetime2pov(TradeDataPortOpt.TradeTime, ...
```

```
    TradeDataPortOpt.Shares);
```

Estimate the market-impact costs `MI` for the number of shares to liquidate.

```
MI = marketImpact(k,TradeDataPortOpt)/10000;
```

To view the market-impact cost in decimal format, specify the display format. Display the market-impact cost for the first three stocks in the portfolio.

```
format

MI(1:3)

ans =

    1.0e-03 *

    0.1477
    0.1405
    0.1405
```

To view the target number of shares with two decimal places, specify the display format. Display the target number of shares for the first three stocks in the portfolio.

```
format bank

TargetShares(1:3)

ans =

     -23640.11
   -154656.73
    -61193.04
```

The negative values denote selling shares from the portfolio.

Display the traded value for the first three stocks in the portfolio.

```
TargetValue(1:3)

ans =

    -968062.45
   -1521760.41
   -2448131.64
```

To simulate trading the target number of shares on a historical date range, you can now conduct a stress test on the optimized portfolio. For details about conducting a stress test, see "Conduct Stress Test on Portfolio" on page 3-50.

References

[1] Kissell, Robert. "Creating Dynamic Pre-Trade Models: Beyond the Black Box." *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[2] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[3] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Chung, Grace and Robert Kissell. "An Application of Transaction Costs in the Portfolio Optimization Process." *Journal of Trading*. Vol. 11, Number 2, Spring 2016, pp. 11–20.

See Also

fmincon | krg | marketImpact | optimoptions

More About

- "Conduct Stress Test on Portfolio" on page 3-50
- "Optimize Trade Schedule Trading Strategy" on page 3-39
- "Optimize Long Portfolio" on page 3-62
- "Kissell Research Group Data Sets" on page 3-9

Optimize Long Portfolio

This example shows how to determine the optimal portfolio weights for a specified dollar value using transaction cost analysis from the Kissell Research Group. The sample portfolio contains only long shares of stock. You can incorporate risk, return, and market-impact cost during implementation of the investment decision.

This example requires an Optimization Toolbox license. For background information, see "Optimization Theory Overview" (Optimization Toolbox).

The `KRGPortfolioOptimizationExample` function, which you can access by entering `edit KRGPortfolioOptimizationExample.m`, addresses three different optimization scenarios:

1 Maximize the trade off between net portfolio return and portfolio risk. The trade off maximization is expressed as

$$\arg\max_{x}\left[R'x - MI'|x| - \lambda x'Cx\right],$$

where:

- R is the estimated return for each stock in the portfolio.
- x denotes the weights for each stock in the portfolio.
- MI is the market-impact cost for the specified dollar value and share quantities.
- λ is the specified risk aversion parameter.
- C is the covariance matrix of the stock data.

2 Minimize the portfolio risk subject to a minimum return target using

$$\arg\min_{x}\left[x'Cx\right].$$

3 Maximize net portfolio return subject to a maximum risk exposure target using

$$\arg\max_{x}\left[R'x - MI'|x|\right].$$

Lower and upper bounds constrain x in each scenario. Each optimization finds a local optimum. For ways to search for the global optimum, see "Local vs. Global Optima" (Optimization Toolbox).

Retrieve Market-Impact Parameters and Load Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data `TradeDataPortOpt` and the covariance data `CovarianceData` from the file `KRGExampleData.mat`, which is included with the Trading Toolbox. Limit the data set to the first 50 rows.

```
load KRGExampleData TradeDataPortOpt CovarianceData

n = 50;
TradeDataPortOpt = TradeDataPortOpt(1:n,:);
CovarianceData = CovarianceData(1:n,1:n);
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Maximize Net Portfolio Return

Run the optimization scenario using the example and covariance data. To run the first optimization, specify `1` in the last input argument.

```
[Weight,Shares,Value,MI] = KRGPortfolioOptimizationExample(TradeDataPortOpt, ...
```

```
    CovarianceData,1);
```

`KRGPortfolioOptimizationExample` returns the optimized values for each stock in the portfolio:

- Portfolio weight
- Number of shares
- Portfolio dollar value
- Market-impact cost

To run the other two scenarios, specify `2` or `3` in the last input argument of `KRGPortfolioOptimizationExample`.

Display the portfolio weight for the first three stocks in the portfolio in decimal format.

```
format

Weight(1:3)

ans =

    0.0100
    0.3198
    0.1610
```

Display the number of shares using two decimal places for the first three stocks in the portfolio.

```
format bank

Shares(1:3)

ans =

      24420.02
    3249893.71
     402364.47
```

Display the portfolio dollar value for the first three stocks in the portfolio.

```
Value(1:3)

ans =
```

```
   1000000.00
  31977654.17
  16097274.50
```

Display the market-impact cost for the first three stocks in the portfolio in decimal format.

```
format

MI(1:3)

ans =

   1.0e-03 *

   0.1250
   0.7879
   0.3729
```

References

[1] Kissell, Robert. "Creating Dynamic Pre-Trade Models: Beyond the Black Box." *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[2] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[3] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Chung, Grace and Robert Kissell. "An Application of Transaction Costs in the Portfolio Optimization Process." *Journal of Trading*. Vol. 11, Number 2, Spring 2016, pp. 11–20.

See Also
fmincon | krg | marketImpact | optimoptions

More About

-

Determine Buy-Sell Imbalance Using Cost Index

This example shows how to determine buy-sell imbalance using transaction cost analysis from the Kissell Research Group. Imbalance is the difference between buy-initiated and sell-initiated volume given actual market conditions on the day and over the specified trading period. A positive imbalance indicates buying pressure in the stock and a negative imbalance indicates selling pressure. The cost index helps investors to understand how the trading cost environment affects the order flow in the market. The index can be a performance-based index, such as the S&P 500, that shows market movement and value or a volatility index that shows market uncertainty.

The imbalance share quantity is the value of x such that

$$0 = |MICost| \cdot 10000 - MI(x),$$

where

$$MI(x) = \left[b_1 \cdot \left(\frac{x}{Volume} \right)^{a_4} + (1 - b_1) \right] \cdot \left[a_1 \cdot \left(\frac{x}{ADV} \right)^{a_2} \cdot \sigma^{a_3} \cdot Price^{a_5} \right].$$

MI is the market-impact cost for a stock transaction. The estimated trading costs represent the incremental price movement of the stock in relation to the underlying index price movement. $Volume$ is the actual daily volume of a stock in the basket. ADV is the average daily volume of a stock in the basket. $Price$ is the price of a stock in the basket. The other variables in the equations are:

- σ — Price volatility.
- a_1 — Price sensitivity to order flow.
- a_2 — Order size shape.
- a_3 — Volatility shape.
- a_4 — Percentage of volume rate shape.
- a_5 — Price shape.
- $1 - b_1$ — Percentage of permanent market impact. Permanent impact is the unavoidable impact cost that occurs because of the information content of the trade.

- b_1 — Percentage of temporary market impact. Temporary impact is dependent upon the trading strategy. Temporary impact occurs because of the liquidity demands of the investor.

- $MICost = TotalCost - Beta \cdot IndexCost$, where:

  - *TotalCost* — Change in the volume-weighted average price compared to the open price for the stocks.
  - *Beta* — Beta.
  - *IndexCost* — Change in the volume-weighted average price compared to the open price for the index. Index cost adjusts the price for market movement using an underlying index and beta.

In this example, you can run this code using current or historical data. Current data includes prices starting from the open time through the current time. Historical data uses the prices over the entire day. Historical costs use market-impact parameters for the specified region and date. Therefore, historical costs change from record to record.

For a current cost index, you load the example table `TradeDataCurrent` from the file `KRGExampleData.mat`. For a historical cost index, you load the example table `TradeDataHistorical` from the file `KRGExampleData.mat`. This example calculates a current cost index.

To access the example code, enter `edit KRGCostIndexExample.m` at the command line.

After running this code, you can submit an order for execution using Bloomberg, for example.

This example requires an Optimization Toolbox license. For background information, see "Optimization Theory Overview" (Optimization Toolbox).

Retrieve Market-Impact Parameters and Load Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
```

```
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object k. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data TradeDataCurrent, which is included with the Trading Toolbox. Calculate the number of stocks in the portfolio.

```
load KRGExampleData.mat TradeDataCurrent
TradeData = TradeDataCurrent;

numStocks = height(TradeData);
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Define Optimization Parameters

Define the maximum number of function iterations for optimization. Set 'MaxIterations' to a large value so that the optimization can iterate many times to solve a system of nonlinear equations.

```
options = optimoptions('fsolve','MaxIterations',4000);
```

Estimate Trading Costs Using Cost Index

Determine the total cost and beta cost. Calculate the side of the initial market-impact cost estimate. Determine the initial volume x0.

```
totalCost = TradeData.VWAP ./ TradeData.Open - 1;
indexCost = TradeData.Beta .* ...
    (TradeData.IndexVWAP ./ TradeData.IndexOpen - 1);
miCost = totalCost - indexCost;
sideIndicator = sign(miCost);
x0 = 0.5 * TradeData.Volume;
```

Create a table that stores all output data. First, add these variables:

- Symbol — Stock symbol
- Date — Transaction date

- `Side` — Side
- `TotalVolume` — Transaction volume
- `TotalCost` — Total transaction cost
- `IndexCost` — Index cost

```
costIndexTable = table;
costIndexTable.Symbol = TradeData.Symbol;
costIndexTable.Date = TradeData.Date;
costIndexTable.Side = sideIndicator;
costIndexTable.TotalVolume = TradeData.Volume;
costIndexTable.TotalCost = totalCost;
costIndexTable.IndexCost = indexCost;
```

Use a `for`-loop to calculate the cost index for each stock in the portfolio. Each stock might have different market-impact codes and dates. Use the `costIndexExampleEq` function that contains the nonlinear equation to solve. To access the code for the `costIndexExampleEq` function, enter `edit KRGCostIndexExample.m`.

Add these variables to the output table:

- `Imbalance` — Imbalance
- `ImbalancePctADV` — Imbalance as percentage of average daily volume
- `ImbalancePctDayVolume` — Imbalance as percentage of the daily volume
- `BuyVolume` — Buy volume
- `SellVolume` — Sell volume
- `MI` — Market-impact cost
- `ExcessCost` — Excess cost

```
for i = 1:numStocks

  % Set the MiCode and MiDate of the object for each stock
  k.MiCode = TradeData.MICode(i);
  k.MiDate = TradeData.Date(i);

  % Solve for Shares for each stock that results in the target market
  % impact cost.

  % In this example, x is the number of shares (imbalance) that causes
  % the MI impact cost, the number of shares that result in a market
  % impact cost of MI. Here use abs(MI) since market-impact
```

```
    % cost is always positive. If the market-impact cost is 0.0050 then
    % fsolve tries to find the number of shares x so that the market
    % impact formula returns 0.0050.
    % Note that fsolve is using the cost in basis points.
    x = fsolve(@(x) costIndexExampleEq(x,miCost(i),TradeData(i,:),k), ...
        x0(i),options);

    % The imbalance must be between 0 and the actual traded volume.
    x = max(min(x,TradeData.Volume(i)),0);

    % Recalculate the percentage of volume and shares based on x.
    TradeData.POV(i) = x/TradeData.Volume(i);
    TradeData.Shares(i) = x;

    % Calculate the new cost as a decimal value.
    mi = marketImpact(k,TradeData(i,:))/10000;

    % imbalance is the share amount specified as buy or sell by the
    % sideIndicator.
    imbalance = sideIndicator(i) * x;

    % Calculate the buy and sell volumes.
    % Knowing that:
    %
    % Volume = buyVolume + sellVolume;
    % Imbalance = buyVolume - sellVolume;
    %
    % Solve for buyVolume and sellVolume
    buyVolume = (TradeData.Volume(i) + imbalance) / 2;
    sellVolume = (TradeData.Volume(i) - imbalance) / 2;

    % Fill output table
    costIndexTable.Imbalance(i,1) = imbalance;
    costIndexTable.ImbalancePctADV(i,1) = imbalance/TradeData.ADV(i);
    costIndexTable.ImbalancePctDayVolume(i,1) = imbalance/TradeData.Volume(i);
    costIndexTable.BuyVolume(i,1) = buyVolume;
    costIndexTable.SellVolume(i,1) = sellVolume;
    costIndexTable.MI(i,1) = mi * sideIndicator(i);
    costIndexTable.ExcessCost(i,1) = totalCost(i) - mi - indexCost(i);

end
```

Display the imbalance amount for the first stock in the output data.

```
costIndexTable.Imbalance(1)
```

```
ans =

  -8.7894e+04
```

The negative imbalance amount indicates selling pressure. Decide whether to buy, hold, or sell shares of this stock in the portfolio.

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management.* Cambridge, MA: Elsevier/Academic Press, 2013.

[2] Malamut, Roberto. "Multi-Period Optimization Techniques for Trade Scheduling." Presentation at the QWAFAFEW New York Conference, April 2002.

[3] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies.* New York, NY: AMACOM, Inc., 2003.

See Also

fsolve | krg | marketImpact | optimoptions

More About

- "Conduct Back Test on Portfolio" on page 3-46
- "Conduct Stress Test on Portfolio" on page 3-50
- "Kissell Research Group Data Sets" on page 3-9

Rank Broker Performance

This example shows how to determine the best performing brokers across transactions using transaction cost analysis from the Kissell Research Group. You rank brokers based on broker value add and arrival cost, and then determine which brokers perform best in which market conditions and trading characteristics. A positive value add indicates that the broker exceeds performance expectations given actual market conditions and trade characteristics, which results in fund savings. A negative value add indicates that the broker did not meet performance expectations, which result in an incremental cost to the fund.

In this example, you find which brokers over- or under-perform by comparing arrival costs and estimated trading costs. A broker with an arrival cost that is less than the estimated trading cost over-performs, which causes the fund to save money. A broker with an arrival cost that is greater than the estimated trading cost under-performs, which causes the fund to incur an incremental cost.

This example also shows how to estimate costs by broker, which requires custom market-impact parameters for each broker.

You can use similar steps as in this example to rank trading venues and algorithms.

To access the example code, enter `edit KRGTradePerformanceRankingExample.m` at the command line.

After executing the code in this example, you can submit an order for execution using Bloomberg, for example.

Retrieve Market-Impact Parameters and Load Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data with broker codes in the `MI_Broker.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Broker.csv');
close(f)
```

```
miData = readtable('MI_Broker.csv','delimiter',',', ...
    'ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object k. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data TradeData, Basket, and BrokerNames, which is included with the Trading Toolbox.

```
load KRGExampleData.mat TradeData Basket BrokerNames
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Calculate Costs for Each Broker

Select the trade categories. Calculate the average arrival cost, market-impact cost, and broker value add for each broker.

```
TradeData.TradeSize = TradeData.Shares ./ TradeData.ADV;
TradeData.ArrivalCost = TradeData.SideIndicator .* ...
    (TradeData.AvgExecPrice ./ TradeData.ArrivalPrice-1) * 10000;
TradeData.MI = marketImpact(k,TradeData);
TradeData.ValueAdd = TradeData.MI - TradeData.ArrivalCost;
```

Retrieve broker names and the number of brokers. Preallocate output data variables.

```
uniqueBrokers = unique(TradeData.Broker);
numBrokers = length(uniqueBrokers);
avgCost = NaN(numBrokers,1);
avgMI = NaN(numBrokers,1);
avgValueAdd = NaN(numBrokers,1);
```

Rank Brokers by Average Broker Value Add

Calculate broker ranking using a transaction size between 5% and 10% of average daily volume (ADV). Calculate average arrival cost, average market-impact cost, and average broker value add.

```
indBroker = (TradeData.TradeSize >= 0.05) & (TradeData.TradeSize <= 0.10);
```

```
if any(indBroker)
    TD = TradeData(indBroker,:);
    for i = 1:numBrokers
        j = strcmp(TD.Broker,uniqueBrokers(i));
        if any(j)
            avgCost(i) = mean(TD.ArrivalCost(j));
            avgMI(i) = mean(TD.MI(j));
            avgValueAdd(i) = mean(TD.ValueAdd(j));
        end
    end
end

% Get valid average cost values (non NaN's)
indAvgCost = ~isnan(avgCost);
```

Create a table to store the broker ranking. Sort the ranking by average cost.

```
BrokerRankings = table(uniqueBrokers(indAvgCost),(1:sum(indAvgCost))', ...
    avgCost(indAvgCost),avgMI(indAvgCost),avgValueAdd(indAvgCost), ...
    'VariableNames',{'Broker','Rank','AvgArrivalCost','AvgMI','AvgValueAdd'});
BrokerRankings = sortrows(BrokerRankings,-5);
BrokerRankings.Rank = (1:sum(indAvgCost))';  % Reset rank
```

Compare the average broker value add in basis points using a bar graph.

```
bar(BrokerRankings.AvgValueAdd)
set(gca,'XTickLabel',BrokerRankings.Broker)
ylabel('Average Value Add (bp)')
title('Average Value Add by Broker')
grid
```
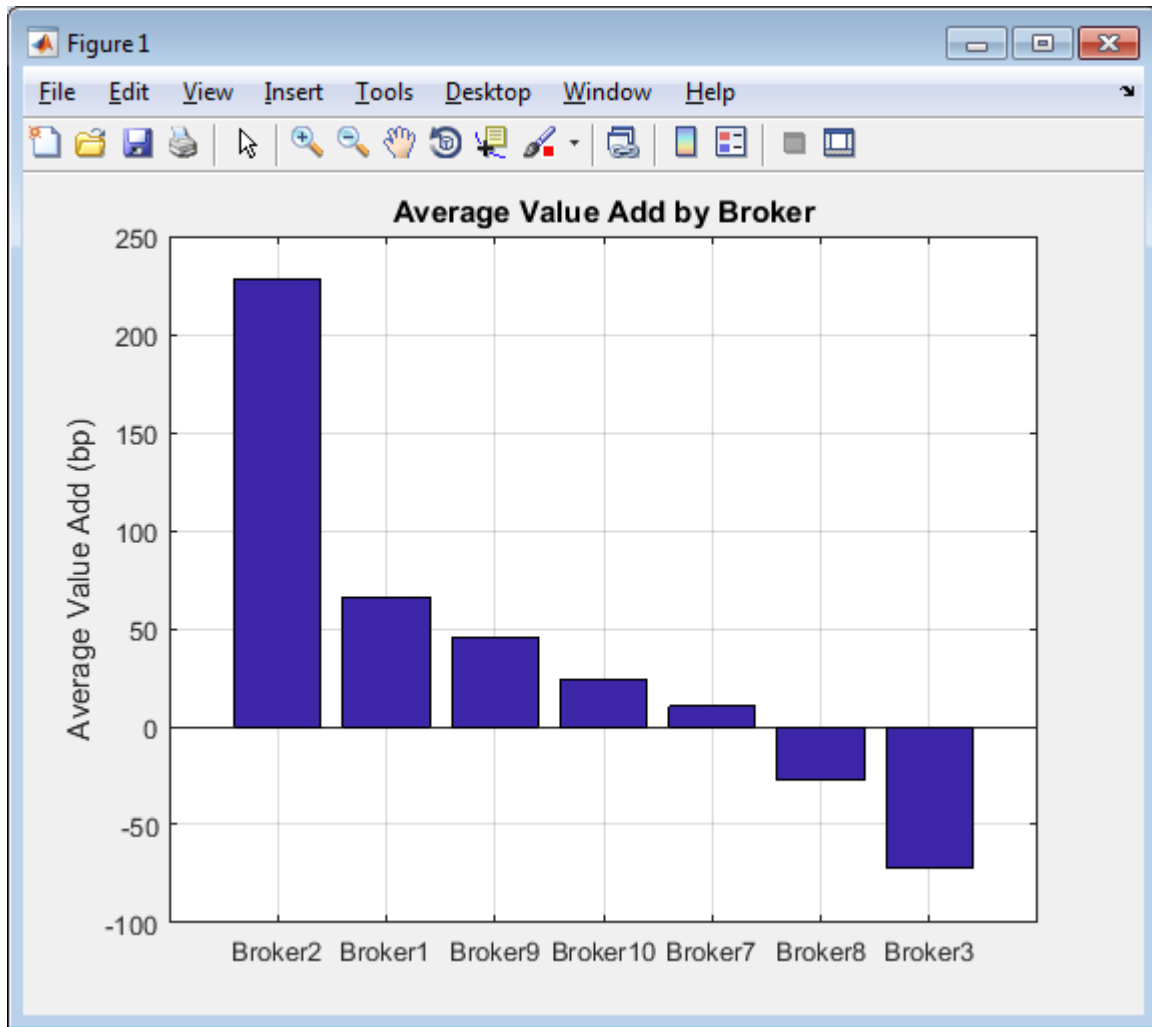
The broker `Broker2` over-performs while `Broker3` under-performs across transactions. Decide to use `Broker2` for future transactions.

Estimate Trading Costs for Trade List

Estimate the trading costs for each broker using a specified order or trade list.

```
% Get the number of orders from the trade list table
```

```
numOrders = size(Basket.Symbols,1);

% Calculate pre-trade cost for each broker for each order
BrokerPreTrade = zeros(numOrders,numBrokers);
for i = 1:numBrokers
    % Market-impact code for broker corresponds to the MICode in the market
    % impact data, for example, Broker1 = 1.
    k.MiCode = i;

    % Calculate market-impact cost for each broker
    BrokerPreTrade(:,i) = marketImpact(k,Basket);
end

% Convert output to a table with the symbols used as the row names.
BrokerPreTrade = array2table(BrokerPreTrade,'VariableNames', ...
    BrokerNames.Broker,'RowNames',Basket.Symbols);
```

Compare Market-Impact Costs by Broker

For one stock ABC, compare market-impact cost in basis points for each broker using a bar graph.

```
% Plot best broker for given stock
bar(table2array(BrokerPreTrade(1,:)))
set(gca,'XTickLabel',BrokerNames.Broker)
ylabel('Market-Impact Cost (bp)')
title(['Market-Impact Cost by Broker for Symbol ' ...
    BrokerPreTrade.Properties.RowNames{1}])
grid
```
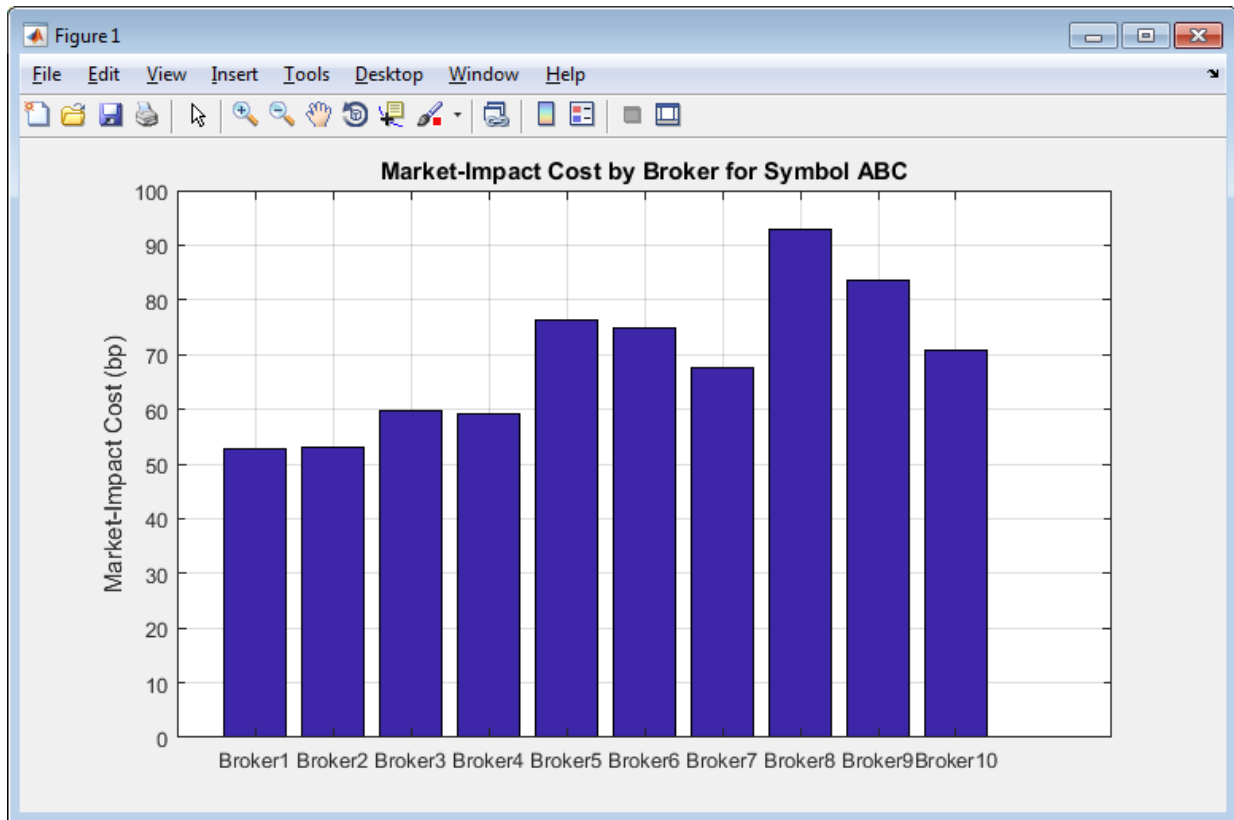
The broker `Broker8` has the highest market-impact cost and `Broker1` has the lowest one. Decide to use `Broker1` for executing the transaction using stock `ABC`.

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[2] Malamut, Roberto. "Multi-Period Optimization Techniques for Trade Scheduling." Presentation at the QWAFAFEW New York Conference, April 2002.

[3] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

krg | marketImpact

More About

Optimize Trade Schedule Trading Strategy for Basket

This example shows how to optimize the strategy for a basket by minimizing trading costs using transaction cost analysis from the Kissell Research Group. Using this optimization, you determine the optimal order slicing strategy for the basket based on the trade-off between trading cost, risk, and the specified level of risk aversion. The optimization minimizes trading costs associated with the trade schedule trading strategy and a specified risk aversion parameter *Lambda*. The trading cost minimization is expressed as

$$\min\left[(MI + PA) + Lambda \cdot TR\right],$$

where trading costs are market impact *MI*, price appreciation *PA*, and timing risk *TR*.

To access the example code, enter `edit KRGTradeOptimizationExample.m` at the command line. In this example, you can run this code using a trade schedule trading strategy or a percentage of volume trading strategy. This example shows the trade schedule trading strategy. An exponential function determines the optimal trade schedule.

After executing the code in this example, you can submit an order for execution using Bloomberg, for example.

This example requires an Optimization Toolbox license. For background information, see "Optimization Theory Overview" (Optimization Toolbox).

Retrieve Market-Impact Parameters and Load Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
```

```
',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object k. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data TradeDataTradeOpt and the covariance data CovarianceTradeOpt from the file KRGExampleData.mat, which is included with the Trading Toolbox.

```
load KRGExampleData TradeDataTradeOpt CovarianceTradeOpt
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Define Optimization Parameters

Specify initial values for risk, trading periods, portfolio value, and covariance matrix. Convert to a buy-only problem. Set the initial trade schedule.

```
% Convert table to array
CovarianceTradeOpt = table2array(CovarianceTradeOpt);

% Use total trading time of 1 day with 13 trading periods
totalDays = 1;
periodsPerDay = 13;

% Set risk aversion level
Lambda = 0.5;

% Set minimum and maximum percentage of volume
minPOV = 0.00;
maxPOV = 0.60;

% total number of trading periods
totalNumberPeriods = totalDays * periodsPerDay;

% Portfolio Value
PortfolioValue = TradeDataTradeOpt.Price'*TradeDataTradeOpt.Shares;

% Number of stocks
numberStocks = height(TradeDataTradeOpt);
```

```
% Covariance matrix is annualized covariance matrix in decimals.
% Convert to ($/Shares)^2 units for the trade period; this matrix is for a
% two-sided portfolio, buys and sells or long and short.
CC = diag(TradeDataTradeOpt.Price) * CovarianceTradeOpt * ...
    diag(TradeDataTradeOpt.Price);

% Scale to one trading period
CC = CC / periodsPerDay / k.TradeDaysInYear;

% Convert to buy-only problem (e.g., one-sided problem)
CC = TradeDataTradeOpt.SideIndicator * TradeDataTradeOpt.SideIndicator' .* CC;

% Convert Alpha_bp from basis points per day to cents/share per period
TradeDataTradeOpt.Alpha_bp = TradeDataTradeOpt.Alpha_bp / 1000 .* ...
    TradeDataTradeOpt.Price / totalNumberPeriods;

% Set the initial trade schedule or POV values
theta0 = rand(numberStocks,1);
```

Define optimization options using the optimset function. For details about these options, see "Optimization Options Reference" (Optimization Toolbox).

```
optionsold = optimset;
options = optimset(optionsold,'LargeScale','on','GradObj','off', ...
    'DerivativeCheck','off','FinDiffType','central','FinDiffRelStep',1E-12, ...
    'TolFun',10E-5,'TolX',10E-12,'TolCon',10E-12,'TolPCG',10E-12, ...
    'MaxFunEvals',20000,'MaxIter',20000,'DiffMinChange',10E-04);
```

Define lower and upper bounds of shares traded per interval for optimization.

```
LB = zeros(numberStocks,1);
UB = 100 * ones(numberStocks,1);
```

Minimize Trading Costs for Trade Strategy

Minimize the trading costs for the trade schedule strategy. fmincon finds the optimal value for the trade schedule trade strategy based on the lower and upper bound values. It does this by finding a local minimum for the trading cost. Use the objective function optimizeTradingSchedule. To access the code for this function, enter edit KRGTradeOptimizationExample.m.

```
[theta,fval,exitflag,output] = fmincon(@optimizeTradingSchedule,theta0,[], ...
    [],[],[],LB,UB,[],options,totalNumberPeriods,numberStocks,periodsPerDay, ...
    TradeDataTradeOpt,CC,Lambda,k);
```

To check whether `fmincon` found a local minimum, display the reason why the function stopped.

```
exitflag

exitflag =

    1.00
```

`fmincon` returns 1 when it finds a local minimum. For details, see exitflag.

Calculate shares to trade, residual shares, price appreciation, and timing risk. Then, calculate the average percentage of volume rate and trade time.

```
numPeriods = 1:totalNumberPeriods;
K_Matrix = repmat(numPeriods,numberStocks,1);
Theta_Matrix = repmat(theta,1,totalNumberPeriods);
Volume_Matrix = repmat(TradeDataTradeOpt.ADV/periodsPerDay,1, ...
    totalNumberPeriods);
TradeDataTradeOpt.VolumeProfile = Volume_Matrix;
Shares_Matrix = repmat(TradeDataTradeOpt.Shares,1,totalNumberPeriods);

% X = Shares to trade in period i
Xpct = (exp(-K_Matrix .* Theta_Matrix) .* (exp(Theta_Matrix)-1)) ./ ...
    (1 - exp(-totalNumberPeriods * Theta_Matrix));
X = repmat(TradeDataTradeOpt.Shares,1,totalNumberPeriods) .* Xpct;
TradeDataTradeOpt.TradeSchedule = X;

% R = Residual Shares at beginning of period i
Rpct = (exp(-(K_Matrix-1).*Theta_Matrix) - exp(-totalNumberPeriods.*Theta_Matrix)) ./
    (1-exp(-totalNumberPeriods.*Theta_Matrix));
R = repmat(TradeDataTradeOpt.Shares,1,totalNumberPeriods) .* Rpct;

% Price Appreciations in Dollars
PA = sum(R,2) .* TradeDataTradeOpt.Alpha_bp;

% Market Impact in Dollars
MI = marketImpact(k,TradeDataTradeOpt) .* TradeDataTradeOpt.Shares .* ...
    TradeDataTradeOpt.Price ./10000;

% Timing Risk in Dollars
TR = sqrt(sum(R.^2,2) .* diag(CC));
TR_bp = TR ./ (TradeDataTradeOpt.Shares .* TradeDataTradeOpt.Price) * 10000;

% Avg POV Rate
```

```
kTR = ((TR_bp/10000*1./TradeDataTradeOpt.Volatility).^2).*(k.TradeDaysInYear*3 ./ ...
    (TradeDataTradeOpt.Shares./TradeDataTradeOpt.ADV));
POV = 1./(1+kTR);
POV = max(POV,TradeDataTradeOpt.Shares./(TradeDataTradeOpt.Shares+totalDays .* ...
    TradeDataTradeOpt.ADV));

% TradeTime
TradeDataTradeOpt.TradeTime = TradeDataTradeOpt.Shares./TradeDataTradeOpt.ADV .* ...
    (1-POV)./POV;
```

Estimate total trading costs using the optimized trade strategy.

```
TotMI = sum(MI) / (TradeDataTradeOpt.Shares' * TradeDataTradeOpt.Price) ...
    .* 10000;        % bp
TotPA = sum(PA) / (TradeDataTradeOpt.Shares' * TradeDataTradeOpt.Price) ...
    .* 10000;        % bp
TotTR = sqrt(trace(R'*CC*R)) ./ (TradeDataTradeOpt.Shares' * ...
    TradeDataTradeOpt.Price) * 10000;
```

Display total market-impact cost, price appreciation, and timing risk.

```
totalcosts = [TotMI TotPA TotTR]

totalcosts =

   38.2902         0   26.5900
```

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[2] Malamut, Roberto. "Multi-Period Optimization Techniques for Trade Scheduling." Presentation at the QWAFAFEW New York Conference, April 2002.

[3] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

fmincon | krg | marketImpact | optimset | priceAppreciation | timingRisk

More About

Create Basket Summary and Efficient Trading Frontier

This example shows how to evaluate trading cost and risk components for a basket using transaction cost analysis from the Kissell Research Group. To create a basket summary, estimate trading costs for the entire basket using basket optimization techniques, and then calculate risk statistics for the basket. Using the basket summary, you can provide brokers and third parties with enough information to assess the overall execution costs and trading difficulty of the basket. The basket summary enables providing transaction information without revealing the actual orders. Another way brokers use a basket summary is to assess a fair value principal bid estimate. A principal bid is a transaction where the broker charges a bid premium that is higher than the associated commission. Brokers present this transaction with guaranteed completion for a given price.

In this example, you can see a basket summary analysis table and a principal bid summary. The basket summary provides trading cost estimates for the basket across different categories, such as side, market capitalization, and market sector. The principal bid summary contains the efficient trading frontier that provides the different estimated trading costs for different time periods. The efficient trading frontier shows how cost and risk change by trading more aggressively or passively. With passive trading, market impact decreases as timing risk increases. With aggressive trading, market impact increases as timing risk decreases.

The code in this example depends on the output data from the example "Optimize Trade Schedule Trading Strategy for Basket" on page 3-80. Run the code in that example first and then run the code in this example.

To access the example code, enter `edit KRGBasketAnalysisExample.m` at the command line.

After executing the code in this example, you can submit an order for execution using Bloomberg, for example.

Estimate Trading Costs in Basket

Determine the covariance matrix. Covariance indicates how the prices of stocks in the basket relate to each other.

```
% Covariance matrix is annualized covariance matrix in decimals.
% Convert to ($/Shares)^2 units for the trade period, this matrix is for a
% two-sided portfolio, buys and sells or long and short.
diagPrice = diag(TradeDataTradeOpt.Price);
```

```
C1 = TradeDataTradeOpt.SideIndicator * TradeDataTradeOpt.SideIndicator' .* ...
    diagPrice * CovarianceTradeOpt * diagPrice;

% Covariance Matrix in $/Share^2 by Day
CD = diagPrice * CovarianceTradeOpt * diagPrice;     % compute Covariance Matrix in ($/s
CD = CD / k.TradeDaysInYear;                          % scale to 1-day
CD = TradeDataTradeOpt.SideIndicator * TradeDataTradeOpt.SideIndicator' ...
    .* CD;
```

Add the estimated trading costs from the trade schedule optimization to the basket data.

```
% Market impact in basis points
TradeDataTradeOpt.MI = MI ./ (TradeDataTradeOpt.Shares .* ...
    TradeDataTradeOpt.Price) .* 10000;

% Timing risk in basis points
TradeDataTradeOpt.TR = TR ./ (TradeDataTradeOpt.Shares .* ...
    TradeDataTradeOpt.Price) .* 10000;

% Percentage of volume, price appreciation and liquidity factor
TradeDataTradeOpt.POV = POV;
TradeDataTradeOpt.PA = PA;
TradeDataTradeOpt.LF = liquidityFactor(k,TradeDataTradeOpt);
```

Calculate trading costs in basis points, cents per share, and dollars.

```
% Build optimal cost table
OptimalCostTable = table(cell(3,1),zeros(3,1),zeros(3,1),zeros(3,1), ...
    zeros(3,1),'VariableNames',{'CostUnits','MI','PA','TotalCost','TR'});
OptimalCostTable.CostUnits(1) = {'Basis Points'};
OptimalCostTable.CostUnits(2) = {'Cents per Share'};
OptimalCostTable.CostUnits(3) = {'Dollars'};

% Market impact,
OptimalCostTable.MI(1) = TotMI;
OptimalCostTable.MI(2) = TotMI / 100 * mean(TradeDataTradeOpt.Price);
OptimalCostTable.MI(3) = TotMI / 100 * (TradeDataTradeOpt.Shares' * ...
    TradeDataTradeOpt.Price);

% Price appreciation
OptimalCostTable.PA(1) = TotPA;
OptimalCostTable.PA(2) = TotPA / 100 * mean(TradeDataTradeOpt.Price);
OptimalCostTable.PA(3) = TotPA / 100 * (TradeDataTradeOpt.Shares' * ...
    TradeDataTradeOpt.Price);
```

```
% Total cost
OptimalCostTable.TotalCost(1) = TotMI + TotPA;
OptimalCostTable.TotalCost(2) = (TotMI + TotPA) / 100 * mean(TradeDataTradeOpt.Price);
OptimalCostTable.TotalCost(3) = (TotMI + TotPA) / 100 * ...
    (TradeDataTradeOpt.Shares' * TradeDataTradeOpt.Price);

% Timing risk
OptimalCostTable.TR(1) = TotTR;
OptimalCostTable.TR(2) = TotTR / 100 * mean(TradeDataTradeOpt.Price);
OptimalCostTable.TR(3) = TotTR / 100 * ...
    (TradeDataTradeOpt.Shares' * TradeDataTradeOpt.Price);
```

Display the optimal costs for the basket. Format the display output to show cents and dollars. Optimal costs are market impact, price appreciation, total cost, and timing risk.

```
format bank
OptimalCostTable

OptimalCostTable =

  3×5 table array
```

CostUnits	MI	PA	TotalCost	TR
'Basis Points'	38.30	0.00	38.30	26.57
'Cents per Share'	14.88	0.00	14.88	10.32
'Dollars'	171134479.73	0.00	171134479.73	118710304.48

Determine Risk Components in Basket

Calculate risk statistics. The marginal contribution to risk captures the risk of changing one of the components in the basket, such as the number of shares. The risk contribution is the risk for each trade in the basket.

```
% Portfolio Risk in Dollars
PortfolioRisk = sqrt(TradeDataTradeOpt.Shares' * CD * ...
    TradeDataTradeOpt.Shares);

% MCR and RC calculations
PortfolioRiskMCR = zeros(numberStocks,1);
PortfolioRiskRC =zeros(numberStocks,1);
SharesMCR = TradeDataTradeOpt.Shares;
SharesRC = TradeDataTradeOpt.Shares;
for i = 1:numberStocks
```

```
  SharesMCR(i) = TradeDataTradeOpt.Shares(i) * 0.90;
  SharesRC(i) = 0;
  PortfolioRiskMCR(i) = sqrt(SharesMCR' * CD * SharesMCR);
  PortfolioRiskRC(i) = sqrt(SharesRC' * CD * SharesRC);
end
TradeDataTradeOpt.MCR = PortfolioRisk ./ PortfolioRiskMCR - 1;
TradeDataTradeOpt.RC = PortfolioRisk ./ PortfolioRiskRC - 1;
```

Display the side, symbol, and number of shares for the safest trade in the basket using the risk contribution.

```
minrisk = min(TradeDataTradeOpt.RC);
for i = 1:25
    if TradeDataTradeOpt.RC(i) == minrisk
        idx = i;
    end
end
[TradeDataTradeOpt.Side(idx) TradeDataTradeOpt.Symbol(idx) ...
    TradeDataTradeOpt.Shares(idx)]

ans =

  1×3 cell array

    'B'     'ABC'     [100000]
```

The buy order of 100,000 shares of stock ABC contributes the most overall portfolio risk.

Create Basket Report Summary

Create a table for the basket report summary.

```
% Get sector identifiers
uniqueSectors = unique(TradeDataTradeOpt.Sector);
numSectors = size(uniqueSectors,1);
numGroups = 14 + size(uniqueSectors,1);  % Using 14 categories plus number of sectors

% Preallocate BasketReport table
BasketReport = table;
BasketReport.BasketCategory = cell(numGroups,1);
BasketReport.Number = zeros(numGroups,1);
BasketReport.Weight = zeros(numGroups,1);
BasketReport.MI = zeros(numGroups,1);
BasketReport.TR = zeros(numGroups,1);
BasketReport.POV = zeros(numGroups,1);
```

```
BasketReport.TradeTime = zeros(numGroups,1);
BasketReport.PctADV = zeros(numGroups,1);
BasketReport.Price = zeros(numGroups,1);
BasketReport.Volatility = zeros(numGroups,1);
BasketReport.Risk = zeros(numGroups,1);
BasketReport.RC = zeros(numGroups,1);
BasketReport.MCR = zeros(numGroups,1);
BasketReport.Beta = zeros(numGroups,1);
BasketReport.LF = zeros(numGroups,1);
BasketReport.TotalValue = zeros(numGroups,1);
BasketReport.BuyValue = zeros(numGroups,1);
BasketReport.SellValue = zeros(numGroups,1);
BasketReport.NetValue = zeros(numGroups,1);
BasketReport.Shares = zeros(numGroups,1);
BasketReport.BuyShares = zeros(numGroups,1);
BasketReport.SellShares = zeros(numGroups,1);
```

Calculate the basket report summary.

Divide the trades in the basket into these categories:

- `Total` — All trades in basket
- `Buy` — Buy trades
- `Cover` — Buy trades that cover a short position
- `Sell` — Sell trades
- `Short` — Short trades
- `<=1%` — Trades that have percentage of average daily volume less than or equal to 1%
- `1%-3%` — Trades that have percentage of average daily volume between 1% and 3%
- `3%-5%` — Trades that have percentage of average daily volume between 3% and 5%
- `5%-10%` — Trades that have percentage of average daily volume between 5% and 10%
- `10%-20%` — Trades that have percentage of average daily volume between 10% and 20%
- `>20%` — Trades that have percentage of average daily volume greater than 20%
- `LC` — Large-capitalization stock trades
- `MC` — Mid-capitalization stock trades
- `SC` — Small-capitalization stock trades
- `Consumer Discretionary` — Trades in the consumer discretionary industry

- `Consumer Staples` — Trades in the consumer staples industry
- `Energy` — Trades in the energy industry
- `Financials` — Trades in the financial industry
- `Health Care` — Trades in the health care industry
- `Industrials` — Trades in the industrial industry
- `Information Technology` — Trades in the information technology industry
- `Materials` — Trades in the materials industry
- `Telecommunication Services` — Trades in the telecommunication services industry
- `Utilities` — Trades in the utilities industry

For stocks in each category, calculate these values:

- `Weight` — Total trade value weight
- `MI` — Weighted average market-impact cost
- `TR` — Timing risk
- `POV` — Weighted average percentage of volume rate
- `TradeTime` — Weighted average trade time to complete the order
- `PctADV` — Weighted average order size (measured as percentage of average daily volume)
- `Price` — Weighted average share price
- `Volatility` — Weighted average volatility
- `Risk` — Portfolio risk
- `RC` — Risk contribution to the overall portfolio risk (shows the amount of risk that an order contributes to the basket)
- `MCR` — Marginal contribution to risk (shows the amount of risk that 10% of shares in the order contribute to the basket)
- `Beta` — Weighted average beta
- `LF` — Weighted average liquidity factor
- `TotalValue` — Total trade value
- `BuyValue` — Total trade value of the buy transactions
- `SellValue` — Total trade value of the sell transactions

- `NetValue` — Difference between total trade value of the buy and sell transactions

- `Shares` — Number of shares

- `BuyShares` — Number of shares to buy

- `SellShares` — Number of shares to sell

```matlab
% Fill table, indRecord is index of matching TradeData rows
j = 0;
for i = 1:24

    switch i

      % Total
      case 1

        indRecord = true(numberStocks,1);
        BasketReport.BasketCategory(i) = {'Total'};

      % Side
      case 2
        indRecord = strcmp(TradeDataTradeOpt.Side,'B') | ...
            strcmp(TradeDataTradeOpt.Side,'Buy');
        BasketReport.BasketCategory(i) = {'Buy'};

      case 3
        indRecord = strcmp(TradeDataTradeOpt.Side,'C') | ...
            strcmp(TradeDataTradeOpt.Side,'Cover');
        BasketReport.BasketCategory(i) = {'Cover'};

      case 4
        indRecord = strcmp(TradeDataTradeOpt.Side,'S') | ...
            strcmp(TradeDataTradeOpt.Side,'Sell');
        BasketReport.BasketCategory(i) = {'Sell'};

      case 5
        indRecord = strcmp(TradeDataTradeOpt.Side,'SS') | ...
            strcmp(TradeDataTradeOpt.Side,'Short') | ...
            strcmp(TradeDataTradeOpt.Side,'Sell Short');
        BasketReport.BasketCategory(i) = {'Short'};

      % Liquidity Category
      case 6

        % Percentage of average daily volume is less than 1 %
```

```
  indRecord = (TradeDataTradeOpt.PctADV <= 0.01);
  BasketReport.BasketCategory(i) = {'<=1%'};

case 7

  % Percentage of average daily volume is between 1 and 3 %
  indRecord = (TradeDataTradeOpt.PctADV > 0.01 & ...
      TradeDataTradeOpt.PctADV <= 0.03);
  BasketReport.BasketCategory(i) = {'1%-3%'};

case 8

  % Percentage of average daily volume is between 3 and 5 %
  indRecord = (TradeDataTradeOpt.PctADV > 0.03 & ...
      TradeDataTradeOpt.PctADV <= 0.05);
  BasketReport.BasketCategory(i) = {'3%-5%'};

case 9

  % Percentage of average daily volume is between 5 and 10 %
  indRecord = (TradeDataTradeOpt.PctADV > 0.05 & ...
      TradeDataTradeOpt.PctADV <= 0.10);
  BasketReport.BasketCategory(i) = {'5%-10%'};

case 10

  % Percentage of average daily volume is between 10 and 20 %
  indRecord = (TradeDataTradeOpt.PctADV > 0.10 & ...
      TradeDataTradeOpt.PctADV <= 0.20);
  BasketReport.BasketCategory(i) = {'10%-20%'};

case 11

  % Percentage of average daily volume is greater than 20 %
  indRecord = (TradeDataTradeOpt.PctADV > 0.20);
  BasketReport.BasketCategory(i) = {'>20%'};

% Market cap
case 12

  % Large cap
  indRecord = (TradeDataTradeOpt.MktCap > 10000000000);
  BasketReport.BasketCategory(i) = {'LC'};
```

```matlab
  case 13

    % Mid cap
    indRecord = (TradeDataTradeOpt.MktCap > 1000000000 & ...
        TradeDataTradeOpt.MktCap <= 10000000000);
    BasketReport.BasketCategory(i) = {'MC'};

  case 14

    % Small cap
    indRecord = (TradeDataTradeOpt.MktCap <= 1000000000);
    BasketReport.BasketCategory(i)={'SC'};

  % Sectors
  % Description of basket category
  case {15, 16, 17, 18, 19, 20, 21, 22, 23, 24}
    j = j + 1;
    if j <= numSectors
      indRecord = strcmp(TradeDataTradeOpt.Sector,uniqueSectors(j));
      BasketReport.BasketCategory(i) = uniqueSectors(j);
    end

end

% Get subset of TradeData
TD = TradeDataTradeOpt(indRecord,:);

if ~isempty(TD)

    % Covariance Matrix in $/Shares^2
  CC2 = CC(indRecord,indRecord);       %Trading Period Covariance Matrix in $/Shares^2
  C2 = C1(indRecord,indRecord);        %Annualized Covariance Matrix in $/Shares^2
  RR = R(indRecord,:);                 %Residuals for Stocks in group

  % Basket Summary Calculations
  Weight2 = TD.Value / sum(TD.Value);

  % Side
  I_Buy = (TD.SideIndicator == 1);
  I_Sell = (TD.SideIndicator == -1);

  % Fill basket report table
  BasketReport.Number(i) = size(TD,1);                     % Number of records that
  BasketReport.Weight(i) = sum(TD.Value)/PortfolioValue;   % Weight of assets in cu
```

```matlab
BasketReport.MI(i) = Weight2' * TD.MI;                      % Market impact of asset
BasketReport.TR(i) = sqrt(trace(RR'*CC2*RR)) / sum(TD.Value) * 10000;   % Timing
BasketReport.POV(i) = Weight2' * TD.POV;                    % POV of assets
BasketReport.TradeTime(i) = Weight2' * TD.TradeTime;       % Tradetime of assets
BasketReport.PctADV(i) = Weight2' * TD.PctADV;             % Percentage of ADV
BasketReport.Price(i) = Weight2' * TD.Price;               % Total price of assets
BasketReport.Volatility(i) = Weight2' * TD.Volatility;     % Volatility
BasketReport.Risk(i) = sqrt(TD.Shares' * C2 * TD.Shares) / ...
    sum(TD.Value);    % Risk value

% RC and MCR
Shares2 = TradeDataTradeOpt.Shares;
Shares3 = TradeDataTradeOpt.Shares;
Shares2(indRecord) = 0;
Shares3(indRecord) = Shares3(indRecord) * 0.90;

if sum(Shares2) > 0
  BasketReport.RC(i) = PortfolioRisk / sqrt(Shares2' * CD * Shares2) - 1;
else
  BasketReport.RC(i) = 0;
end
BasketReport.MCR(i) = PortfolioRisk / sqrt(Shares3' * CD * Shares3) - 1;

% Beta value, liquidity factor and total value
BasketReport.Beta(i) = sum(Weight2 .* TD.SideIndicator .* TD.Beta);
BasketReport.LF(i) = Weight2' * TD.LF;
BasketReport.TotalValue(i) = sum(TD.Value);

% Calculate buy share values
if sum(I_Buy) > 0
  BasketReport.BuyValue(i) = sum(TD.Value(I_Buy));
  BasketReport.BuyShares(i) = sum(TD.Shares(I_Buy));
else
  BasketReport.BuyValue(i) = 0;
  BasketReport.BuyShares(i) = 0;
end

% Calculate sell share values
if sum(I_Sell) > 0
  BasketReport.SellValue(i) = sum(TD.Value(I_Sell));
  BasketReport.SellShares(i) = sum(TD.Shares(I_Sell));
else
  BasketReport.SellValue(i) = 0;
  BasketReport.SellShares(i) = 0;
```

```
        end

        % Calculate net value of criteria and number of shares
        BasketReport.NetValue(i) = BasketReport.BuyValue(i) - ...
            BasketReport.SellValue(i);
        BasketReport.Shares(i) = sum(TD.Shares);

    end
end

% Remove rows with no stocks
indRecord = (BasketReport.Number > 0);
BasketReport = BasketReport(indRecord,:);
```

Display market capitalization by volatility as a pie chart.

```
pie(BasketReport.Volatility(8:10),BasketReport.BasketCategory(8:10))
title('Market Capitalization by Volatility')
```
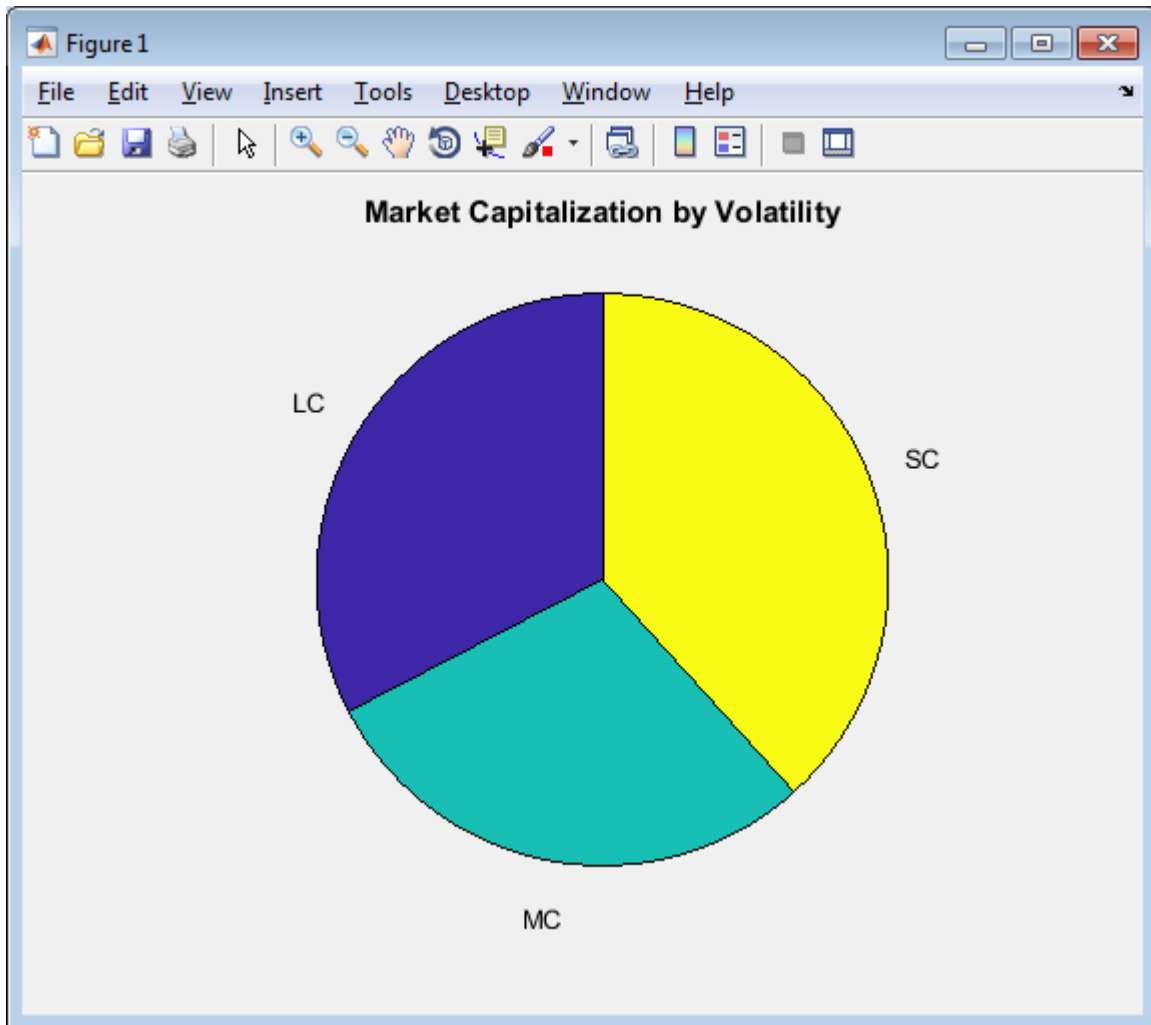
Create Principal Bid Summary

Determine the efficient trading frontier by time. Use different trade time scenarios. Estimate trading costs for price appreciation, market impact, and timing risk for each scenario.

```
ScenarioTime = [0.10;0.25;0.50;0.75;1.0;1.50;2.0;2.5;3.0;3.5;4.0;4.5;5.0];
numScenarios = size(ScenarioTime,1);
```

```matlab
ETFCosts = zeros(numScenarios,5);

TableVariableNames = TradeDataTradeOpt.Properties.VariableNames;
if sum(strcmp(TableVariableNames,'DeltaP')) > 0
    DeltaP = TradeDataTradeOpt.DeltaP;
elseif sum(strcmp(TableVariableNames,'Alpha_bp')) > 0
    DeltaP = TradeDataTradeOpt.Alpha_bp;
else
    DeltaP = zeros(NumberStocks,1);
end

% Convert DeltaP from basis points per day to cents/share per period
DeltaP = DeltaP / 1000 .* TradeDataTradeOpt.Price / totalNumberPeriods;

for i = 1:numScenarios

    TradeTime = ScenarioTime(i);
    TradeDataTradeOpt.POV = TradeDataTradeOpt.Shares ./ ...
        (TradeDataTradeOpt.Shares + TradeTime .* TradeDataTradeOpt.ADV);

    % Price Appreciations in Dollars
    PA = 1/2 * TradeDataTradeOpt.Shares .* DeltaP .* TradeTime;
    TotPA = sum(PA) / (TradeDataTradeOpt.Shares' * ...
        TradeDataTradeOpt.Price) .* 10000;         % bp
    PA = PA ./ (TradeDataTradeOpt.Shares .* ...
        TradeDataTradeOpt.Price) * 10000;          % bp

    % Market Impact in Dollars
    MI = marketImpact(k,TradeDataTradeOpt) .* TradeDataTradeOpt.Shares .* ...
        TradeDataTradeOpt.Price ./ 10000; %dollars;
    TotMI = sum(MI) / (TradeDataTradeOpt.Shares' * ...
        TradeDataTradeOpt.Price) .* 10000;         % bp
    MI = MI ./ (TradeDataTradeOpt.Shares .* ...
        TradeDataTradeOpt.Price) * 10000;          % bp

    % Timing Risk in Dollars
    TotTR = sqrt(1/3 * TradeDataTradeOpt.Shares' * ...
        (CD * TradeTime) * TradeDataTradeOpt.Shares) / ...
        (TradeDataTradeOpt.Shares' * TradeDataTradeOpt.Price) * 10000;

    % Total Cost Dollars
    TotTC = (TotMI + TotPA);

    % ETF Cost Table
```

```
    ETFCosts(i,1) = TradeTime;
    ETFCosts(i,2) = TotMI;
    ETFCosts(i,3) = TotPA;
    ETFCosts(i,4) = TotTC;
    ETFCosts(i,5) = TotTR;

end

% Save as Table
ETFCosts = table(ETFCosts(:,1),ETFCosts(:,2),ETFCosts(:,3),ETFCosts(:,4), ...
    ETFCosts(:,5),'VariableNames',{'Days','MI_bp','PA_bp','TotalCost_bp', ...
    'TR_bp'});
```

Determine the trade time with the lowest total cost.

```
mintotcost = min(ETFCosts.TotalCost_bp);
for i = 1:numScenarios
    if(ETFCosts.TotalCost_bp(i) == mintotcost)
        scenario = ETFCosts.Days(i);
    end
end
scenario

scenario =

     5
```

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management.* Cambridge, MA: Elsevier/Academic Press, 2013.

[2] Malamut, Roberto. "Multi-Period Optimization Techniques for Trade Scheduling." Presentation at the QWAFAFEW New York Conference, April 2002.

[3] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies.* New York, NY: AMACOM, Inc., 2003.

See Also
krg | liquidityFactor | marketImpact

More About

- "Optimize Trade Schedule Trading Strategy for Basket" on page 3-80
- "Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22
- "Liquidate Dollar Value from Portfolio" on page 3-56
- "Kissell Research Group Data Sets" on page 3-9

4

Sample Code for Workflows

Listen for X_TRADER Price Updates

This example shows how to connect to X_TRADER and listen for price update event data.

Connect to X_TRADER

```
X = xtrdr;
```

Create an Event Notifier

The event notifier is the X_TRADER mechanism that lets you define MATLAB functions to use as callbacks for specific events.

```
createNotifier(X)
```

Create an Instrument

Create an instrument and attach it to the notifier.

```
createInstrument(X,'Exchange','CME','Product','2F',...
                 'ProdType','Future','Contract','Aug13',...
                 'Alias','PriceInstrument1')
X.InstrNotify(1).AttachInstrument(X.Instrument(1))
```

Define Events

Assign callbacks for validating or invalidating an instrument, and for handling data updates for a previously validated instrument.

```
registerevent(X.InstrNotify(1),{'OnNotifyFound',...
                      @(varargin)ttinstrumentfound(varargin{:})})
registerevent(X.InstrNotify(1),{'OnNotifyNotFound',...
                      @(varargin)ttinstrumentnotfound(varargin{:})})
registerevent(X.InstrNotify(1),{'OnNotifyUpdate',...
                      @(varargin)ttinstrumentupdate(varargin{:})})
```

Monitor Events

Set the update filter to monitor the desired fields. In this example, events are monitored for updates to last price, last quantity, previous last quantity, and a change in prices. Listen for this event data.

```
X.InstrNotify(1).UpdateFilter = 'Last$,LastQty$,~LastQty$,Change$';
X.Instrument(1).Open(0)
```

The last command tells X_TRADER to start monitoring the attached instruments using the specified event settings.

Close the Connection

```
close(X)
```

See Also

close | createInstrument | createNotifier | xtrdr

Related Examples

- "Create an Order Using X_TRADER" on page 1-17
- "Listen for X_TRADER Price Market Depth Updates" on page 4-4
- "Submit X_TRADER Orders" on page 4-8

More About

- "Workflows for Trading Technologies X_TRADER" on page 2-4

Listen for X_TRADER Price Market Depth Updates

This example shows how to connect to X_TRADER and turn on event handling for level-two market data (for example, bid and ask orders in the market for an instrument) and then create a figure window to display the depth data.

Connect to X_TRADER

```
X = xtrdr;
```

Create an Event Notifier

Create an event notifier and enable depth updates. The event notifier is the X_TRADER mechanism lets you define MATLAB functions to use as callbacks for specific events.

```
createNotifier(X)
X.InstrNotify(1).EnableDepthUpdates = 1;
```

Create an Instrument

```
createInstrument(X,'Exchange','CME','Product','2F','ProdType','Future',...
                'Contract','Aug13','Alias','PriceInstrumentDepthUpdate')
```

Attach an Instrument to a Notifier

Assign one or more notifiers to an instrument. A notifier can have one or more instruments attached to it.

```
X.InstrNotify(1).AttachInstrument(X.Instrument(1))
```

Define Events

Assign callbacks for validating or invalidating an instrument, and updating the example order book window.

```
registerevent(X.InstrNotify(1),{'OnNotifyFound',...
                                @ttinstrumentfound})
registerevent(X.InstrNotify(1),{'OnNotifyNotFound',...
                                @ttinstrumentnotfound})
registerevent(X.InstrNotify(1),{'OnNotifyDepthData',...
                                @ttinstrumentdepthupdate})
```

Set Up the Figure Window

Set up the figure window to display depth data.

```
f = figure('Numbertitle','off','Tag','TTPriceUpdateDepthFigure',...
```

```
        'Name',['Order Book - ' X.Instrument(1).Alias])
pos = f.Position;
f.Position = [pos(1) pos(2) 360 315];
f.Resize = 'off';
```

Create Controls

Create controls for the last price data.

```
bspc = 5;
bwid = 80;
bhgt = 20;

uicontrol('Style','text','String','Exchange',...
          'Position',[bspc 4*bspc+3*bhgt bwid bhgt])
uicontrol('Style','text','String','Product',...
          'Position',[2*bspc+bwid 4*bspc+3*bhgt bwid bhgt])
uicontrol('Style','text','String','Type',...
          'Position',[3*bspc+2*bwid 4*bspc+3*bhgt bwid bhgt])
uicontrol('Style','text','String','Contract',...
          'Position',[4*bspc+3*bwid 4*bspc+3*bhgt bwid bhgt])
ui.Exchange = uicontrol('Style','text','Tag','',...
          'Position',[bspc 3*bspc+2*bhgt bwid bhgt]);
ui.Product = uicontrol('Style','text','Tag','',...
          'Position',[2*bspc+bwid 3*bspc+2*bhgt bwid bhgt]);
ui.Type = uicontrol('Style','text','Tag','',...
          'Position',[3*bspc+2*bwid 3*bspc+2*bhgt bwid bhgt]);
ui.Contract = uicontrol('Style','text','Tag','',...
          'Position',[4*bspc+3*bwid 3*bspc+2*bhgt bwid bhgt]);
uicontrol('Style','text','String','Last Price',...
          'Position',[bspc 2*bspc+bhgt bwid bhgt])
uicontrol('Style','text','String','Last Qty',...
          'Position',[2*bspc+bwid 2*bspc+bhgt bwid bhgt])
uicontrol('Style','text','String','Change',...
          'Position',[3*bspc+2*bwid 2*bspc+bhgt bwid bhgt])
ui.Last = uicontrol('Style','text','Tag','',...
          'Position',[bspc bspc bwid bhgt]);
ui.Quantity = uicontrol('Style','text','Tag','',...
          'Position',[2*bspc+bwid bspc bwid bhgt]);
ui.Change = uicontrol('Style','text','Tag','',...
          'Position',[3*bspc+2*bwid bspc bwid bhgt]);
```

Create a Table

Create a table containing order information.

```
data = {' '};
data = data(ones(10,4));
uibook = uitable('Data',data,'ColumnName',...
                 {'Bid','Bid Size','Ask','Ask Size'},...
                  'Position',[5 105 350 205]);
```
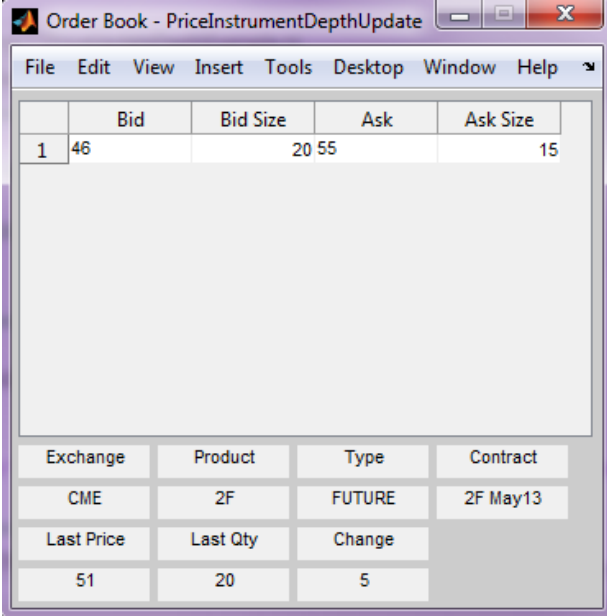
Store Data

```
setappdata(0,'TTOrderBookHandle',uibook)
setappdata(0,'TTOrderBookUIData',ui)
```

Listen for Event Data

Listen for event data with depth updates enabled.

```
X.Instrument(1).Open(1)
```


The last command instructs X_TRADER to start monitoring the attached instruments using the specified event settings.

Close the Connection

```
close(X)
```

See Also
close | createInstrument | createNotifier | getData | xtrdr

Related Examples
- "Create an Order Using X_TRADER" on page 1-17
- "Listen for X_TRADER Price Updates" on page 4-2
- "Submit X_TRADER Orders" on page 4-8

More About
- "Workflows for Trading Technologies X_TRADER" on page 2-4

Submit X_TRADER Orders

This example shows how to connect to X_TRADER and submit orders.

Connect to X_TRADER

```
X = xtrdr;
```

Create an Instrument

```
createInstrument(X,'Exchange','CME','Product','2F',...
                 'ProdType','Future','Contract','Aug13',...
                 'Alias','SubmitOrderInstrument1')
```

Register Event Handlers

Register event handlers for the order server. The callback `ttorderserverstatus` is assigned to the event `OnExchangeStateUpdate` to verify that the requested instrument's exchange order server is running. Otherwise, no orders can be submitted.

```
sExchange = X.Instrument.Exchange;
registerevent(X.Gate,{'OnExchangeStateUpdate',...
     @(varargin)ttorderserverstatus(varargin{:},sExchange)})
```

Create an Order Set

The `OrderSet` object sends orders to X_TRADER.

Set properties of the `OrderSet` object and detail the level of the order status events. Enable order update and reject (failure) events so you can assign callbacks to handle these conditions.

```
createOrderSet(X)
X.OrderSet(1).EnableOrderRejectData = 1;
X.OrderSet(1).EnableOrderUpdateData = 1;
X.OrderSet(1).OrderStatusNotifyMode = 'ORD_NOTIFY_NORMAL';
```

Set Position Limit Checks

Set whether the order set checks self-imposed position limits when submitting an order.

```
X.OrderSet(1).Set('NetLimits',false)
```

Set a Callback Function

Set a callback to handle the `OnOrderFilled` events. Each time an order is filled (or partially filled), this callback is invoked.

```
registerevent(X.OrderSet(1),{'OnOrderFilled',...
                    @(varargin)ttorderevent(varargin{:},X)})
```

Enable Order Submission

You must first enable order submission before you can submit orders to X_TRADER.

```
X.OrderSet(1).Open(1)
```

Build an Order Profile

Build an order profile using an existing instrument. The order profile contains the settings that define a submitted order. The valid `Set` parameters are shown:

```
orderProfile = createOrderProfile(X);
orderProfile.Instrument = X.Instrument(1);
orderProfile.Customer = '<Default>';
```

Sample: Create a Market Order

Create a market order to buy 100 shares.

```
orderProfile.Set('BuySell','Buy')
orderProfile.Set('Qty',100)
orderProfile.Set('OrderType','M')
```

Sample: Create a Limit Order

Create a limit order by setting the `OrderType` and limit order price.

```
orderProfile.Set('OrderType','L')
orderProfile.Set('Limit$','127000')
```

Sample: Create a Stop Market Order

Create a stop market order and set the order restriction to a stop order and a stop price.

```
orderProfile.Set('OrderType','M')
orderProfile.Set('OrderRestr','S')
orderProfile.Set('Stop$','129800')
```

Sample: Create a Stop Limit Order

Create a stop limit order and set the order restriction, type, limit price, and stop price.

```
orderProfile.Set('OrderType','L')
orderProfile.Set('OrderRestr','S')
orderProfile.Set('Limit$','128000')
orderProfile.Set('Stop$','127500')
```

Check the Order Server Status

Check the order server status before submitting the order and add a counter so the example doesn't delay.

```
nCounter = 1;
while ~exist('bServerUp','var') && nCounter < 20
   pause(1)
   nCounter = nCounter + 1;
end
```

Verify the Order Server Availability

Verify that the exchange's order server in question is available before submitting the order.

```
if exist('bServerUp','var') && bServerUp
   submittedQuantity = X.OrderSet(1).SendOrder(orderProfile);
   disp(['Quantity Sent: ' num2str(submittedQuantity)])
else
   disp('Order Server is down. Unable to submit order')
end
```

Close the Connection

```
close(X)
```

See Also
close | createInstrument | createOrderProfile | createOrderSet | xtrdr

Related Examples
- "Create an Order Using X_TRADER" on page 1-17
- "Listen for X_TRADER Price Updates" on page 4-2

·   "Listen for X_TRADER Price Market Depth Updates" on page 4-4

More About

·   "Workflows for Trading Technologies X_TRADER" on page 2-4

Create and Manage a Bloomberg EMSX Order

This example shows how to connect to Bloomberg EMSX, create an order, and interact with the order.

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg EMSX

Connect to the Bloomberg EMSX test service. Display the current event queue contents using `processEvent`.

```
c = emsx('//blp/emapisvc_beta');
processEvent(c)

c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194

SessionConnectionUp = {
    server = localhost/127.0.0.1:8194
}

SessionStarted = {
}

ServiceOpened = {
    serviceName = //blp/emapisvc_beta
}
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

`processEvent` displays events associated with connecting to Bloomberg EMSX.

Set Up the Order Subscription

Subscribe to order events using the Bloomberg EMSX connection `c` associated with these Bloomberg EMSX fields.

```
fields = {'EMSX_TICKER','EMSX_AMOUNT','EMSX_FILL'};

[events,subs] = orders(c,fields)

events =

                      MSG_TYPE: {'E'}
                  MSG_SUB_TYPE: {'O'}
                  EVENT_STATUS: 4
                  ...

subs =

com.bloomberglp.blpapi.SubscriptionList@4bc3dc78
```

`events` contains fields for the events associated with the existing Bloomberg EMSX orders. `subs` contains the Bloomberg EMSX subscription list object.

Create the Order

Create an order request structure `order` for a buy market order of 400 shares of IBM. Specify the broker as `EFIX`, use any hand instruction, and set the time in force to `DAY`.

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_SIDE = 'BUY';
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(400);
order.EMSX_BROKER = 'EFIX';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_TIF = 'DAY';
```

Create the order using the Bloomberg EMSX connection `c` and the order request structure `order`.

```
events = createOrder(c,order)

order_events =

  EMSX_SEQUENCE: 354646
```

```
MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Modify the Order

Define the structure `modorder` that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`

This code modifies order number 354646 for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(354646);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Modify the order using the Bloomberg EMSX connection `c` and modify order structure `modorder`.

```
events = modifyOrder(c,modorder)

events =

    EMSX_SEQUENCE: 354646
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying an order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Delete the Order

Define the structure `ordernum` that contains the order sequence number 354646 for the order to delete. Delete the order using the Bloomberg EMSX connection `c` and the delete order number structure `ordernum`.

```
ordernum.EMSX_SEQUENCE = 354646;

events = deleteOrder(c,ordernum)

events =

     STATUS: 'O'
    MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting an order. `deleteOrder` returns `events` as a structure that contains these fields:

· Bloomberg EMSX status
· Bloomberg EMSX message

Stop the Order Subscription

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX Connection

```
close(c)
```

See Also
close | createOrder | deleteOrder | emsx | modifyOrder | orders

Related Examples
· "Create an Order Using Bloomberg EMSX" on page 1-14
· "Create and Manage a Bloomberg EMSX Route" on page 4-16
· "Manage a Bloomberg EMSX Order and Route" on page 4-21

More About
· "Workflow for Bloomberg EMSX" on page 2-2

Create and Manage a Bloomberg EMSX Route

This example shows how to connect to Bloomberg EMSX, set up a route subscription, create and route an order, and interact with the route.

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg EMSX

Connect to the Bloomberg EMSX test service. Display the current event queue contents using processEvent.

```
c = emsx('//blp/emapisvc_beta');
processEvent(c)

c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194

SessionConnectionUp = {
    server = localhost/127.0.0.1:8194
}

SessionStarted = {
}

ServiceOpened = {
    serviceName = //blp/emapisvc_beta
}
```

MATLAB returns c as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

`processEvent` displays events associated with connecting to Bloomberg EMSX.

Set Up the Route Subscription

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Return the status for existing routes.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,subs] = routes(c,fields)

events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...

subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Create and Route the Order

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and the order request structure `order`.

```
events = createOrderAndRoute(c,order)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Modify the Route

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 50 shares of IBM for order sequence number 335877 and route identifier 1. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(335877)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(50);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and modify route request `modroute`.

```
events = modifyRoute(c,modroute)

events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Delete the Modified Route

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` and the route number `EMSX_ROUTE_ID` associated with the modified route.

```
routenum.EMSX_SEQUENCE = O;
routenum.EMSX_ROUTE_ID = O;
```

Delete the route using the Bloomberg EMSX connection `c` and delete route number structure `routenum`.

```
events = deleteRoute(c,routenum)

events =

    STATUS: '1'
   MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting a route. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Stop the Route Subscription

Unsubscribe from route events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX Connection

```
close(c)
```

See Also
```
close | createOrderAndRoute | deleteRoute | emsx | modifyRoute |
routeOrder | routes
```

Related Examples

More About

Manage a Bloomberg EMSX Order and Route

This example shows how to connect to Bloomberg EMSX, set up an order and route
subscription, create and route an order, and interact with the route.

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the
WAPI <GO> option from the Bloomberg terminal.

Connect to Bloomberg EMSX

Connect to the Bloomberg EMSX test service. Display the current event queue contents
using `processEvent`.

```
c = emsx('//blp/emapisvc_beta');
processEvent(c)

c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194

SessionConnectionUp = {
    server = localhost/127.0.0.1:8194
}

SessionStarted = {
}

ServiceOpened = {
    serviceName = //blp/emapisvc_beta
}
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the
following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service

- Port number of the machine running the Bloomberg EMSX test service

`processEvent` displays events associated with connecting to Bloomberg EMSX.

Set Up the Order and Route Subscription

Subscribe to order events using the Bloomberg EMSX connection `c` associated with these Bloomberg EMSX fields.

```
fields = {'EMSX_TICKER','EMSX_AMOUNT','EMSX_FILL'};

[events,osubs] = orders(c,fields)

events =

            MSG_TYPE: {'E'}
        MSG_SUB_TYPE: {'O'}
        EVENT_STATUS: 4
        ...

osubs =

com.bloomberglp.blpapi.SubscriptionList@4bc3dc78
```

`events` contains fields for the events associated with the existing Bloomberg EMSX orders. `osubs` contains the Bloomberg EMSX subscription list object.

Subscribe to route events for the Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Return the status for existing routes.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,rsubs] = routes(c,fields)

events =

            MSG_TYPE: {5x1 cell}
        MSG_SUB_TYPE: {5x1 cell}
        EVENT_STATUS: [5x1 int32]
        ...

rsubs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

events contains fields for the events currently in the event queue. rsubs contains the Bloomberg EMSX subscription list object.

Create and Route the Order

Create the order request structure order to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker BB with the time in force set to DAY and any hand instruction. Convert the number of shares to a 32-bit signed integer using int32.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection c and the order request structure order.

```
events = createOrderAndRoute(c,order)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. createOrderAndRoute returns events as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Modify the Route

Define the modroute structure that contains these fields:

- Bloomberg EMSX order sequence number EMSX_SEQUENCE
- Bloomberg EMSX ticker symbol EMSX_TICKER
- Bloomberg EMSX number of shares EMSX_AMOUNT

- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 50 shares of IBM for order sequence number `335877` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(335877)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(50);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and modify route request `modroute`.

```
events = modifyRoute(c,modroute)

events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Delete the Route

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 0;
routenum.EMSX_ROUTE_ID = 0;
```

Delete the route using the Bloomberg EMSX connection `c` and delete route number structure `routenum`.

```
events = deleteRoute(c,routenum)

events =

    STATUS: '1'
```

```
MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting a route. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Stop the Order and Route Subscription

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX Connection

```
close(c)
```

See Also
`close` | `createOrderAndRoute` | `deleteRoute` | `emsx` | `modifyRoute` | `orders` | `routes`

Related Examples
- "Create an Order Using Bloomberg EMSX" on page 1-14
- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16

More About
- "Workflow for Bloomberg EMSX" on page 2-2

Create and Manage an Interactive Brokers Order

This example shows how to connect to the IB Trader Workstation, request open order data, create IB Trader Workstation `IContract` and `IOrder` objects, and execute the order. For details about the `IContract` and `IOrder` objects, see Interactive Brokers API Reference Guide.

This example uses the sample event handler function `ibExampleOrderEventHandler` to populate an order blotter figure with Interactive Brokers order information. Use this event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

To access the code for this example, enter `edit IBOrderWorkflow.m`.

Connect to the IB Trader Workstation

Connect to the IB Trader Workstation and create connection `ib` using the local host and port number `7496`.

```
ib = ibtws('',7496);
```

Create an Example Order Blotter

Create an example order blotter that the event handler populates.

This MATLAB code creates a MATLAB figure to contain the Interactive Brokers order information.

```
f = findobj('Tag','IBOrderBlotter');
if isempty(f)
  f = figure('Tag','IBOrderBlotter','MenuBar','none',...
    'NumberTitle','off','Name','IB Order Blotter')
  pos = f.Position;
  f.Position = [pos(1) pos(2) 687 335];
  colnames = {'Status','Filled','Remaining','Avg Fill Price','Id',...
    'Parent Id','Last Fill Price','Client Id','Why Held'};
  data = cell(15,9);
  uitable(f,'Data',data,'RowName',[],'ColumnName',colnames,...
    'Position',[10 30 677 300],'Tag','OrderDataTable')
  uicontrol('Style','text','Position',[10 5 592 20],...
    'Tag','IBOrderMessage')
  uicontrol('Style','pushbutton','String','Close',...
    'Callback','evalin(''base'',''close(ib);close(findobj(''''Tag'''',''''IBOrderBlotter''''));'')',...
    'Position',[607 5 80 20])
end
```

MATLAB displays the IB Order Blotter.

Request Open Order Data

Request information for all open orders using only this client and the sample event handler `ibExampleOrderEventHandler`.

```
o = orders(ib,true,@ibExampleOrderEventHandler);
```

`o` is an empty double because `ibExampleOrderEventHandler` displays the data for all open orders in the IB Order Blotter.

Create the IB Trader Workstation `IContract` and `IOrder` Objects

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- XYZ symbol
- Stock security type
- Aggregate exchange
- Primary exchange
- USD currency

XYZ is a sample symbol name and EX is a sample primary exchange name. To create orders for your security, substitute the symbol name in `ibContract.symbol` and primary exchange name in `ibContract.primaryExchange`.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'XYZ';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'EX';
ibContract.currency = 'USD'
```

```
ibContract =

 Interface.Tws_ActiveX_Control_module.IContract
```

Create the IB Trader Workstation `IOrder` object `ibOrder` for a buy market order for two shares.

```
ibOrder = ib.Handle.createOrder;
ibOrder.action = 'BUY';
ibOrder.totalQuantity = 2;
ibOrder.orderType = 'MKT'

ibOrder =

 Interface.Tws_ActiveX_Control_module.IOrder
```

`ibOrder` contains the action, total quantity, and order type.

Create the Interactive Brokers Order
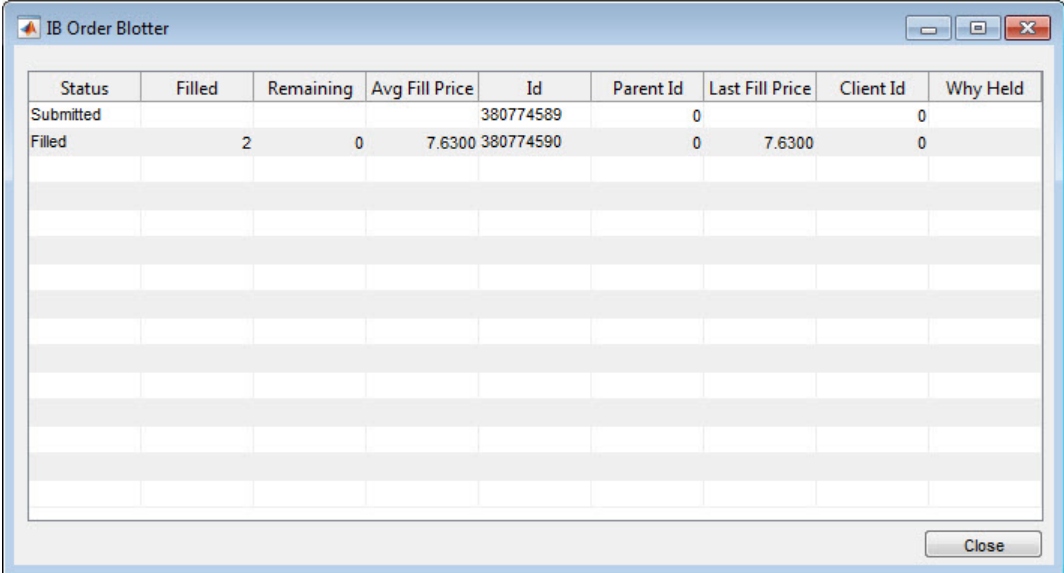
Obtain the next valid order identification number using IB Trader Workstation connection `ib`.

```
id = orderid(ib);
```

Execute the buy market order for two shares using the unique order identifier `id` and sample event handler `ibExampleOrderEventHandler`.

```
createOrder(ib,ibContract,ibOrder,id,@ibExampleOrderEventHandler)
```

MATLAB displays order information in the IB Order Blotter. The IB Order Blotter shows the open order and the filled order.

Cancel the Interactive Brokers Order

```
ib.Handle.cancelOrder(id)
```

After canceling the existing order, create an order by modifying the IB Trader Workstation `IOrder` object `ibOrder`. Then, create the order by executing `createOrder`.

Cancel all open Interactive Brokers orders.

```
ib.Handle.reqGlobalCancel
```

This method cancels all open Interactive Brokers orders globally. The order is canceled despite where it is created.

Close the Connection

Close the IB Trader Workstation connection `ib`.

```
close(ib)
```

See Also
```
close | createOrder | getdata | history | ibtws | orderid | orders |
timeseries
```

Related Examples

- "Create an Order Using IB Trader Workstation" on page 1-8
- "Create Interactive Brokers Combination Order" on page 4-39
- "Request Interactive Brokers Historical Data" on page 4-32
- "Request Interactive Brokers Real-Time Data" on page 4-35

More About

- "Workflow for Interactive Brokers" on page 2-6
- "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites

- Interactive Brokers API Reference Guide

Request Interactive Brokers Historical Data

This example shows how to connect to the IB Trader Workstation, create an IB Trader Workstation `IContract` object, and request historical data. For details about the `IContract` object, see Interactive Brokers API Reference Guide. To access the code for this example, enter `edit IBHistoricalDataWorkflow.m`.

Connect to the IB Trader Workstation and Create the `IContract` Object

Connect to the IB Trader Workstation and create connection `ib` using the local host and port number `7496`.
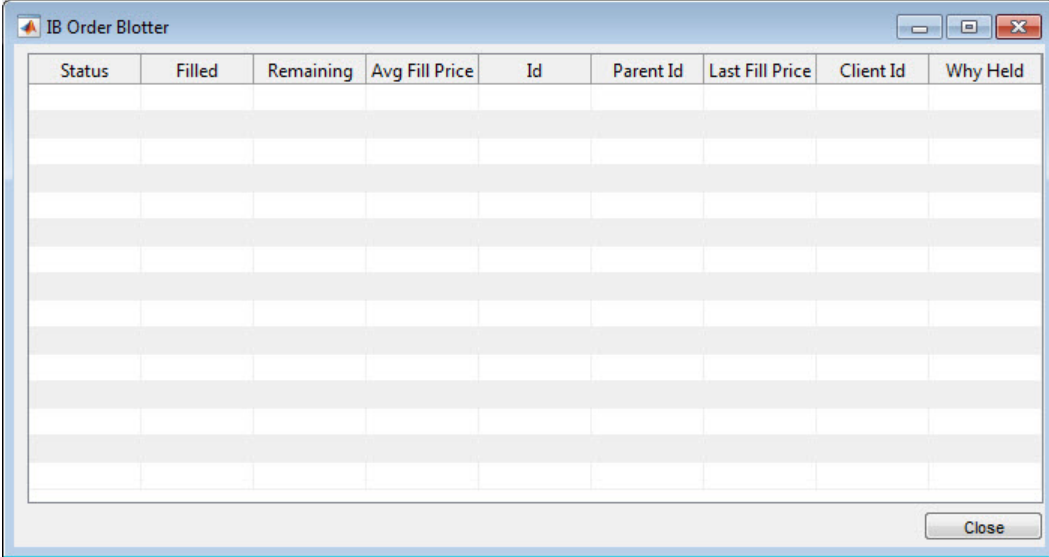
```
ib = ibtws('',7496);
```

MATLAB returns `ib` as the connection to the IB Trader Workstation with the Interactive Brokers ActiveX® object, the local host, and the port number that you choose.

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- XYZ symbol
- Stock security type
- Aggregate exchange
- Primary exchange
- USD currency

XYZ is a sample symbol name and EX is a sample primary exchange name. To create orders for your security, substitute the symbol name in `ibContract.symbol` and primary exchange name in `ibContract.primaryExchange`.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'XYZ';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'EX';
ibContract.currency = 'USD'

ibContract =

 Interface.Tws_ActiveX_Control_module.IContract
```

Request Interactive Brokers Historical Data

Request the last 5 days of historical data using `ibContract`.

```
startdate = floor(now) - 5;
enddate = floor(now);

d = history(ib,ibContract,startdate,enddate)
d =

  Columns 1 through 5

    736308.00        751.83        755.85        743.83        749.46
    736309.00        742.69        745.71        736.75        738.20
    736312.00        743.08        748.73        724.17        748.48
    736313.00        752.50        758.08        744.43        750.45

  Columns 6 through 9

    12513.00       9107.00        751.28             0
    15984.00      11121.00        740.39             0
    17125.00      11355.00        736.61             0
     1935.00       2371.00        751.67             0
```

`d` contains the historical data for 5 days.

Each row of `d` contains historical data for 1 day. The columns in matrix `d` are:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Close the Connection

Close the IB Trader Workstation connection `ib`.

```
close(ib)
```

See Also
close | createOrder | getdata | history | ibtws | timeseries

Related Examples
- "Create an Order Using IB Trader Workstation" on page 1-8
- "Create Interactive Brokers Combination Order" on page 4-39
- "Create and Manage an Interactive Brokers Order" on page 4-26
- "Request Interactive Brokers Real-Time Data" on page 4-35

More About
- "Workflow for Interactive Brokers" on page 2-6

External Websites
- Interactive Brokers API Reference Guide

Request Interactive Brokers Real-Time Data

This example shows how to connect to the IB Trader Workstation, create IB Trader Workstation `IContract` objects, and request real-time data. For details about the `IContract` object, see Interactive Brokers API Reference Guide.

This example uses the sample event handler function `ibExampleRealtimeEventHandler` to handle events associated with requesting real-time data. Use this event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Here, AAA, BBB, and DDDD are sample symbol names. EX is a sample primary exchange name. To create orders for your securities, substitute symbol names in `ibContract.symbol` and primary exchange names in `ibContract.primaryExchange`.

To access the code for this example, enter `edit IBStreamingDataWorkflow.m`.

Connect to the IB Trader Workstation and Create the Real-Time Data Display Figure

Connect to the IB Trader Workstation and create connection `ib` using the local host and port number 7496.

```
ib = ibtws('',7496);
```

MATLAB returns `ib` as the connection to the IB Trader Workstation with the Interactive Brokers ActiveX object, the local host, and the port number that you choose.

To display real-time data, create an example figure.

This MATLAB code creates a MATLAB figure to contain the Interactive Brokers real-time data.
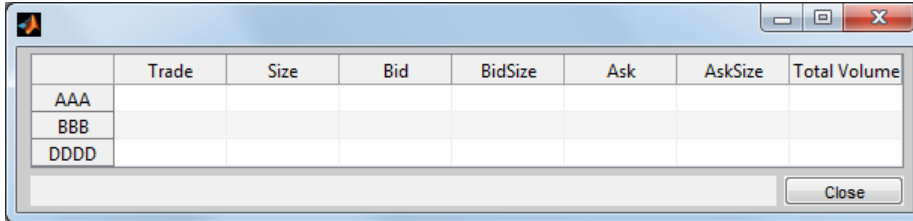
```
f = findobj('Tag','IBStreamingDataWorkflow');
if isempty(f)
  f = figure('Tag','IBStreamingDataWorkflow','MenuBar','none',...
     'NumberTitle','off')
  pos = f.Position;
  f.Position = [pos(1) pos(2) pos(3)+37 109];
  colnames = {'Trade','Size','Bid','BidSize','Ask','AskSize',...
     'Total Volume'};
  rownames = {'AAA','BBB','DDDD'};
  data = cell(3,6);
  uitable(f,'Data',data,'RowName',rownames,'ColumnName',colnames,...
  'Position',[10 30 582 76],'Tag','SecurityDataTable')
  uicontrol('Style','text','Position',[10 5 497 20],'Tag','IBMessage')
```

```matlab
uicontrol('Style','pushbutton','String','Close',...
   'Callback',...
   'evalin(''base'',''close(ib);close(findobj(''''Tag'''',''''IBStreamingDataWorkflow''''));'')',...
   'Position',[512 5 80 20])
end
```

MATLAB displays the empty figure.

Create IB Trader Workstation `IContract` Objects

Create the IB Trader Workstation `IContract` object for the first security. Here, this object describes a security with these property values:

- AAA symbol
- Stock security type
- Aggregate exchange
- Primary exchange
- USD currency

```matlab
ibContract1 = ib.Handle.createContract;
ibContract1.symbol = 'AAA';
ibContract1.secType = 'STK';
ibContract1.exchange = 'SMART';
ibContract1.primaryExchange = 'EX';
ibContract1.currency = 'USD';
```

Create the IB Trader Workstation `IContract` object for the second security symbol BBB.

```matlab
ibContract2 = ib.Handle.createContract;
ibContract2.symbol = 'BBB';
ibContract2.secType = 'STK';
ibContract2.exchange = 'SMART';
ibContract2.primaryExchange = 'EX';
ibContract2.currency = 'USD';
```

Create the IB Trader Workstation `IContract` object for the third security symbol DDDD.

```
ibContract3 = ib.Handle.createContract;
ibContract3.symbol = 'DDDD';
ibContract3.secType = 'STK';
ibContract3.exchange = 'SMART';
ibContract3.primaryExchange = 'EX';
ibContract3.currency = 'USD';
```

Display the data in the `symbol` property of `ibContract1`.

```
ibContract1.symbol
```

```
ans =
    AAA
```

Request real-time data for the three securities. Set `f` to `100` to retrieve the Option Volume tick type. For details about other generic market data tick types, see Interactive Brokers API Reference Guide. Use the sample event handler `ibExampleRealtimeEventHandler` to process the real-time data events or write a custom event handler function.
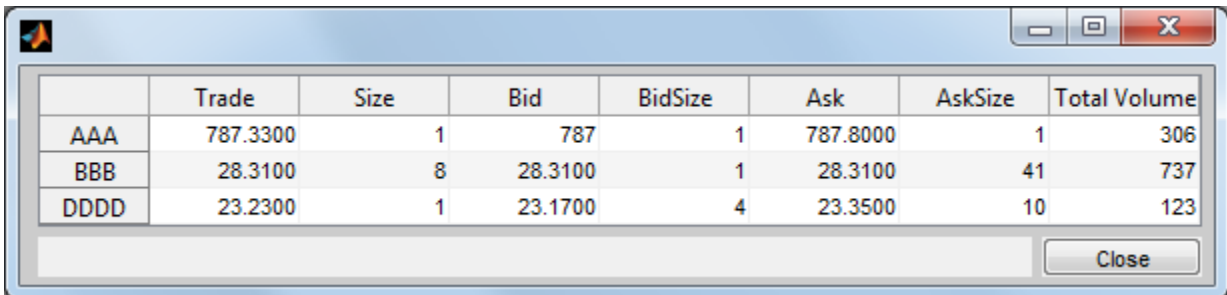
```
contracts = {ibContract1;ibContract2;ibContract3};
f = '100';

tickerID = realtime(ib,contracts,f,...
                    @(varargin)ibExampleRealtimeEventHandler(varargin{:}));
```

MATLAB displays the figure populated with real-time data for stock symbols AAA, BBB, and DDDD.

	Trade	Size	Bid	BidSize	Ask	AskSize	Total Volume
AAA	787.3300	1	787	1	787.8000	1	306
BBB	28.3100	8	28.3100	1	28.3100	41	737
DDDD	23.2300	1	23.1700	4	23.3500	10	123

Close the Connection

Close the IB Trader Workstation connection `ib`.

```
close(ib)
```

See Also
close | createOrder | getdata | history | ibtws | timeseries

Related Examples
- "Create an Order Using IB Trader Workstation" on page 1-8
- "Create Interactive Brokers Combination Order" on page 4-39
- "Create and Manage an Interactive Brokers Order" on page 4-26
- "Request Interactive Brokers Historical Data" on page 4-32

More About
- "Workflow for Interactive Brokers" on page 2-6
- "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites
- Interactive Brokers API Reference Guide

Create Interactive Brokers Combination Order

This example shows how to connect to the IB Trader Workstation, create IB Trader Workstation `IContract` and `IComboLegList` objects, and create a combination order for a calendar spread. A calendar spread is one of many combination order strategies. This strategy takes advantage of different stock option expiration dates. This example creates a buy order on a calendar spread for Google®. For details about `IContract` objects, `IComboLegList` objects, and combination orders, see Interactive Brokers API Reference Guide.

This example uses the sample event handler function `ibExampleEventHandler` to handle events associated with creating a combination order. Use this event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

To access the code for this example, enter `edit IBCombinationOrder.m`.

Connect to the IB Trader Workstation

Connect to the IB Trader Workstation and create connection `ib` using the local host and port number `7496`.

```
ib = ibtws('',7496);
```

MATLAB returns `ib` as the connection to the IB Trader Workstation with the Interactive Brokers ActiveX object, the local host, and the port number that you choose.

Create IB Trader Workstation `IContract` Objects

Create the IB Trader Workstation `IContract` object `ibContract1`. Here, this object describes the first call option in the calendar spread. Create an `IContract` object with these property values:

- Google symbol.
- Stock option.
- Expiry date is August 2014.
- Strike price is $535.00.
- Call option.
- Number of shares is 100.
- Aggregate exchange.

- Primary exchange
- USD currency.

Here, `EX` is a sample primary exchange name. Substitute your primary exchange name in `ibContract1.primaryExchange`.

```
ibContract1 = ib.Handle.createContract;
ibContract1.symbol = 'GOOG';
ibContract1.secType = 'OPT';
ibContract1.expiry = '201408';
ibContract1.strike = 535;
ibContract1.right = 'C';
ibContract1.multiplier = '100';
ibContract1.exchange = 'SMART';
ibContract1.primaryExchange = 'EX';
ibContract1.currency = 'USD';
```

Request contract details for `ibContract1`.

```
[cd1,ibReqID1] = contractdetails(ib,ibContract1);
```

`cd1` returns the contract details data for `ibContract1`. `ibReqID1` returns the request identifier for this contract details request.

Create the IB Trader Workstation `IContract` object `ibContract2`. Here, this object describes the second call option in the calendar spread. Create an `IContract` object with these property values:

- Google symbol.
- Stock option.
- Expiry date is September 2014.
- Strike price is $535.00.
- Call option.
- Number of shares is 100.
- Aggregate exchange.
- Primary exchange
- USD currency.

Here, `EX` is a sample primary exchange name. Substitute your primary exchange name in `ibContract2.primaryExchange`.

```
ibContract2 = ib.Handle.createContract;
ibContract2.symbol = 'GOOG';
ibContract2.secType = 'OPT';
ibContract2.expiry = '201409';
ibContract2.strike = 535;
ibContract2.right = 'C';
ibContract2.multiplier = '100';
ibContract2.exchange = 'SMART';
ibContract2.primaryExchange = 'EX';
ibContract2.currency = 'USD';
```

Request contract details for `ibContract2`.

```
[cd2,ibReqID2] = contractdetails(ib,ibContract2);
```

`cd2` returns the contract details data for `ibContract2`. `ibReqID2` returns the request identifier for this contract details request.

Create IB Trader Workstation `IComboLegList` Object

To define the legs of the combination order, create the IB Trader Workstation `IComboLegList` object `comboLegs`.

```
comboLegs = ib.Handle.createComboLegList;
```

Here, this combination order has two legs. Add the first leg to `comboLegs`. The first leg contains these property values:

- IB Trader Workstation contract identifier for the first contract.
- One-to-one leg ratio.
- Sell the call option.
- Aggregate exchange.
- Identify an open or close order based on the parent security.
- IB Trader Workstation routes the order without a designated broker.
- Blank designated broker.

```
ibLeg1 = comboLegs.Add;
ibLeg1.conId = cd1.summary.conId;
ibLeg1.ratio = 1;
ibLeg1.action = 'SELL';
ibLeg1.exchange = 'SMART';
ibLeg1.openClose = 0;
```

```
ibLeg1.shortSaleSlot = 0;
ibLeg1.designatedLocation = '';
```

Add the second leg to `comboLegs`. The second leg contains these property values:

*   IB Trader Workstation contract identifier for the second contract.
*   One-to-one leg ratio.
*   Buy the call option.
*   Aggregate exchange.
*   Identify an open or close order based on the parent security.
*   IB Trader Workstation routes the order without a designated broker.
*   Blank designated broker.

```
ibLeg2 = comboLegs.Add;
ibLeg2.conId = cd2.summary.conId;
ibLeg2.ratio = 1;
ibLeg2.action = 'BUY';
ibLeg2.exchange = 'SMART';
ibLeg2.openClose = 0;
ibLeg2.shortSaleSlot = 0;
ibLeg2.designatedLocation = '';
```

Create the Interactive Brokers Combination Order

Create the IB Trader Workstation `IContract` object `orderContract` for the combination order. Create an `IContract` object with these property values:

*   Google symbol
*   Combination order type `BAG`
*   Aggregate exchange
*   Primary exchange
*   USD currency
*   IB Trader Workstation `IComboLegList` object `comboLegs`

Here, `EX` is a sample primary exchange name. Substitute your primary exchange name in `orderContract.primaryExchange`.

```
orderContract = ib.Handle.createContract;
orderContract.symbol = 'GOOG';
```

```
orderContract.secType = 'BAG';
orderContract.exchange = 'SMART';
orderContract.primaryExchange = 'EX';
orderContract.currency = 'USD';
orderContract.comboLegs = comboLegs;
```

Create the IB Trader Workstation `IOrder` object `ibOrder`. Here, the combination order is a market order to buy one combination of the two legs.

```
ibOrder = ib.Handle.createOrder;
ibOrder.action = 'BUY';
ibOrder.totalQuantity = 1;
ibOrder.orderType = 'MKT';
```

Request the next valid order identification number `id` using `orderid`.

```
id = orderid(ib);
```

Execute the combination order `ibOrder` using these arguments:

- IB Trader Workstation connection `ib`
- Combination order `IContract` object `orderContract`
- IB Trader Workstation `IOrder` object `ibOrder`
- Order identifier `id`
- Sample event handler `ibExampleEventHandler`

```
d = createOrder(ib,orderContract,ibOrder,id,@ibExampleEventHandler)

d =

    768413.00
```

`d` returns the unique order identifier for this combination order.

Close the Connection

Close the IB Trader Workstation connection `ib`.

```
close(ib)
```

See Also
close | contractdetails | createOrder | ibtws | orderid

Related Examples

More About

External Websites

- Interactive Brokers API Reference Guide

Create CQG Orders

This example shows how to connect to CQG, define the event handlers, subscribe to the security, define the account handle, and submit orders for execution.

Create the CQG Connection

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Define Event Handlers

Register the sample event handler `cqgconnectioneventhandler` to track events associated with the connection status.

```
eventNames = {'CELStarted','DataError','IsReady', ...
    'DataConnectionStatusChanged', ...
    'GWConnectionStatusChanged', ...
    'GWEnvironmentChanged'};

for i = 1:length(eventNames)
  registerevent(c.Handle,{eventNames{i}, ...
    @(varargin)cqgconnectioneventhandler(varargin{:})})
end
```

`cqgconnectioneventhandler` is assigned to the events in `eventNames`.

Set the API configuration properties. For example, to set the time zone to Eastern Time, enter the following.

```
c.APIConfig.TimeZoneCode = 'tzEastern';
```

`c.APIConfig` is a CQG configuration object. For details about setting API configuration properties, see *CQG API Reference Guide*.

Establish the connection to CQG.

```
startUp(c)

CELStarted
DataConnectionStatusChanged
GWConnectionStatusChanged
```

The connection event handler displays event names for a successful CQG connection.

Register an event handler to track events associated with a CQG instrument subscription.

```matlab
streamEventNames = {'InstrumentSubscribed','InstrumentChanged', ...
    'IncorrectSymbol'};

for i = 1:length(streamEventNames)
  registerevent(c.Handle,{streamEventNames{i}, ...
    @(varargin)cqgrealtimeeventhandler(varargin{:})})
end
```

Register an event handler to track events associated with a CQG order and account.

```matlab
orderEventNames = {'AccountChanged','OrderChanged','AllOrdersCanceled'};

for i = 1:length(orderEventNames)
  registerevent(c.Handle,{orderEventNames{i}, ...
    @(varargin)cqgordereventhandler(varargin{:})})
end
```

Subscribe to the CQG Instrument

With the connection established, subscribe to the CQG instrument. The instrument must be successfully subscribed first before it is available for transactions. You must format the instrument name in the CQG long symbol view. For example, to subscribe to a security tied to the EURIBOR, enter the following.

```matlab
realtime(c,'F.US.IE')
pause(2)
```

```
F.US.IEK13 subscribed
```

`pause` causes MATLAB to wait 2 seconds before continuing to give time for CQG to subscribe to the instrument.

Create the CQG instrument object.

To use the instrument in `createOrder`, import the name of the instrument `cqgInstrumentName` into the current MATLAB workspace. Then, create the `CQGInstrument` object `cqgInst`.

```matlab
cqgInstrumentName = evalin('base','cqgInstrument');
cqgInst = c.Handle.Instruments.Item(cqgInstrumentName);
```

Set Up Account Credentials

Set the CQG flags to enable account information retrieval.

```
set(c.Handle,'AccountSubscriptionLevel','aslNone')
set(c.Handle,'AccountSubscriptionLevel','aslAccountUpdatesAndOrders')
pause(2)

ans =
    AccountChanged
```

The CQG API shows that account information changed.

Set up the CQG account credentials.

Retrieve the `CQGAccount` object into `accountHandle` to use your account information in `createOrder`. For details about creating a `CQGAccount` object, see *CQG API Reference Guide*.

```
accountHandle = c.Handle.Accounts.ItemByIndex(0);
```

Create CQG Market, Limit, Stop, and Stop Limit Orders

Create a market order that buys one share of the subscribed security `cqgInst` using the account credentials `accountHandle`.

```
quantity = 1;

oMarket = createOrder(c,cqgInst,1,accountHandle,quantity);
oMarket.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oMarket` contains the order. The CQG API executes the market order using the CQG API function `Place`. After execution, the order status changes.

To use a character vector for the security, subscribe to the security `'EZC'` as shown above. Then, create a market order that buys one share of the security `'EZC'` using the defined account credentials `accountHandle`.

```
cqgInstrumentName = 'EZC';
quantity = 1;
```

```matlab
oMarket = createOrder(c,cqgInstrumentName,1,accountHandle,quantity);
oMarket.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oMarket` contains the order. The CQG API executes the market order using the CQG API function `Place`. After execution, the order status changes.

To create a limit order, you can use the bid price. Extract the CQG bid object `qtBid` from the previously defined `CQGInstrument` object `cqgInst`. For details about the `CQGInstrument` object, see *CQG API Reference Guide*.

```matlab
qtBid = cqgInst.get('Bid');
```

Create a limit order that buys one share of the previously subscribed security `cqgInst` using the previously defined account credentials `accountHandle` and `qtBid` for the limit price.

```matlab
quantity = 1;
limitprice = qtBid.get('Price');

oLimit = createOrder(c,cqgInst,2,accountHandle,quantity,limitprice);
oLimit.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oLimit` contains the order. The CQG API executes the limit order using the CQG API function `Place`. After execution, the order status changes.

To create a stop order, you can use the trade price. Extract the CQG trade object `qtTrade` from the previously defined `CQGInstrument` object `cqgInst`.

```matlab
qtTrade = cqgInst.get('Trade');
```

Create a stop order that buys one share of the previously subscribed security `cqgInst` using the previously defined account credentials `accountHandle` and `qtTrade` for the stop price.

```matlab
quantity = 1;
stopprice = qtTrade.get('Price');

oStop = createOrder(c,cqgInst,3,accountHandle,quantity,stopprice);
```

```
oStop.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oStop` contains the order. The CQG API executes the stop order using the CQG API function `Place`. After execution, the order status changes.

To create a stop limit order, use both the bid and trade prices defined above. Create a stop limit order that buys one share of the subscribed security `cqgInst` using the defined account credentials `accountHandle`.

```
quantity = 1;

oStopLimit = createOrder(c,cqgInst,4,accountHandle,quantity, ...
    limitprice,stopprice);
oStopLimit.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oStopLimit` contains the order. The CQG API executes the stop limit order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG Connection

```
shutDown(c)
```

See Also

`close` | `cqg` | `createOrder` | `history` | `realtime` | `shutDown` | `startUp` | `timeseries`

Related Examples

- "Create an Order Using CQG" on page 1-12
- "Request CQG Historical Data" on page 4-51
- "Request CQG Real-Time Data" on page 4-58
- "Request CQG Intraday Tick Data" on page 4-54

More About

- "Workflow for CQG" on page 2-8

External Websites

- CQG API Reference Guide

Request CQG Historical Data

This example shows how to connect to CQG, define event handlers, and request historical data.

Connect to CQG

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Define Event Handlers

Register the sample event handler `cqgconnectioneventhandler` to track events associated with connection status.

```
eventNames = {'CELStarted','DataError','IsReady', ...
    'DataConnectionStatusChanged'};

for i = 1:length(eventNames)
  registerevent(c.Handle,{eventNames{i}, ...
    @(varargin)cqgconnectioneventhandler(varargin{:})})
end
```

`cqgconnectioneventhandler` is assigned to the events in `eventNames`.

Set the API configuration properties. For example, to set the time zone to Eastern Time, enter the following.

```
c.APIConfig.TimeZoneCode = 'tzEastern';
```

`c.APIConfig` is a CQG configuration object. For details about setting API configuration properties, see *CQG API Reference Guide*.

Create the CQG connection.

```
startUp(c)
```

```
CELStarted
DataConnectionStatusChanged
```

The connection event handler displays event names for a successful CQG connection.

Register an event handler to build and initialize the output data matrix `cqgHistoryData`.

```matlab
histEventNames = {'ExpressionResolved','ExpressionAdded', ...
    'ExpressionUpdated'};

for i = 1:length(histEventNames)
  registerevent(c.Handle,{histEventNames{i}, ...
    @(varargin)cqgexpressioneventhandler(varargin{:})})
end
```

Pass an Additional Optional Request Property

Pass an additional optional request property by creating the structure x and setting the optional property.

```matlab
x.UpdatesEnabled = false;
```

For additional optional properties you can set, see *CQG API Reference Guide*.

Request CQG Historical Data

Request daily data for instrument XYZ.XYZ for the last 10 days using the additional optional request property x. XYZ.XYZ is a sample instrument name. To request historical data for your instrument, substitute the symbol name in instrument.

```matlab
instrument = {'Close(XYZ.XYZ)','Open(XYZ.XYZ)'};
startdate = floor(now) - 10;
enddate = floor(now);
period = 'hpDaily';

history(c,instrument,startdate,enddate,period,x)
pause(1)
```

MATLAB writes the variable cqgHistoryData to the Workspace browser.

Display cqgHistoryData.

```matlab
cqgHistoryData
```

```matlab
cqgHistoryData =
   1.0e+05 *
    7.3533    0.0063    0.0063
    7.3533    0.0064    0.0064
    7.3533    0.0065    0.0065
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0065    0.0065
```

```
7.3534    0.0066    0.0066
7.3534    0.0066    0.0066
7.3534    0.0064    0.0064
```

Each row in `cqgHistoryData` represents data for 1 day. The columns in `cqgHistoryData` show the numerical representation of the timestamp, the close price, and the open price for the instrument during the day.

Close the CQG Connection

```
close(c)
```

See Also

`close` | `cqg` | `createOrder` | `history` | `realtime` | `shutDown` | `startUp` | `timeseries`

Related Examples

- "Create an Order Using CQG" on page 1-12
- "Create CQG Orders" on page 4-45
- "Request CQG Real-Time Data" on page 4-58
- "Request CQG Intraday Tick Data" on page 4-54

More About

- "Workflow for CQG" on page 2-8

External Websites

- CQG API Reference Guide

Request CQG Intraday Tick Data

This example shows how to connect to CQG, define event handlers, and request intraday and timed bar data.

Connect to CQG and Define Event Handlers

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Register the sample event handler `cqgconnectioneventhandler` to track events associated with the connection status.

```
eventNames = {'CELStarted','DataError','IsReady', ...
    'DataConnectionStatusChanged'};

for i = 1:length(eventNames)
  registerevent(c.Handle,{eventNames{i}, ...
    @(varargin)cqgconnectioneventhandler(varargin{:})})
end
```

`cqgconnectioneventhandler` is assigned to the events in `eventNames`.

Set the API configuration properties. For example, to set the time zone to Eastern Time, enter the following.

```
c.APIConfig.TimeZoneCode = 'tzEastern';
```

`c.APIConfig` is a CQG configuration object. For details about setting API configuration properties, see *CQG API Reference Guide*.

Create the CQG connection.

```
startUp(c)
```

```
CELStarted
DataConnectionStatusChanged
```

The connection event handler displays event names for a successful CQG connection.

Register an event handler to build and initialize the output data structure `cqgTickData` used for storing intraday tick data.

```
rawEventNames = {'TicksResolved','TicksAdded'};
```

```
for i = 1:length(rawEventNames)
  registerevent(c.Handle,{rawEventNames{i}, ...
    @(varargin)cqgintradayeventhandler(varargin{:})})
end
```

Request CQG Intraday Tick Data

Pass an additional optional request property by creating the structure x, and setting the optional property. To see only bid tick data, for example, set TickFilter to 'tfBid'.

```
x.TickFilter = 'tfBid';
```

TickFilter and SessionsFilter are the only valid additional optional properties for calling timeseries without a timed bar request. For additional property values you can set, see *CQG API Reference Guide*.

Request intraday tick data for instrument XYZ.XYZ for the last 2 days using the additional optional request property x. XYZ.XYZ is a sample instrument name. To request intraday tick data for your instrument, substitute the symbol name in instrument.

```
instrument = 'XYZ.XYZ';
startdate = now - 2;
enddate = now;

timeseries(c,instrument,startdate,enddate,[],x)
pause(1)
```

pause causes MATLAB to wait 1 second before continuing to give time for CQG to subscribe to the instrument. MATLAB writes the variable cqgTickData to the Workspace browser.

Display cqgTickData.

```
cqgTickData

cqgTickData =
             Timestamp: {2x1 cell}
                 Price: [2x1 double]
                Volume: [2x1 double]
             PriceType: {2x1 cell}
        CorrectionType: {2x1 cell}
    SalesConditionLabel: {2x1 cell}
```

```
     SalesConditionCode: [2x1 double]
           ContributorId: {2x1 cell}
       ContributorIdCode: [2x1 double]
             MarketState: {2x1 cell}
```

Display data in the `Timestamp` property of `cqgTickData`.

```
cqgTickData.Timestamp

ans =
    '4/17/2013 2:14:00 PM'
    '4/18/2013 2:14:00 PM'
```

Request CQG Timed Bar Data

Register an event handler to build and initialize the output data matrix
`cqgTimedBarData` used for storing timed bar data.

```
aggEventNames = {'TimedBarsResolved','TimedBarsAdded', ...
    'TimedBarsUpdated','TimedBarsInserted', ...
    'TimedBarsRemoved'};

for i = 1:length(aggEventNames)
  registerevent(c.Handle,{aggEventNames{i}, ...
    @(varargin)cqgintradayeventhandler(varargin{:})})
end
```

Pass additional optional request properties by creating the structure `x`, and setting the
optional property.

```
x.UpdatesEnabled = false;
```

Request timed bar data for instrument `XYZ.XYZ` for the last fraction of a day using
the additional optional request property `x`. `XYZ.XYZ` is a sample instrument name. To
request timed bar data for your instrument, substitute the symbol name in `instrument`.

```
instrument = 'XYZ.XYZ';
startdate = now - .1;
enddate = now;
intraday = 1;

timeseries(c,instrument,startdate,enddate,intraday,x)
pause(1)
```

MATLAB writes the variable `cqgTimedBarData` to the Workspace browser.

Display `cqgTimedBarData`.

`cqgTimedBarData`

```
cqgTimedBarData =
   1.0e+09 *
     0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
     0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
     0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
     0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
     0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
     ...
```

`cqgTimedBarData` returns timed bar data for the specified instrument. The columns of `cqgTimedBarData` display data corresponding to the timestamp, open price, high price, low price, close price, mid-price, HLC3, average price, and tick volume.

Close the CQG Connection

`close(c)`

See Also

`close` | `cqg` | `createOrder` | `history` | `realtime` | `shutDown` | `startUp` | `timeseries`

Related Examples

- "Create an Order Using CQG" on page 1-12
- "Create CQG Orders" on page 4-45
- "Request CQG Historical Data" on page 4-51
- "Request CQG Real-Time Data" on page 4-58

More About

- "Workflow for CQG" on page 2-8

External Websites

- CQG API Reference Guide

Request CQG Real-Time Data

This example shows how to connect to CQG, define event handlers, and request current data.

Connect to CQG

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Define Event Handlers

Register the sample event handler `cqgconnectioneventhandler` to track events for the connection status.

```
eventNames = {'CELStarted','DataError','IsReady', ...
    'DataConnectionStatusChanged','GWConnectionStatusChanged', ...
    'GWEnvironmentChanged'};

for i = 1:length(eventNames)
  registerevent(c.Handle,{eventNames{i}, ...
    @(varargin)cqgconnectioneventhandler(varargin{:})})
end
```

`cqgconnectioneventhandler` is assigned to the events in `eventNames`.

Set the API configuration properties. For example, to set the time zone to Eastern Time, enter the following.

```
c.APIConfig.TimeZoneCode = 'tzEastern';
```

`c.APIConfig` is a CQG configuration object. For details about setting the API configuration properties, see *CQG API Reference Guide*.

Establish the connection to CQG.

```
startUp(c)

CELStarted
DataConnectionStatusChanged
GWConnectionStatusChanged
```

The connection event handler displays event names for a successful CQG connection.

Register an event handler to track events associated with the CQG instrument subscription.

```
streamEventNames = {'InstrumentSubscribed','InstrumentChanged', ...
    'IncorrectSymbol'};

for i = 1:length(streamEventNames)
  registerevent(c.Handle,{streamEventNames{i}, ...
    @(varargin)cqgrealtimeeventhandler(varargin{:})})
end
```

Request CQG Real-Time Data

With the connection established, subscribe to the instrument. The instrument name must be formatted in the CQG long symbol view. For example, to subscribe to a security tied to corn, enter the following.

```
instrument = 'F.US.EZC';
realtime(c,instrument)
```

MATLAB writes the structure variable `cqgDataEZC` to the Workspace browser.

Display `cqgDataEZC`.

```
cqgDataEZC(1,1)

ans =
               Price: {15x1 cell}
              Volume: {15x1 cell}
     ServerTimestamp: {15x1 cell}
           Timestamp: {15x1 cell}
                Type: {15x1 cell}
                Name: {15x1 cell}
             IsValid: {15x1 cell}
          Instrument: {15x1 cell}
           HasVolume: {15x1 cell}
```

`cqgDataEZC` returns the current quotes for the security.

Display data in the `Price` property of `cqgDataEZC`.

```
cqgDataEZC(1,1).Price

ans =
    [-2.1475e+09]
```

```
[-2.1475e+09]
[-2.1475e+09]
[    660.5000]
[]
[]
[-2.1475e+09]
[-2.1475e+09]
[-2.1475e+09]
[-2.1475e+09]
[-2.1475e+09]
[-2.1475e+09]
[-2.1475e+09]
[    660.5000]
[-2.1475e+09]
```

Close the CQG Connection

```
close(c)
```

See Also
```
close | cqg | createOrder | history | realtime | shutDown | startUp |
timeseries
```

Related Examples

More About

External Websites
- CQG API Reference Guide

Functions — Alphabetical List

emsx

Create Bloomberg EMSX connection

Syntax

```
c = emsx(servicename)
```

Description

`c = emsx(servicename)` creates a connection to the local Bloomberg EMSX communications server using the service `servicename`.

Examples

Connect to the Bloomberg EMSX Test Service

Create a connection `c` to the Bloomberg EMSX test service. You can place test calls using this service.

```
 c = emsx('//blp/emapisvc_beta')

c =

  emsx with properties:

     Session: [1x1 com.bloomberglp.blpapi.Session]
     Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
   Ipaddress: 'localhost'
        Port: 8194
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

Close the Bloomberg EMSX connection.

```
close(c)
```

Connect to the Bloomberg EMSX Production Service

Create a connection `c` to the Bloomberg EMSX production service. You can place live calls using this service.

```
c = emsx('//bmp/emapisvc')

c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX production service
- Port number of the machine running the Bloomberg EMSX production service

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`servicename` — Bloomberg EMSX service name
`'//blp/emapisvc_beta'` | `'//bmp/emapisvc'`

Bloomberg EMSX service name, specified as one of these connection types.

Connection Type	Bloomberg EMSX Service Name
Test	`'//blp/emapisvc_beta'`
Production	`'//bmp/emapisvc'`

Output Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, returned as a connection object with these properties.

Property	Description
`Session`	Bloomberg EMSX session object
`Service`	Bloomberg EMSX service object
`Ipaddress`	IP address of the machine where Bloomberg EMSX is running
`Port`	Port number of the machine where Bloomberg EMSX is running

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
close | createOrder | createOrderAndRoute | routeOrder

Topics

Introduced in R2013a

close

Close Bloomberg EMSX connection

Syntax

```
close(c)
```

Description

`close(c)` closes the Bloomberg EMSX connection `c`.

Examples

Close the Bloomberg EMSX Connection

Create the Bloomberg EMSX connection `c`.

```
c = emsx('//blp/emapisvc_beta');
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
createOrder | createOrderAndRoute | emsx | routeOrder

Topics
"Create an Order Using Bloomberg EMSX" on page 1-14
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21
"Workflow for Bloomberg EMSX" on page 2-2

Introduced in R2013a

createOrder

Create Bloomberg EMSX order

Syntax

```
events = createOrder(c,order)
events = createOrder(c,order,'timeOut',timeout)

createOrder( ___ ,'useDefaultEventHandler',false)

___ = createOrder(,c,order,options)
```

Description

`events = createOrder(c,order)` creates a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and order request `order` that contains the required fields for creating an order. `createOrder` returns the order sequence number and status message using the default event handler.

`events = createOrder(c,order,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrder( ___ ,'useDefaultEventHandler',false)` creates a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = createOrder(,c,order,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Create an Order Using the Default Event Handler

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)

events =

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create an Order Using a Timeout

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`. Set the timeout value to 200 milliseconds.

```
events = createOrder(c,order,'timeOut',200)

events =

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create an Order Using a Custom Event Handler

To create a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using timer. Start the timer to run `eventhandler` immediately using `start`. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see timer.

Create the order using the Bloomberg EMSX connection `c` and `order`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating an order.

```
createOrder(c,order,'useDefaultEventHandler',false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create an Order Using an Options Structure

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Create the order using the Bloomberg EMSX connection `c`, `order`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = createOrder(c,order,options)

events =

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`order` — Order request
structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX amount of shares
EMSX_ORDER_TYPE	Bloomberg EMSX order type
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_TIF	Bloomberg EMSX time in force
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction

Field	Description
EMSX_SIDE	Bloomberg EMSX buy or sell specification

Example: `order.EMSX_TICKER = 'XYZ';`
`order.EMSX_AMOUNT = int32(100);`
`order.EMSX_ORDER_TYPE = 'MKT';`
`order.EMSX_BROKER = 'BB';`
`order.EMSX_TIF = 'DAY';`
`order.EMSX_HAND_INSTRUCTION = 'ANY';`
`order.EMSX_SIDE = 'BUY';`

Data Types: `struct`

`timeout` — Timeout value
500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
timer | close | createOrder | createOrderAndRoute | createOrderAndRouteWithStrat | delete | deleteOrder | deleteRoute | emsx | modifyOrder | orders | routeOrder | routes | start | stop

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21
"Workflow for Bloomberg EMSX" on page 2-2
"Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25

Introduced in R2013a

createOrderAndRoute

Create and route Bloomberg EMSX order

Syntax

```
events = createOrderAndRoute(c,order)
events = createOrderAndRoute(c,order,'timeOut',timeout)

createOrderAndRoute( ___ ,'useDefaultEventHandler',false)

___ = createOrderAndRoute(c,order,options)
```

Description

`events = createOrderAndRoute(c,order)` creates and routes a Bloomberg EMSX order using Bloomberg EMSX connection `c` and order request `order`. `createOrderAndRoute` returns the order sequence number, route number, and status message using the default event handler.

`events = createOrderAndRoute(c,order,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrderAndRoute( ___ ,'useDefaultEventHandler',false)` creates and routes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating and routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = createOrderAndRoute(c,order,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Create and Route an Order Using the Default Event Handler

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrderAndRoute(c,order)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Timeout

To create and route a Bloomberg EMSX order, create the connection c using emsx and set up the order and route subscription using orders and routes. For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Create the order request structure order to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker BB with the time in force set to DAY and any hand instruction. Convert the number of shares to a 32-bit signed integer using int32.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection c and order. Set the timeout value to 200 milliseconds.

```
events = createOrderAndRoute(c,order,'timeOut',200)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. createOrderAndRoute returns events as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier

- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects osubs and rsubs. This code assumes that orders creates osubs and routes creates rsubs.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Custom Event Handler

To create and route a Bloomberg EMSX order, create the Bloomberg EMSX connection c using emsx and set up the order and route subscription using orders and routes. For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Create the order request structure order to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker BB with the time in force set to DAY and any hand instruction. Convert the number of shares to a 32-bit signed integer using int32.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Suppose you create a custom event handler function called eventhandler with input argument c. Run eventhandler using timer. Start the timer to run eventhandler immediately using start. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

t is the MATLAB timer object. For details, see timer.

Create and route the order using the Bloomberg EMSX connection `c` and `order`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating and routing an order.

```
createOrderAndRoute(c,order,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using an Options Structure

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Create

and route the order using the Bloomberg EMSX connection `c`, `order`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = createOrderAndRoute(c,order,options)

events =

    EMSX_SEQUENCE: 728924
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- "Create an Order Using Bloomberg EMSX" on page 1-14
- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`order` — Order request
structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX amount of shares
EMSX_ORDER_TYPE	Bloomberg EMSX order type
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_TIF	Bloomberg EMSX time in force
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction
EMSX_SIDE	Bloomberg EMSX buy or sell specification

Example: `order.EMSX_TICKER = 'XYZ';`
`order.EMSX_AMOUNT = int32(100);`
`order.EMSX_ORDER_TYPE = 'MKT';`
`order.EMSX_BROKER = 'BB';`
`order.EMSX_TIF = 'DAY';`
`order.EMSX_HAND_INSTRUCTION = 'ANY';`
`order.EMSX_SIDE = 'BUY';`

Data Types: `struct`

`timeout` — Timeout value
500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
timer | `close` | `createOrder` | `createOrderAndRouteWithStrat` | `delete` | `deleteOrder` | `deleteRoute` | `emsx` | `modifyOrder` | `orders` | `routeOrder` | `routes` | `start` | `stop`

Topics
"Create an Order Using Bloomberg EMSX" on page 1-14
"Create and Manage a Bloomberg EMSX Order" on page 4-12

Introduced in R2013a

createOrderAndRouteWithStrat

Create and route Bloomberg EMSX order with strategies

Syntax

```
events = createOrderAndRouteWithStrat(c,order,strat)
events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',
timeout)

createOrderAndRouteWithStrat( ___ ,'useDefaultEventHandler',false)

 ___ = createOrderAndRouteWithStrat(c,order,strat,options)
```

Description

`events = createOrderAndRouteWithStrat(c,order,strat)` creates and routes a Bloomberg EMSX order with strategies using Bloomberg EMSX connection `c`, order request `order`, and order strategy `strat`. `createOrderAndRouteWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrderAndRouteWithStrat( ___ ,'useDefaultEventHandler',false)` creates and routes a Bloomberg EMSX order with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating and routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

` ___ = createOrderAndRouteWithStrat(c,order,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure

and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Create and Route an Order Using the Default Event Handler

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`.

```
events = createOrderAndRouteWithStrat(c,order,strat)

events =
```

```
    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Timeout

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',200)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Custom Event Handler

To create and route a Bloomberg EMSX order with strategies, create the Bloomberg EMSX connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker BB with the time in force set to DAY and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy SSP. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using timer. Start the timer to run `eventhandler` immediately using `start`. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see timer.

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating and routing an order.

```
createOrderAndRouteWithStrat(c,order,strat,...
                             'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

```
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using an Options Structure

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Create and route the order using the Bloomberg EMSX connection `c`, `order`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
```

```
events = createOrderAndRouteWithStrat(c,order,strat,options)

events =

    EMSX_SEQUENCE: 728924
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

order — Order request
structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX amount of shares
EMSX_ORDER_TYPE	Bloomberg EMSX order type
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_TIF	Bloomberg EMSX time in force
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction
EMSX_SIDE	Bloomberg EMSX buy or sell specification

Example: `order.EMSX_TICKER = 'XYZ';`
`order.EMSX_AMOUNT = int32(100);`
`order.EMSX_ORDER_TYPE = 'MKT';`
`order.EMSX_BROKER = 'BB';`
`order.EMSX_TIF = 'DAY';`
`order.EMSX_HAND_INSTRUCTION = 'ANY';`
`order.EMSX_SIDE = 'BUY';`

Data Types: `struct`

`strat` — Order strategies
structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `EMSX_FIELD_DATA`.

Example: `strat.EMSX_STRATEGY_NAME = 'SSP';`
`strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);`

```
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Data Types: `struct`

`timeout` — Timeout value
500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
timer | close | createOrder | delete | deleteOrder | deleteRoute | emsx | getBrokerInfo | modifyOrder | orders | routeOrder | routes | start | stop

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21
"Workflow for Bloomberg EMSX" on page 2-2
"Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25

Introduced in R2013a

deleteOrder

Delete Bloomberg EMSX order

Syntax

```
events = deleteOrder(c,ordernum)
events = deleteOrder(c,ordernum,'timeOut',timeout)

deleteOrder( ___ ,'useDefaultEventHandler',false)

___ = deleteOrder(c,ordernum,options)
```

Description

`events = deleteOrder(c,ordernum)` deletes a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and order number or structure `ordernum`. `deleteOrder` returns a status message using the default event handler.

`events = deleteOrder(c,ordernum,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`deleteOrder( ___ ,'useDefaultEventHandler',false)` deletes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with deleting orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = deleteOrder(c,ordernum,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Delete an Order Using the Default Event Handler

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`.

```
events = deleteOrder(c,ordernum)

events =

     STATUS: '0'
    MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using the Order Number Integer

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Delete the order using the Bloomberg EMSX connection `c` and the order sequence number 335877 for the order to delete.

```
events = deleteOrder(c,335877)

events =

     STATUS: '0'
    MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using a Timeout

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`. Set the timeout value to 200 milliseconds.

```
events = deleteOrder(c,ordernum,'timeOut',200)

events =
```

```
   STATUS: 'O'
  MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using a Custom Event Handler

To delete a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using timer. Start the timer to run `eventhandler` immediately using `start`. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see timer.

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with deleting an order.

```
deleteOrder(c,ordernum,'useDefaultEventHandler',false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using an Options Structure

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Delete the order using the Bloomberg EMSX connection `c`, `ordernum`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = deleteOrder(c,ordernum,options)

events =

    STATUS: 'O'
   MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`ordernum` — Order numbers to delete
structure | integer

Order numbers to delete, specified as a structure or an integer to denote one or more order sequence numbers.

Data Types: `struct` | `int32`

`timeout` — Timeout value
500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
`timer` | `close` | `createOrder` | `createOrderAndRoute` | `delete` | `deleteRoute` | `emsx` | `modifyOrder` | `orders` | `routeOrder` | `routes` | `start` | `stop`

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21

Introduced in R2013a

deleteRoute

Delete Bloomberg EMSX active shares

Syntax

```
events = deleteRoute(c,routenum)
events = deleteRoute(c,routenum,'timeOut',timeout)

deleteRoute( ___ ,'useDefaultEventHandler',false)

___ = deleteRoute(c,routenum,options)
```

Description

`events = deleteRoute(c,routenum)` deletes the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and route number `routenum`. `deleteRoute` returns a status message using the default event handler.

`events = deleteRoute(c,routenum,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`deleteRoute( ___ ,'useDefaultEventHandler',false)` deletes the active shares that are routed but not filled using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with deleting the active shares. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = deleteRoute(c,routenum,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Delete Active Shares

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1 Create the connection `c` using `emsx`.
2 Set up an order and route subscription using `orders` and `routes`.
3 Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see "Create and Manage a Bloomberg EMSX Route" on page 4-16.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`.

```
events = deleteRoute(c,routenum)

events =

    STATUS: '1'
   MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting the active shares. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete Active Shares Using a Timeout

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1 Create the connection `c` using `emsx`.

2 Set up an order and route subscription using `orders` and `routes`.

3 Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see "Create and Manage a Bloomberg EMSX Route" on page 4-16.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`. Set the timeout value to 200 milliseconds.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = deleteRoute(c,routenum,'timeOut',200)

events =

     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting the active shares. `deleteRoute` returns `events` as a structure that contains these fields:

• Bloomberg EMSX status
• Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete Active Shares Using a Custom Event Handler

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1 Create the Bloomberg EMSX connection `c` using `emsx`.
2 Set up an order and route subscription using `orders` and `routes`.
3 Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see "Create and Manage a Bloomberg EMSX Route" on page 4-16.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using timer. Start the timer to run `eventhandler` immediately using `start`. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see timer.

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with deleting the active shares.

```
deleteRoute(c,routenum,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

```
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete Active Shares Using an Options Structure

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1 Create the connection `c` using `emsx`.

2 Set up an order and route subscription using `orders` and `routes`.

3 Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see "Create and Manage a Bloomberg EMSX Route" on page 4-16.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c`, `routenum`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = deleteRoute(c,routenum,options)

events =

    STATUS: '1'
   MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting the active shares. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

`c` — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`routenum` — Route to delete
structure

Route to delete, specified as a structure containing fields `EMSX_SEQUENCE` and `EMSX_ROUTE_ID`.

Example: `routenum.EMSX_SEQUENCE = 728918;`
`routenum.EMSX_ROUTE_ID = 1;`

Data Types: `struct`

`timeout` — Timeout value
500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: double

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: struct

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
timer | close | createOrder | createOrderAndRoute | delete | deleteOrder |
emsx | modifyOrder | modifyRoute | orders | routeOrder | routes | start |
stop

Topics

Introduced in R2013a

getAllFieldMetaData

Obtain Bloomberg EMSX field information

Syntax

```
r = getAllFieldMetaData(c)
```

Description

`r = getAllFieldMetaData(c)` returns the Bloomberg EMSX field information using the Bloomberg EMSX connection `c`.

Examples

Request All Field Information

Create a connection `c` to the Bloomberg EMSX.

```
c = emsx('//blp/emapisvc_beta');
```

Request all fields supported by Bloomberg EMSX service using the Bloomberg EMSX connection `c`.

```
r = getAllFieldMetaData(c)

r =

      EMSX_FIELD_NAME: {113x1 cell}
       EMSX_DISP_NAME: {113x1 cell}
            EMSX_TYPE: {113x1 cell}
           EMSX_LEVEL: [113x1 double]
             EMSX_LEN: [113x1 double]
```

Display all field information for the first Bloomberg EMSX field using a cell array. Create a cell array from the fields in the returned data structure `r`.

```
{r.EMSX_FIELD_NAME{1} r.EMSX_DISP_NAME{1} r.EMSX_TYPE{1} r.EMSX_LEVEL(1) r.EMSX_LEN(1)}
```

```
'MSG_TYPE'    'Msg Type'    'String'    [0]    [1]
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

Output Arguments

r — Return information for all fields
structure

Return information for all fields, returned as a structure for all fields supported by Bloomberg EMSX.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
close | createOrder | createOrderAndRoute |
createOrderAndRouteWithStrat | emsx

Topics

Introduced in R2013a

getBrokerInfo

Obtain Bloomberg EMSX broker and strategy information

Syntax

```
r = getBrokerInfo(c,brokerstrat)
```

Description

`r = getBrokerInfo(c,brokerstrat)` obtains Bloomberg EMSX broker and strategy information using the Bloomberg EMSX connection `c` and broker and strategy request structure `brokerstrat`.

Examples

Obtain Broker Information

Create a connection `c` to the Bloomberg EMSX.

```
c = emsx('//blp/emapisvc_beta');
```

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';

r = getBrokerInfo(c,brokerstrat)

r =

      EMSX_BROKERS: {2x1 cell}
```

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

Close the Bloomberg EMSX connection.

```
close(c)
```

Obtain Strategy Information

Create a connection c to the Bloomberg EMSX.

```
c = emsx('//blp/emapisvc_beta');
```

Define the broker and strategy information structure brokerstrat. Obtain strategy information using the Bloomberg EMSX connection c and structure brokerstrat.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
brokerstrat.EMSX_BROKER = 'BMTB';

r = getBrokerInfo(c,brokerstrat)

r =

        EMSX_STRATEGIES: {16x1 cell}
```

The EMSX_STRATEGIES field lists the Bloomberg EMSX strategies.

Close the Bloomberg EMSX connection.

```
close(c)
```

Obtain Field Information

Create a connection c to the Bloomberg EMSX.

```
c = emsx('//blp/emapisvc_beta');
```

Define the broker and strategy information structure brokerstrat. Obtain field information using the Bloomberg EMSX connection c and structure brokerstrat.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
brokerstrat.EMSX_BROKER = 'BMTB';
brokerstrat.EMSX_STRATEGY = 'SSP';

r = getBrokerInfo(c,brokerstrat)

r =

        FieldName: {3x1 cell}
```

```
        Disable: {3x1 cell}
    StringValue: {3x1 cell}
```

The structure field `FieldName` lists the Bloomberg EMSX fields. The structure fields `Disable` and `StringValue` contain information about the Bloomberg EMSX fields.

Close the Bloomberg EMSX connection.

`close(c)`

- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

`c` — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`brokerstrat` — Broker and strategy request
structure

Broker and strategy request, specified as a structure that contains Bloomberg EMSX fields. Use `getAllFieldMetaData` to view all available fields for `brokerStrategyStruct`.

Example: `brokerstrat.EMSX_TICKER = 'ABCD US Equity';`

Data Types: `struct`

Output Arguments

`r` — Broker and strategy information
structure

Broker and strategy information, returned as a structure.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
close | createOrder | createOrderAndRoute | createOrderAndRouteWithStrat | deleteOrder | deleteRoute | emsx | modifyOrder | orders | routeOrder | routes

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21
"Workflow for Bloomberg EMSX" on page 2-2

Introduced in R2013a

modifyOrder

Modify Bloomberg EMSX order

Syntax

```
events = modifyOrder(c,modorder)
events = modifyOrder(c,modorder,'timeOut',timeout)

modifyOrder( ___ ,'useDefaultEventHandler',false)

___ = modifyOrder(c,modorder,options)
```

Description

`events = modifyOrder(c,modorder)` modifies a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and modify order request structure `modorder`. `modifyOrder` returns a status message using the default event handler.

`events = modifyOrder(c,modorder,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyOrder( ___ ,'useDefaultEventHandler',false)` modifies a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = modifyOrder(c,modorder,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Modify an Order Using the Default Event Handler

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Modify the order using the Bloomberg EMSX connection `c` and `modorder`.

```
events = modifyOrder(c,modorder)

events =

    EMSX_SEQUENCE: 728905
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying the order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify an Order Using a Timeout

To modify a Bloomberg EMSX order, create the connection c using emsx, set up the order subscription using orders, and create an order using createOrder. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Define the structure modorder that contains the order sequence number EMSX_SEQUENCE, the security EMSX_TICKER, and the number of shares EMSX_AMOUNT. This code modifies the order number 728905 for 200 shares of IBM. Convert the numbers to 32-bit signed integers using int32.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Modify the order using the Bloomberg EMSX connection c and modorder. Set the timeout value to 200 milliseconds.

```
events = modifyOrder(c,modorder,'timeOut',200)

events =

    EMSX_SEQUENCE: 728905
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying the order. modifyOrder returns events as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object subs. This code assumes that orders creates subs.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify an Order Using a Custom Event Handler

To modify a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number 728905 for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using timer. Start the timer to run `eventhandler` immediately using `start`. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see timer.

Modify the order using the Bloomberg EMSX connection `c` and `modorder`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying an order.

```
modifyOrder(c,modorder,'useDefaultEventHandler',false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify an Order Using an Options Structure

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Modify the order using the Bloomberg EMSX connection `c`, `modorder`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyOrder(c,modorder,options)

events =

    EMSX_SEQUENCE: 728905
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying the order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

modorder — Modify order request
structure

Modify order request, specified as a structure that contains these fields.

Use `getAllFieldMetaData` to view all available fields for `modorder`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares

```
Example: modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'XYZ';
modorder.EMSX_AMOUNT = int32(100);
```

Data Types: `struct`

`timeout` — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
timer | close | createOrder | createOrderAndRoute |
createOrderAndRouteWithStrat | delete | deleteOrder | deleteRoute | emsx
| orders | routeOrder | routes | start | stop

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21
"Workflow for Bloomberg EMSX" on page 2-2
"Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page
1-25

Introduced in R2013a

modifyRoute

Modify Bloomberg EMSX route

Syntax

```
events = modifyRoute(c,modroute)
events = modifyRoute(c,modroute,'timeOut',timeout)

modifyRoute( ___ ,'useDefaultEventHandler',false)

 ___  = modifyRoute(c,modroute,options)
```

Description

`events = modifyRoute(c,modroute)` modifies a Bloomberg EMSX route using the Bloomberg EMSX connection `c` and route request `modroute`. `modifyRoute` returns a status message using the default event handler.

`events = modifyRoute(c,modroute,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyRoute( ___ ,'useDefaultEventHandler',false)` modifies a Bloomberg EMSX route using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying routes. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = modifyRoute(c,modroute,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Modify a Route Using the Default Event Handler

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code instructs Bloomberg EMSX to route 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and `modroute`.

```
events = modifyRoute(c,modroute)

events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects osubs and rsubs. This code assumes that orders creates osubs and routes creates rsubs.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route Using a Timeout

To modify a route for a Bloomberg EMSX order:

- Create the connection c using emsx.
- Set up the order and route subscription using orders and routes.
- Create and route the order using createOrderAndRoute.

For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Define the modroute structure that contains these fields:

- Bloomberg EMSX order sequence number EMSX_SEQUENCE
- Bloomberg EMSX ticker symbol EMSX_TICKER
- Bloomberg EMSX number of shares EMSX_AMOUNT
- Bloomberg EMSX route identifier EMSX_ROUTE_ID

This code modifies the route to 100 shares of IBM for order sequence number 731128 and route identifier 1. Convert the numbers to 32-bit signed integers using int32.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and `modroute`. Set the timeout value to 200 milliseconds.

```
events = modifyRoute(c,modroute,'timeOut',200)

events =

    EMSX_SEQUENCE: O
    EMSX_ROUTE_ID: O
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route Using a Custom Event Handler

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number EMSX_SEQUENCE

- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier 1. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using timer. Start the timer to run `eventhandler` immediately using `start`. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see timer.

Modify the route using the Bloomberg EMSX connection `c` and `modroute`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying a route.

```
modifyRoute(c,modroute,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route Using an Options Structure

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyRoute(c,modroute,options)

events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

modroute — Modify route request
structure

Modify route request, specified as a structure with these fields.

Use `getAllFieldMetaData` to view all available fields for `modroute`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number

Field	Description
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares
EMSX_ROUTE_ID	Bloomberg EMSX route identifier

Example: `modroute.EMSX_SEQUENCE = int32(731128);`
`modroute.EMSX_TICKER = 'XYZ';`
`modroute.EMSX_AMOUNT = int32(100);`
`modroute.EMSX_ROUTE_ID = int32(1);`

Data Types: `struct`

`timeout` — Timeout value
500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
timer | `createOrder` | `createOrderAndRoute` | `delete` | `deleteOrder` | `modifyRouteWithStrat` | `orders` | `routes` | `start` | `stop`

Topics

Introduced in R2013a

modifyRouteWithStrat

Modify route with strategies for Bloomberg EMSX

Syntax

```
events = modifyRouteWithStrat(c,modroute,strat)
events = modifyRouteWithStrat(c,modroute,strat,'timeOut',timeout)

modifyRouteWithStrat( ___ ,'useDefaultEventHandler',false)

___ = modifyRouteWithStrat(c,modroute,strat,options)
```

Description

`events = modifyRouteWithStrat(c,modroute,strat)` modifies a Bloomberg EMSX route with strategies using the Bloomberg EMSX connection `c`, route request `modroute`, and order strategy `strat`. `modifyRouteWithStrat` returns the order sequence number, route identifier, and status message using the default event handler.

`events = modifyRouteWithStrat(c,modroute,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyRouteWithStrat( ___ ,'useDefaultEventHandler',false)` modifies a Bloomberg EMSX route with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying routes. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = modifyRouteWithStrat(c,modroute,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Modify a Route with Strategies Using the Default Event Handler

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`.

```
events = modifyRouteWithStrat(c,modroute,strat)

events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
```

```
        MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route with Strategies Using a Timeout

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
```

```
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure strat using the strategy SSP. Convert the field indicators to a 32-bit signed integer using int32.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Modify the route using the Bloomberg EMSX connection c, modroute, and strat. Set the timeout value to 200 milliseconds.

```
events = modifyRouteWithStrat(c,modroute,strat,'timeOut',200)

events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. modifyRouteWithStrat returns events as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects osubs and rsubs. This code assumes that orders creates osubs and routes creates rsubs.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route with Strategies Using a Custom Event Handler

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection c using emsx.

- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using timer. Start the timer to run `eventhandler` immediately using `start`. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see timer.

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying a route.

```
modifyRouteWithStrat(c,modroute,strat,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route with Strategies Using an Options Structure

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see "Manage a Bloomberg EMSX Order and Route" on page 4-21.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure strat using the strategy SSP. Convert the field indicators to a 32-bit signed integer using int32.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure options. To use the default event handler, set the field useDefaultEventHandler to true. Set the field timeOut to 200 milliseconds. Modify the route using the Bloomberg EMSX connection c, modroute, strat, and options structure options.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyRouteWithStrat(c,modroute,strat,options)

events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. modifyRouteWithStrat returns events as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects osubs and rsubs. This code assumes that orders creates osubs and routes creates rsubs.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16

- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

modroute — Modify route request
structure

Modify route request, specified as a structure with these fields.

Use `getAllFieldMetaData` to view all available fields for `modroute`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares
EMSX_ROUTE_ID	Bloomberg EMSX route identifier

Example: `modroute.EMSX_SEQUENCE = int32(731128);`
`modroute.EMSX_TICKER = 'XYZ';`
`modroute.EMSX_AMOUNT = int32(100);`
`modroute.EMSX_ROUTE_ID = int32(1);`

Data Types: `struct`

strat — Order strategies
structure

Order strategies, specified as a structure that contains the fields:
`EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and
`EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert EMSX_STRATEGY_FIELD_INDICATORS to a 32-bit signed integer using `int32`. Set EMSX_STRATEGY_FIELD_INDICATORS equal to 0 for each field to use the field data setting in EMSX_FIELD_DATA. Or, set EMSX_STRATEGY_FIELD_INDICATORS equal to 1 to ignore the data in EMSX_FIELD_DATA.

Example: `strat.EMSX_STRATEGY_NAME = 'SSP';`
`strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);`
`strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};`

Data Types: `struct`

`timeout` — Timeout value
500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also

timer | `createOrder` | `createOrderAndRouteWithStrat` | `delete` | `deleteOrder` | `getBrokerInfo` | `modifyRoute` | `orders` | `routeOrder` | `routes` | `start` | `stop`

Topics

"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21
"Workflow for Bloomberg EMSX" on page 2-2
"Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25

Introduced in R2013a

orders

Obtain Bloomberg EMSX order subscription

Syntax

```
[events,subs] = orders(c,fields)

[events,subs] = orders(c,fields,Name,Value)
[events,subs] = orders(c,fields,options)
```

Description

[events,subs] = orders(c,fields) subscribes to Bloomberg EMSX fields `fields` using the Bloomberg EMSX connection `c`. `orders` returns existing event data `events` from the event queue and the Bloomberg EMSX subscription list `subs`.

[events,subs] = orders(c,fields,Name,Value) uses additional options specified by one or more `Name`,`Value` pair arguments to specify a custom event handler or timeout value for the event handler.

[events,subs] = orders(c,fields,options) uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The `options` structure fields and values correspond to names and values of name-value pair arguments, respectively.

Examples

Subscribe to Order Events Using the Default Event Handler

Create the Bloomberg EMSX connection `c`.

```
c = emsx('//blp/emapisvc_beta');
```

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`.

```
fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

[events,subs] = orders(c,fields)

events =

                        MSG_TYPE: {'E'}
                    MSG_SUB_TYPE: {'O'}
                    EVENT_STATUS: 4
                    ...

subs =

com.bloomberglp.blpapi.SubscriptionList@4bc3dc78
```

events contains fields for the events currently in the event queue. subs contains the
Bloomberg EMSX subscription list object.

Unsubscribe from order events using the Bloomberg EMSX subscription list object subs.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Subscribe to Order Events Using the Custom Event Handler

Create the Bloomberg EMSX connection c.

```
 c = emsx('//blp/emapisvc_beta');
```

Write a custom event handler function named eventhandler. Run the custom event
handler using timer. Start the timer to run eventhandler immediately using start.
For details, see "Writing and Running Custom Event Handler Functions with Bloomberg
EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate');
start(t)
```

t is the timer object.

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection
c and Bloomberg EMSX field list fields. Use the custom event handler by setting the
name-value pair argument 'useDefaultEventHandler' to false.

```
fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

[events,subs] = orders(c,fields,'useDefaultEventHandler',false)

events =

     []


subs =

com.bloomberglp.blpapi.SubscriptionList@2c5b1c7e
```

`events` contains an empty double. The custom event handler processes the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Subscribe to Order Events Using a Timeout

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`. Specify the name-value pair argument `timeOut` and set it to 200 milliseconds.

```
fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

[events,subs] = orders(c,fields,'timeOut',200)

events =
```

```
                        MSG_TYPE: {'E'}
                    MSG_SUB_TYPE: {'O'}
                    EVENT_STATUS: 4
                    ...

subs =

com.bloomberglp.blpapi.SubscriptionList@4bc3dc78
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Subscribe to Order Events Using the Options Structure

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c`, Bloomberg EMSX field list `fields`, and options structure `options`.

```
options.timeOut = 200;
options.useDefaultEventHandler = true;

fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

[events,subs] = orders(c,fields,options)

events =

                        MSG_TYPE: {'E'}
                    MSG_SUB_TYPE: {'O'}
                    EVENT_STATUS: 4
                    ...
```

```
subs =

com.bloomberglp.blpapi.SubscriptionList@4bc3dc78
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`fields` — Bloomberg EMSX field information
cell array

Bloomberg EMSX field information, specified using a cell array. Use `getAllFieldMetaData` to view available field information for the Bloomberg EMSX service.

Example: `'EMSX_TICKER'`
`'EMSX_AMOUNT'`
`'EMSX_ORDER_TYPE'`

Data Types: `cell`

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure instead of name-value pair arguments to reuse the optional name-value pair arguments to specify a custom event handler or timeout value for the event handler.

The `options` structure field and values correspond to names and values of the name-value pair arguments, respectively.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`'  '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'useDefaultEventHandler',false`

`'useDefaultEventHandler'` — Flag for event handler preference
`true` (default) | `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, specified as the comma-separated pair consisting of `'useDefaultEventHandler'` and the logical values `true` or `false`.

To specify the default event handler, set this flag to `true`.

Otherwise, set this flag to `false` to specify a custom event handler.

Data Types: `logical`

`'timeOut'` — Timeout value for event handler
`500` milliseconds (default) | nonnegative integer

Timeout value for event handler for the Bloomberg EMSX service, specified as the comma-separated pair consisting of `'timeOut'` and a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types: double

Output Arguments

events — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, events is a structure containing the current contents of the event queue. Otherwise, events is an empty double.

When the name-value pair argument 'useDefaultEventHandler' or the same field for the structure options is set to false, events is an empty double.

subs — Bloomberg EMSX subscription list
subscription list object

Bloomberg EMSX subscription list, returned as a Bloomberg EMSX subscription list object.

Tips

- For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
timer | close | createOrder | createOrderAndRoute | createOrderAndRouteWithStrat | delete | deleteOrder | deleteRoute | emsx | getAllFieldMetaData | modifyOrder | routeOrder | routes | start | stop

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16

Introduced in R2013a

emsxOrderBlotter

Bloomberg EMSX example order blotter

Syntax

```
[t,subs] = emsxOrderBlotter(c)
```

Description

[t,subs] = emsxOrderBlotter(c) displays a trader's order information. c is the Bloomberg EMSX connection, t is the timer object associated with the event handler, and subs is the Bloomberg EMSX subscription list.

Examples

Display the Order in an Order Blotter

Create the Bloomberg EMSX connection c.

```
c = emsx('//blp/emapisvc_beta');
```

Open Bloomberg EMSX order blotter using the Bloomberg EMSX connection c.

```
[t,subs] = emsxOrderBlotter(c)

   Timer Object: timer-1

   Timer Settings
      ExecutionMode: fixedRate
             Period: 1
            BusyMode: drop
             Running: on

   Callbacks
           TimerFcn: {@processEventToBlotter [1x1 emsx]}
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''
```
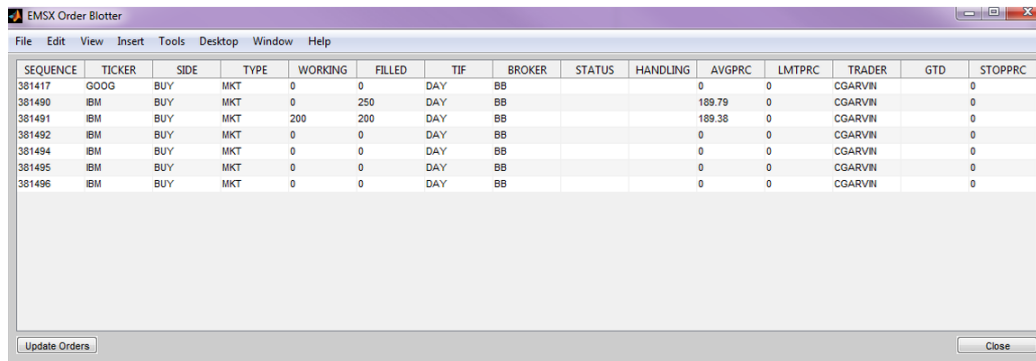
```
subs =

com.bloomberglp.blpapi.SubscriptionList@3e24da58
```

`emsxOrderBlotter` returns the timer object output and the Bloomberg EMSX subscription list object. For details about the timer object, see timer.

SEQUENCE	TICKER	SIDE	TYPE	WORKING	FILLED	TIF	BROKER	STATUS	HANDLING	AVGPRC	LMTPRC	TRADER	GTD	STOPPRC
381417	GOOG	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381490	IBM	BUY	MKT	0	250	DAY	BB			189.79	0	CGARVIN		0
381491	IBM	BUY	MKT	200	200	DAY	BB			189.38	0	CGARVIN		0
381492	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381494	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381495	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381496	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0

The order blotter displays the current order information for a trader.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 330 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(330);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and the order request structure `order`. Use the custom event handler `processEventToBlotter` by setting the name-value pair argument `'useDefaultEventHandler'` to `false`.

```
events = createOrderAndRoute(c,order,'useDefaultEventHandler',false)

events =
```

```
    []

 CreateOrderAndRoute = {

      EMSX_SEQUENCE = 381499

      EMSX_ROUTE_ID = 1

      MESSAGE = Order created and routed

      }
```
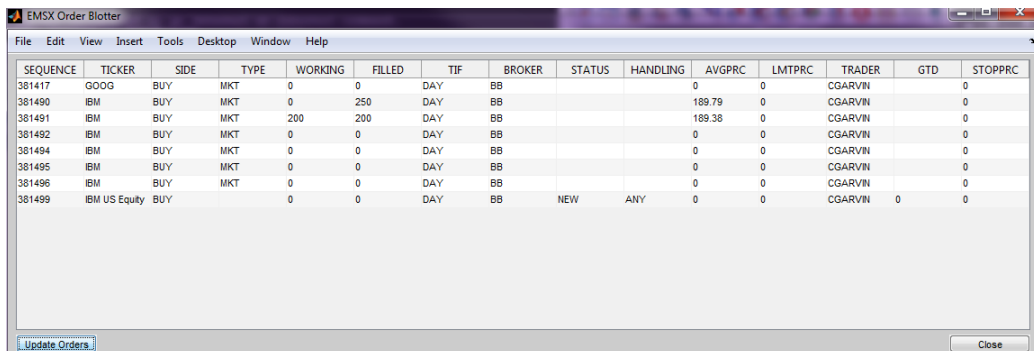
createOrderAndRoute creates the order, routes the order, and returns a structure
events that contains an empty double. processEventToBlotter displays output
from createOrderAndRoute with the order number EMSX_SEQUENCE, route number
EMSX_ROUTE_ID, and message: Order created and routed.

The order blotter updates using the information for the created and routed order,
where order number EMSX_SEQUENCE is 381499, using the event handler function
processEventToBlotter. The order blotter updates as orders are created and
managed.

Close the Bloomberg EMSX connection.

close(c)

- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

Output Arguments

t — MATLAB timer
object

MATLAB timer, returned as a MATLAB object. For details, see timer.

subs — Bloomberg EMSX subscription list
subscription list object

Bloomberg EMSX subscription list, returned as a Bloomberg EMSX subscription list object.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
timer | close | createOrder | createOrder | createOrderAndRoute |
createOrderAndRouteWithStrat | deleteOrder | deleteRoute | emsx |
modifyOrder | orders | routeOrder | routes

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16

Introduced in R2013a

processEvent

Sample Bloomberg EMSX event handler

Syntax

```
processEvent(c)
```

Description

`processEvent(c)` displays and flushes the event queue associated with the Bloomberg EMSX connection `c`. `processEvent` is a sample event handler function. You can build a custom event handler function to process Bloomberg EMSX events.

Examples

Continually Process the Bloomberg EMSX Event Queue

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Use `timer` to continually process the Bloomberg EMSX event queue.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
```

`t` is the MATLAB timer object. For details, see timer.

Close the Bloomberg EMSX connection.

```
close(c)
```

Process the Bloomberg EMSX Event Queue Once

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Use the default event handler function `processEvent` to process the Bloomberg EMSX event queue once.

```
processEvent(c)

SessionConnectionUp = {

    server = "localhost/127.0.0.1:8194"

}

SessionStarted = {

}

ServiceOpened = {

    serviceName = "//blp/emapisvc_beta"

}
```

`processEvent` clears the Bloomberg EMSX event queue.

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
`timer` | `close` | `createOrder` | `createOrderAndRoute` |
`createOrderAndRouteWithStrat` | `deleteOrder` | `deleteRoute` | `emsx` |
`modifyOrder` | `orders` | `routeOrder` | `routes`

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21
"Workflow for Bloomberg EMSX" on page 2-2
"Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25

Introduced in R2013a

routeOrder

Route Bloomberg EMSX order

Syntax

```
events = routeOrder(c,route)
events = routeOrder(c,route,'timeOut',timeout)

routeOrder( ___ ,'useDefaultEventHandler',false)

___ = routeOrder(c,route,options)
```

Description

`events = routeOrder(c,route)` routes a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and route request `route`. `routeOrder` returns a status message using the default event handler.

`events = routeOrder(c,route,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`routeOrder( ___ ,'useDefaultEventHandler',false)` routes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = routeOrder(c,route,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Route an Order Using the Default Event Handler

To route a Bloomberg EMSX order, create the connection c using emsx, set up the order subscription using orders, and create the order using createOrder. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using routes.

Define the route request structure route. Convert the numbers to 32-bit signed integers using int32. This code specifies to route 100 shares of IBM to the broker BB using any hand instruction and the order number 335877.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Route the order using the Bloomberg EMSX connection c and route.

```
events = routeOrder(c,route)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. routeOrder returns events as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects osubs and rsubs. This code assumes that orders creates osubs and routes creates rsubs.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Timeout

To route a Bloomberg EMSX order, create the connection c using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Route the order using the Bloomberg EMSX connection c and `route`. Set the timeout value to 200 milliseconds.

```
events = routeOrder(c,route,'timeOut',200)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
```

```
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Custom Event Handler

To route a Bloomberg EMSX order, create the connection c using emsx, set up the order subscription using orders, and create the order using createOrder. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using routes.

Define the route request structure route. Convert the numbers to 32-bit signed integers using int32. This code specifies to route 100 shares of IBM to the broker BB using any hand instruction and the order number 335877.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Suppose you create a custom event handler function called eventhandler with input argument c. Run eventhandler using timer. Start the timer to run eventhandler immediately using start. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

t is the MATLAB timer object. For details, see timer.

Route the order using the Bloomberg EMSX connection c and route. Set the flag 'useDefaultEventHandler' to false so that eventhandler processes the events associated with routing an order.

```
routeOrder(c,route,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects osubs and rsubs. This code assumes that orders creates osubs and routes creates rsubs. Stop the timer to stop data updates using stop.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using an Options Structure

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Route the order using the Bloomberg EMSX connection `c`, `route`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = routeOrder(c,route,options)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

route — Route request
structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number
EMSX_TICKER	Bloomberg EMSX ticker symbol

Field	Description
EMSX_AMOUNT	Bloomberg EMSX number of shares
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction

Example: `route.EMSX_SEQUENCE = int32(728918);`
`route.EMSX_TICKER = 'XYZ';`
`route.EMSX_AMOUNT = int32(100);`
`route.EMSX_BROKER = 'BB';`
`route.EMSX_HAND_INSTRUCTION = 'ANY';`

Data Types: `struct`

`timeout` — Timeout value
500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
timer | `close` | `createOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `delete` | `deleteOrder` | `deleteRoute` | `emsx` | `modifyOrder` | `orders` | `routeOrderWithStrat` | `routes` | `start` | `stop`

Topics

Introduced in R2013a

groupRouteOrderWithStrat

Route multiple Bloomberg EMSX orders with strategies

Syntax

```
events = groupRouteOrderWithStrat(c,route,strat)
events = groupRouteOrderWithStrat(c,route,strat,'timeOut',timeout)

groupRouteOrderWithStrat( ___ ,'useDefaultEventHandler',false)

 ___ = groupRouteOrderWithStrat(c,route,strat,options)
```

Description

`events = groupRouteOrderWithStrat(c,route,strat)` routes multiple Bloomberg EMSX orders with strategies using the Bloomberg EMSX connection `c`, route request `route`, and strategy `strat`. `routeOrderWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = groupRouteOrderWithStrat(c,route,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`groupRouteOrderWithStrat( ___ ,'useDefaultEventHandler',false)` routes multiple Bloomberg EMSX orders with strategies using any of the input arguments in the previous syntaxes and a custom event handler. To process the events associated with routing orders, write a custom event handler. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = groupRouteOrderWithStrat(c,route,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Route Orders Using the Default Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers 335877 and 335878
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker BMTB
- Any hand instruction
- Time in force set to DAY
- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy SSP. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the orders using the Bloomberg EMSX connection `c`, `route`, and `strat`.

```
events = groupRouteOrderWithStrat(c,route,strat)

events =
```

```
EMSX_SUCCESS_ROUTES: [1x1 struct]
 EMSX_FAILED_ROUTES: [1x1 struct]
             MESSAGE: '1 of 1 Order(s) Routed'
```

where

```
events.EMSX_SUCCESS_ROUTES =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
```

and events.EMSX_FAILED_ROUTES =

```
    EMSX_SEQUENCE: 335878
       ERROR_CODE: 0
    ERROR_MESSAGE: {'Order 335878  View-only orders can not be routed'}
```

The default event handler processes the events associated with routing the order. `groupRouteOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX success routing structure, which contains the order number and route identifier for the orders that successfully routed
- Bloomberg EMSX failed routing structure, which contains the order number, error code, and error message for the orders that failed to route
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Orders Using a Timeout

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers `335877` and `335878`
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker BMTB
- Any hand instruction
- Time in force set to DAY
- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy SSP. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the orders using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = groupRouteOrderWithStrat(c,route,strat,'timeOut',200)

events =

    EMSX_SUCCESS_ROUTES: [1x1 struct]
     EMSX_FAILED_ROUTES: [1x1 struct]
                MESSAGE: '1 of 1 Order(s) Routed'

   where

   events.EMSX_SUCCESS_ROUTES =

       EMSX_SEQUENCE: 335877
```

```
    EMSX_ROUTE_ID: 1

and events.EMSX_FAILED_ROUTES =

    EMSX_SEQUENCE: 335878
       ERROR_CODE: 0
    ERROR_MESSAGE: {'Order 335878  View-only orders can not be routed'}
```

The default event handler processes the events associated with routing the order. `groupRouteOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX success routing structure, which contains the order number and route identifier for the orders that successfully routed
- Bloomberg EMSX failed routing structure, which contains the order number, error code, and error message for the orders that failed to route
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Orders Using a Custom Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers 335877 and 335878
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed

- Broker BMTB
- Any hand instruction
- Time in force set to DAY
- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy SSP. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose that you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using timer. To run `eventhandler` immediately, start the timer using `start`. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see timer.

Route the orders using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with routing an order.

```
groupRouteOrderWithStrat(c,route,strat,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. To stop data updates, stop the timer using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

```
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Orders Using an Options Structure

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers `335877` and `335878`
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker `BMTB`
- Any hand instruction
- Time in force set to `DAY`
- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy SSP. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
```

```
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Route the orders using the Bloomberg EMSX connection `c`, `route`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = groupRouteOrderWithStrat(c,route,strat,options)

events =

    EMSX_SUCCESS_ROUTES: [1x1 struct]
     EMSX_FAILED_ROUTES: [1x1 struct]
                MESSAGE: '1 of 1 Order(s) Routed'

  where

  events.EMSX_SUCCESS_ROUTES =

       EMSX_SEQUENCE: 335877
       EMSX_ROUTE_ID: 1

  and events.EMSX_FAILED_ROUTES =

       EMSX_SEQUENCE: 335878
          ERROR_CODE: 0
       ERROR_MESSAGE: {'Order 335878  View-only orders can not be routed'}
```

The default event handler processes the events associated with routing the order. `groupRouteOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX success routing structure, which contains the order number and route identifier for the orders that successfully routed
- Bloomberg EMSX failed routing structure, which contains the order number, error code, and error message for the orders that failed to route
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- "Create and Manage a Bloomberg EMSX Order" on page 4-12
- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

route — Route request
structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction
EMSX_TIF	Bloomberg EMSX time in force
EMSX_ORDER_TYPE	Bloomberg EMSX order type

Example: `route.EMSX_SEQUENCE = int32(728918);`
`route.EMSX_TICKER = 'XYZ';`

```
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Data Types: `struct`

`strat` — Order strategies
structure

Order strategies, specified as a structure that contains the fields: EMSX_STRATEGY_NAME, EMSX_STRATEGY_FIELD_INDICATORS, and EMSX_STRATEGY_FIELDS. The structure field values must align with the strategy fields specified by EMSX_STRATEGY_NAME. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert EMSX_STRATEGY_FIELD_INDICATORS to a 32-bit signed integer using `int32`. Set EMSX_STRATEGY_FIELD_INDICATORS equal to 0 for each field to use the field data setting in EMSX_FIELD_DATA. Or, set EMSX_STRATEGY_FIELD_INDICATORS equal to 1 to ignore the data in EMSX_FIELD_DATA.

Example: `strat.EMSX_STRATEGY_NAME = 'SSP';`
`strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);`
`strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};`

Data Types: `struct`

`timeout` — Timeout value
500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
`timer` | `close` | `createOrder` | `createOrderAndRoute` |
`createOrderAndRouteWithStrat` | `delete` | `deleteOrder` | `deleteRoute` | `emsx`
| `getBrokerInfo` | `modifyOrder` | `orders` | `routeOrder` | `routeOrderWithStrat`
| `routes` | `start` | `stop`

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21
"Workflow for Bloomberg EMSX" on page 2-2
"Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25

Introduced in R2015b

routeOrderWithStrat

Route Bloomberg EMSX order with strategies

Syntax

```
events = routeOrderWithStrat(c,route,strat)
events = routeOrderWithStrat(c,route,strat,'timeOut',timeout)

routeOrderWithStrat( ___ ,'useDefaultEventHandler',false)

___ = routeOrderWithStrat(c,route,strat,options)
```

Description

`events = routeOrderWithStrat(c,route,strat)` routes a Bloomberg EMSX order with strategies using the Bloomberg EMSX connection `c`, route request `route`, and strategy `strat`. `routeOrderWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = routeOrderWithStrat(c,route,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`routeOrderWithStrat( ___ ,'useDefaultEventHandler',false)` routes a Bloomberg EMSX order with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = routeOrderWithStrat(c,route,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Route an Order Using the Default Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`.

```
events = routeOrderWithStrat(c,route,strat)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Timeout

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = routeOrderWithStrat(c,route,strat,'timeOut',200)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Custom Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using timer. Start the timer to run `eventhandler`

immediately using start. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

t is the MATLAB timer object. For details, see timer.

Route the order using the Bloomberg EMSX connection c, route, and strat. Set the flag 'useDefaultEventHandler' to false so that eventhandler processes the events associated with routing an order.

```
routeOrderWithStrat(c,route,strat,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects osubs and rsubs. This code assumes that orders creates osubs and routes creates rsubs. Stop the timer to stop data updates using stop.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using delete.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using an Options Structure

To route a Bloomberg EMSX order with strategies, create the connection c using emsx, set up the order subscription using orders, and create the order using createOrder. For an example showing these activities, see "Create and Manage a Bloomberg EMSX Order" on page 4-12. Set up the route subscription using routes.

Define the route request structure route. Convert the numbers to 32-bit signed integers using int32. This code specifies to route 100 shares of IBM to the broker BMTB using any hand instruction and the order number 335877.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
```

```
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Route the order using the Bloomberg EMSX connection `c`, `route`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
```

```
events = routeOrderWithStrat(c,route,strat,options)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- "Create and Manage a Bloomberg EMSX Order" on page 4-12

- "Create and Manage a Bloomberg EMSX Route" on page 4-16
- "Manage a Bloomberg EMSX Order and Route" on page 4-21

Input Arguments

c — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

route — Route request
structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction

Example: `route.EMSX_SEQUENCE = int32(728918);`
`route.EMSX_TICKER = 'XYZ';`
`route.EMSX_AMOUNT = int32(100);`
`route.EMSX_BROKER = 'BB';`
`route.EMSX_HAND_INSTRUCTION = 'ANY';`

Data Types: `struct`

strat — Order strategies
structure

Order strategies, specified as a structure that contains the fields:
`EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and

EMSX_STRATEGY_FIELDS. The structure field values must align with the strategy fields specified by EMSX_STRATEGY_NAME. For details about strategy fields and ordering, see getBrokerInfo.

Convert EMSX_STRATEGY_FIELD_INDICATORS to a 32-bit signed integer using int32. Set EMSX_STRATEGY_FIELD_INDICATORS equal to 0 for each field to use the field data setting in EMSX_FIELD_DATA. Or, set EMSX_STRATEGY_FIELD_INDICATORS equal to 1 to ignore the data in EMSX_FIELD_DATA.

Example: strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};

Data Types: struct

timeout — Timeout value
500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time in milliseconds the event handler listens to the queue for an event for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: double

options — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the options structure.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: options.useDefaultEventHandler = false;
options.timeOut = 500;

Data Types: struct

Output Arguments

events — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Tips

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

See Also
timer | close | createOrder | createOrderAndRoute | createOrderAndRouteWithStrat | delete | deleteOrder | deleteRoute | emsx | getBrokerInfo | modifyOrder | orders | routeOrder | routes | start | stop

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21
"Workflow for Bloomberg EMSX" on page 2-2
"Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25

Introduced in R2013a

routes

Obtain Bloomberg EMSX route subscription

Syntax

```
[events,subs] = routes(c,fields)

[events,subs] = routes(c,fields,Name,Value)
[events,subs] = routes(c,fields,options)
```

Description

`[events,subs] = routes(c,fields)` subscribes to Bloomberg EMSX fields `fields` using the Bloomberg EMSX connection `c`. `routes` returns existing event data `events` from the event queue and the Bloomberg EMSX subscription list `subs`.

`[events,subs] = routes(c,fields,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments to specify a custom event handler or timeout value for the event handler.

`[events,subs] = routes(c,fields,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The `options` structure fields and values correspond to names and values of name-value pair arguments, respectively.

Examples

Set Up Route Subscription Using the Default Event Handler

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,subs] = routes(c,fields)

events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...

subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from route events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Set Up Route Subscription Using a Custom Event Handler

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Write a custom event handler function named `eventhandler`. Run the custom event handler using timer. Start the timer to run `eventhandler` immediately using `start`. For details, see "Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
        'ExecutionMode','fixedRate');
start(t)
```

`t` is the timer object.

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Use the custom event handler by setting the name-value pair argument `'useDefaultEventHandler'` to `false`.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,subs] = routes(c,fields,'useDefaultEventHandler',false)

events =

     []

subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` is an empty double. The custom event handler processes the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from route events using the Bloomberg EMSX subscription list object `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Set Up Route Subscription Using a Timeout

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Specify the name-value pair argument `timeOut` and set it to 200 milliseconds.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,subs] = routes(c,fields,'timeOut',200)

events =
```

```
                              MSG_TYPE: {5x1 cell}
                          MSG_SUB_TYPE: {5x1 cell}
                          EVENT_STATUS: [5x1 int32]
                          ...

subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

events contains fields for the events currently in the event queue. subs contains the Bloomberg EMSX subscription list object.

Unsubscribe from route events using the Bloomberg EMSX subscription list object subs.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Set Up Route Subscription Using an Options Structure

Create the Bloomberg EMSX connection c.

```
 c = emsx('//blp/emapisvc_beta');
```

Create a structure options. To use the default event handler, set the field useDefaultEventHandler to true. Set the field timeOut to 200 milliseconds. Set up the route subscription for Bloomberg EMSX fields EMSX_BROKER and EMSX_WORKING using the Bloomberg EMSX connection c and options structure options.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,subs] = routes(c,fields,options)

events =

                              MSG_TYPE: {5x1 cell}
                          MSG_SUB_TYPE: {5x1 cell}
                          EVENT_STATUS: [5x1 int32]
                          ...
```

```
subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from route events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection
connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`fields` — Bloomberg EMSX field information
cell array

Bloomberg EMSX field information, specified using a cell array. Use `getAllFieldMetaData` to view available field information for the Bloomberg EMSX service.

Example: `'EMSX_TICKER'`
`'EMSX_AMOUNT'`
`'EMSX_ORDER_TYPE'`

Data Types: `cell`

`options` — Options for custom event handler or timeout value
structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure instead of name-value pair arguments to reuse the optional name-value pair arguments to specify a custom event handler or timeout value for the event handler.

The `options` structure field and values correspond to names and values of the name-value pair arguments, respectively.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`'  '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'useDefaultEventHandler',false`

`'useDefaultEventHandler'` — Flag for event handler preference
`true` (default) | `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, specified as the comma-separated pair consisting of `'useDefaultEventHandler'` and the logical values `true` or `false`.

To specify the default event handler, set this flag to `true`.

Otherwise, set this flag to `false` to specify a custom event handler.

Data Types: `logical`

`'timeOut'` — Timeout value for event handler
`500` milliseconds (default) | nonnegative integer

Timeout value for event handler for the Bloomberg EMSX service, specified as the comma-separated pair consisting of `'timeOut'` and a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types: `double`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

When the name-value pair argument `'useDefaultEventHandler'` or the same field for the structure `options` is set to `false`, `events` is an empty double.

`subs` — Bloomberg EMSX subscription list
subscription list object

Bloomberg EMSX subscription list, returned as a Bloomberg EMSX subscription list object.

Tips

*   For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.
*   Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using this code.

    ```
    t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
              'ExecutionMode','fixedRate')
    ```

    `t` is the MATLAB timer object. For details, see timer.

See Also

See Also
timer | close | createOrder | createOrderAndRoute |
createOrderAndRouteWithStrat | delete | deleteOrder | deleteRoute | emsx

| `getAllFieldMetaData` | `modifyOrder` | `modifyRoute` | `orders` | `routeOrder` | `start` | `stop`

Topics
"Create and Manage a Bloomberg EMSX Order" on page 4-12
"Create and Manage a Bloomberg EMSX Route" on page 4-16
"Manage a Bloomberg EMSX Order and Route" on page 4-21
"Workflow for Bloomberg EMSX" on page 2-2
"Writing and Running Custom Event Handler Functions with Bloomberg EMSX" on page 1-25

Introduced in R2013a

xtrdr

Create X_TRADER connection

Syntax

```
X = xtrdr
```

Description

`X = xtrdr` starts X_TRADER or connects to an existing X_TRADER session.

Examples

Create a Connection to X_TRADER

```
X = xtrdr

x =

  xtrdr with properties:

          Gate: [1x1 COM.Xtapi_TTGate_1]
    InstrNotify: []
     Instrument: []
       OrderSet: []
```

- "Create an Order Using X_TRADER" on page 1-17
- "Listen for X_TRADER Price Updates" on page 4-2
- "Listen for X_TRADER Price Market Depth Updates" on page 4-4
- "Submit X_TRADER Orders" on page 4-8

Output Arguments

X — X_TRADER connection
connection object

X_TRADER connection, returned as a connection object for an X_TRADER session.

Limitations

*   You should only create one X_TRADER connection per MATLAB session. To create a new X_TRADER connection, start a new MATLAB session.

See Also

See Also
close

Topics
"Create an Order Using X_TRADER" on page 1-17
"Listen for X_TRADER Price Updates" on page 4-2
"Listen for X_TRADER Price Market Depth Updates" on page 4-4
"Submit X_TRADER Orders" on page 4-8
"Workflows for Trading Technologies X_TRADER" on page 2-4

External Websites
X_TRADER API

Introduced in R2013a

close

Close X_TRADER connection

Syntax

```
close(X)
```

Description

close(X) closes the X_TRADER connection X.

Examples

Close X_TRADER Connection

```
close(X)
```

- "Create an Order Using X_TRADER" on page 1-17
- "Listen for X_TRADER Price Updates" on page 4-2
- "Listen for X_TRADER Price Market Depth Updates" on page 4-4
- "Submit X_TRADER Orders" on page 4-8

Input Arguments

X — X_TRADER connection
connection object

X_TRADER connection, specified as a connection object created using xtrdr.

See Also

See Also
xtrdr

Topics

External Websites

X_TRADER API

Introduced in R2013a

createInstrument

Create instrument for X_TRADER

Syntax

```
createInstrument(c,s)
createInstrument(c,Name,Value)
```

Description

`createInstrument(c,s)` creates the X_TRADER instrument defined by the structure `s` with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

`createInstrument(c,Name,Value)` creates the instrument using one or more `Name,Value` pair arguments with names and values corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Examples

Create an X_TRADER Instrument Using an Input Structure

The instruments used in these examples continually expire. To ensure you use a current instrument, see the **Market Explorer** in X_TRADER Pro.

Create the X_TRADER connection.

```
c = xtrdr;
```

Define an input structure `s` with fields corresponding to valid X_TRADER API options. For example, create the input structure for Euro-Bobl Futures.

```
s = [];
s.Exchange = 'Eurex';
s.Product = 'OGBM';
```

```
s.ProdType = 'Option';
s.Contract = 'Jan12 P12300';
s.Alias = 'TestInstrument3';
s

s =

    Exchange: 'Eurex'
     Product: 'OGBM'
    ProdType: 'Option'
    Contract: 'Jan12 P12300'
       Alias: 'TestInstrument3'
```

Requirement: Restart the MATLAB session before reusing an `'Alias'` setting.

Create an X_TRADER instrument.

```
createInstrument(c,s)
```

Close the connection.

```
close(c)
```

Create an X_TRADER Instrument Using Name-Value Pairs

Create the X_TRADER connection.

```
c = xtrdr;
```

Create an X_TRADER instrument for Euro-Bobl Futures using name-value pair arguments corresponding to valid X_TRADER API options.

```
createInstrument(c,'Exchange','Eurex','Product','OGBM',...
                'ProdType','Option','Contract','Jan12 P12300',...
                'Alias','TestInstrument3')
```

Close the connection.

```
close(c)
```

Retrieve Data Using Multiple X_TRADER Instruments

Create the X_TRADER connection.

```
c = xtrdr;
```

Create an X_TRADER instrument for Euro-Bobl Futures using name-value pair arguments corresponding to valid X_TRADER API options.

```
createInstrument(c,'Exchange','Eurex','Product','OGBM',...
                'ProdType','Option','Contract','Jun14 P127',...
                'Alias','PriceInstrumentEurex')
```

Create another X_TRADER instrument for CAISO NP15 EZ Gen Hub 5 MW Peak Calendar-Day Real-Time LMP Futures using name-value pair arguments corresponding to valid X_TRADER API options. This contract expires in April 2014.

```
createInstrument(c,'Exchange','CME','Product','2F',...
                'ProdType','Future','Contract','Apr14',...
                'Alias','PriceInstrumentCMEApr14')
```

Create another X_TRADER instrument for CAISO NP15 EZ Gen Hub 5 MW Peak Calendar-Day Real-Time LMP Futures using name-value pair arguments corresponding to valid X_TRADER API options. This contract expires in October 2014.

```
createInstrument(c,'Exchange','CME','Product','2F',...
                'ProdType','Future','Contract','Oct14',...
                'Alias','PriceInstrumentCMEOct14')
```

Retrieve the exchange and product identifier for all three X_TRADER instruments.

```
d = getData(c,{'Exchange','Product'})

d =
    Exchange: {3x1 cell}
     Product: {3x1 cell}
```

d is a structure containing the Exchange and Product fields. The fields are cell arrays.

Display the Exchange field.

```
d.Exchange

ans =
    'Eurex'
    'CME'
    'CME'
```

The `Exchange` field contains the exchange names Eurex and CME for the three X_TRADER instruments.

Close the connection.

```
close(c)
```

- "Create an Order Using X_TRADER" on page 1-17
- "Listen for X_TRADER Price Updates" on page 4-2
- "Listen for X_TRADER Price Market Depth Updates" on page 4-4
- "Submit X_TRADER Orders" on page 4-8

Input Arguments

c — X_TRADER connection
connection object

X_TRADER connection, specified as a connection object created using `xtrdr`.

s — X_TRADER input structure
structure

X_TRADER input structure, specified using fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Caution: If the symbols for the exchange are entered incorrectly or the exchange server is down, an error appears. For example, if the exchange is "CME" and the CME exchange server is down, then this error appears: The price server for the Exchange CME is down. Unable to create instrument.

```
Example: s = [];
s.Exchange = 'Eurex';
s.Product = 'OGBM';
s.ProdType = 'Option';
s.Contract = 'Jan12 P12300';
```

```
s.Alias = 'TestInstrument3';
```

Data Types: `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
createInstrument(X,'Exchange','Eurex','Product','OGBM','ProdType','Option','Co
P12300','Alias','TestInstrument3')
```

`'Property1'` — Valid X_TRADER API options
character vector

Valid X_TRADER API options, specified as a character vector using information in the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Requirements:

- When using the `'Alias'` name-value pair argument, ensure that every `'Alias'` name is unique across all X_TRADER instruments.
- Restart the MATLAB session before reusing an `'Alias'` name.

Otherwise, `createInstrument` returns an error.

Data Types: `char`

`'Property2'` — Valid X_TRADER API options
character vector

Valid X_TRADER API options, specified as a character vector using information in the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Data Types: `char`

See Also

See Also
`createNotifier | createOrderProfile | createOrderSet | xtrdr`

Topics
"Create an Order Using X_TRADER" on page 1-17
"Listen for X_TRADER Price Updates" on page 4-2
"Listen for X_TRADER Price Market Depth Updates" on page 4-4
"Submit X_TRADER Orders" on page 4-8
"Workflows for Trading Technologies X_TRADER" on page 2-4

External Websites
X_TRADER API

Introduced in R2013a

createNotifier

Create instrument notifier for X_TRADER

Syntax

```
createNotifier(X,S)
createNotifier(X,Name,Value)
```

Description

createNotifier(X,S) creates the xtrdr instrument notifier defined by the structure S with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

createNotifier(X,Name,Value) creates the instrument notifier using X_TRADER API options specified by one or more Name,Value pair arguments with names and values corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Examples

Create an X_TRADER Instrument Notifier Using an Input Structure

Start X_TRADER.

```
X = xtrdr;
```

Define an input structure, S, with fields corresponding to valid X_TRADER API options.

```
S = [];
S.Instrument = [];
S.UpdateFilter = '';
S.EnablePriceUpdates = -1;
```

```
S.EnableDepthUpdates = 0;
S.DebugLogLevel = 3;
S.EnableOrderSetUpdates = -1;
S.PriceList = [];
S.DeliverAllPriceUpdates = 0;
S

S =

                Instrument: []
              UpdateFilter: ''
        EnablePriceUpdates: -1
        EnableDepthUpdates: 0
            DebugLogLevel: 3
    EnableOrderSetUpdates: -1
                 PriceList: []
   DeliverAllPriceUpdates: 0
```

Create an `xtrdr` instrument notifier.

```
createNotifier(X,S)
```

Close the connection.

```
close(X)
```

Create an X_TRADER Instrument Notifier Using Name-Value Pairs

Start X_TRADER.

```
X = xtrdr;
```

Create an `xtrdr` instrument using name-value pairs corresponding to valid X_TRADER API options.

```
createNotifier(X,'Instrument',[],'UpdateFilter','',...
               'EnablePriceUpdates',-1,'EnableDepthUpdates',0,...
               'DebugLogLevel',3,'EnableOrderSetUpdates',-1,...
               'PriceList',[],'DeliverAllPriceUpdates',0)
```

Close the connection.

```
close(X)
```

- "Listen for X_TRADER Price Updates" on page 4-2

Input Arguments

X — X_TRADER connection
connection object

X_TRADER connection, specified as a connection object created using `xtrdr`.

S — `xtrdr` input structure with fields
structure

`xtrdr` input structure, specified with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example: `S = [];`
`S.Exchange = 'Eurex';`
`S.Product = 'OGBM';`
`S.ProdType = 'Option';`
`S.Contract = 'Jan12 P12300';`
`S.Alias = 'TestInstrument3';`

Data Types: `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `createNotifier(X,'Instrument', [],'UpdateFilter','','EnablePriceUpdates',-1,'EnableDepthUpdates',O,'DebugLogL [],'DeliverAllPriceUpdates',O)`

`'Property1'` — Valid X_TRADER API options
character vector

Valid X_TRADER API options, specified as a character vector using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example: `createNotifier(X,'Instrument', [],'UpdateFilter','','EnablePriceUpdates',-1,'EnableDepthUpdates',0,'DebugLogL [],'DeliverAllPriceUpdates',0)`

Data Types: `char`

`'Property2'` — Valid X_TRADER API options
character vector

Valid X_TRADER API options, specified as a character vector using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example: `createNotifier(X,'Instrument', [],'UpdateFilter','','EnablePriceUpdates',-1,'EnableDepthUpdates',0,'DebugLogL [],'DeliverAllPriceUpdates',0)`

Data Types: `char`

See Also

See Also
`createInstrument` | `createOrderProfile` | `createOrderSet` | `xtrdr`

Topics
"Listen for X_TRADER Price Updates" on page 4-2
"Listen for X_TRADER Price Market Depth Updates" on page 4-4
"Submit X_TRADER Orders" on page 4-8
"Workflows for Trading Technologies X_TRADER" on page 2-4

External Websites
X_TRADER API

Introduced in R2013a

createOrderProfile

Create order profile for X_TRADER

Syntax

```
P = createOrderProfile(X,S)
P = createOrderProfile(X,Name,Value)
```

Description

`P = createOrderProfile(X,S)` creates an order profile defined by the structure S with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

`P = createOrderProfile(X,Name,Value)` creates an order profile using X_TRADER API options specified by one or more `Name,Value` pair arguments with names and values corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Examples

Create an Order Profile Using an Input Structure

Start X_TRADER.

```
X = xtrdr;
```

Define an input structure, S, with fields corresponding to valid X_TRADER API options.

```
S = [];
S.Instrument = [];
S.Customer = '';
S.Alias = '';
S.ReadProperties = 'b';
S.WriteProperties = 'b';
```

```
S.Customers = {'<Default>'};
S.RoundOption = 2;
S.CustomerDefaults = [];
S

S =

          Instrument: []
            Customer: ''
               Alias: ''
      ReadProperties: 'b'
     WriteProperties: 'b'
           Customers: {'<Default>'}
         RoundOption: 2
     CustomerDefaults: []
```

Create an order profile.

```
P = createOrderProfile(X,S);
```

Close the connection.

```
close(X)
```

Create an Order Profile Using Name-Value Pairs

Start X_TRADER.

```
X = xtrdr;
```

Create an order profile using name-value pairs corresponding to valid X_TRADER API options.

```
createOrderProfile(X,'Instrument',[],'Customer','',...
            'Alias','','ReadProperties','b',...
            'WriteProperties','b','Customers',{'<Default>'},...
            'RoundOption',2,'CustomerDefaults',[])
```

Close the connection.

```
close(X)
```

- "Create an Order Using X_TRADER" on page 1-17
- "Listen for X_TRADER Price Updates" on page 4-2
- "Listen for X_TRADER Price Market Depth Updates" on page 4-4

- "Submit X_TRADER Orders" on page 4-8

Input Arguments

X — X_TRADER connection
connection object

X_TRADER connection, specified as a connection object created using `xtrdr`.

S — xtrdr input structure with fields
structure

`xtrdr` input structure, specified with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example: `S = [];`
`S.Exchange = 'Eurex';`
`S.Product = 'OGBM';`
`S.ProdType = 'Option';`
`S.Contract = 'Jan12 P12300';`
`S.Alias = 'TestInstrument3';`

Data Types: `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`'  '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `createOrderProfile(X,'Instrument',` `[],'Customer','<Default>','Alias','','RoundOption',2,'CustomerDefaults')`

`'Property1'` — Valid X_TRADER API options
character vector

Valid X_TRADER API options, specified as a character vector using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example: `createOrderProfile(X,'Instrument', [],'Customer','<Default>','Alias','','RoundOption',2,'CustomerDefaults')`

Data Types: `char`

`'Property2'` — Valid X_TRADER API options
character vector

Valid X_TRADER API options, specified as a character vector using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example: `createOrderProfile(X,'Instrument', [],'Customer','<Default>','Alias','','RoundOption',2,'CustomerDefaults')`

Data Types: `char`

Output Arguments

P — Order profile
structure

Order profile, returned as a structure.

See Also

See Also
`createInstrument` | `createNotifier` | `createOrderSet` | `xtrdr`

Topics
"Create an Order Using X_TRADER" on page 1-17
"Listen for X_TRADER Price Updates" on page 4-2
"Listen for X_TRADER Price Market Depth Updates" on page 4-4
"Submit X_TRADER Orders" on page 4-8
"Workflows for Trading Technologies X_TRADER" on page 2-4

External Websites
X_TRADER API

Introduced in R2013a

createOrderSet

Create order set for X_TRADER

Syntax

```
createOrderSet(X)
createOrderSet(X,S)
createOrderSet(X,Name,Value)
```

Description

`createOrderSet(X)` creates an `xtrdr` order set with empty properties. You can set the properties individually using X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

`createOrderSet(X,S)` creates an `xtrdr` order set defined by the structure `S` with fields corresponding to X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

`createOrderSet(X,Name,Value)` creates an order set using X_TRADER API options specified by one or more `Name,Value` pair arguments with names and values corresponding to X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Examples

Create an Empty Order Set

Start X_TRADER.

```
X = xtrdr;
```

Create an order set without any properties.

```
createOrderSet(X)
```

Close the connection.

```
close(X)
```

Create an Order Set Using an Input Structure

Start X_TRADER.

```
X = xtrdr;
```

Define an input structure, S, with fields corresponding to X_TRADER API options.

```
S = [];
S.Count = 0;
S.Alias = '';
S.ReadProperties = 'b';
S.WriteProperties = 'b';
S.EnableOrderSetUpdates = -1;
S.EnableOrderFillData = 0;
S.EnableOrderSend = 0;
S.EnableOrderAutoDelete = 0;
S.QuotingOrderProfile = [];
S.DebugLogLevel = 3;
S.QuoteWithCancelReplace = 0;
S.EnableOrderUpdateData = 0;
S.EnableFillCaching = 0;
S.AvgOpenPriceMode = 'NONE';
S.EnableOrderRejectData = 0;
S.OrderStatusNotifyMode = 'ORD_NOTIFY_NONE';
```

Create an order set.

```
createOrderSet(X,S)
```

Close the connection.

```
close(X)
```

Create an Order Set Using Name-Value Pair Arguments

Start X_TRADER.

```
X = xtrdr;
```

Create an order set using name-value pair arguments corresponding to X_TRADER API options.

```
createOrderSet(X,'Count',0,'Alias','','ReadProperties','b',...
               'WriteProperties','b','EnableOrderSetUpdates',-1,...
               'EnableOrderFillData',0,'EnableOrderSend',0,...
               'EnableOrderAutoDelete',0,'QuotingOrderProfile',[],...
               'DebugLogLevel,3,'QuoteWithCancelReplace',0,...
               'EnableOrderUpdateData',0,'EnableFillCaching',0,...
               'AvgOpenPriceMode','NONE','EnableOrderRejectData',0,...
               'OrderStatusNotifyMode','ORD_NOTIFY_NONE')
```

Close the connection.

```
close(X)
```

Input Arguments

X — X_TRADER connection
connection object

X_TRADER connection, specified as a connection object created using `xtrdr`.

S — X_TRADER API properties
structure

X_TRADER API properties, specified as a structure where the field names match the X_TRADER API properties. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example: `S = [];`
```
S.Exchange = 'Eurex';
S.Product = 'OGBM';
S.ProdType = 'Option';
S.Contract = 'Jan12 P12300';
S.Alias = 'TestInstrument3';
```

Data Types: `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
createOrderSet(X,'Count',O,'Alias','','ReadProperties','b','WriteProperties','
[]'DebugLogLevel,3,'QuoteWithCancelReplace',O,'EnableOrderUpdateData',O,'Enabl
```

`'Property1'` — X_TRADER API options
character vector

X_TRADER API options, specified as a character vector using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Data Types: `char`

`'Property2'` — X_TRADER API options
character vector

X_TRADER API options, specified as a character vector using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Data Types: `char`

See Also

See Also
`createInstrument` | `createNotifier` | `createOrderProfile` | `xtrdr`

Topics
"Create an Order Using X_TRADER" on page 1-17
"Listen for X_TRADER Price Updates" on page 4-2
"Listen for X_TRADER Price Market Depth Updates" on page 4-4
"Submit X_TRADER Orders" on page 4-8

"Workflows for Trading Technologies X_TRADER" on page 2-4

External Websites

X_TRADER API

Introduced in R2013a

getData

Obtain current X_TRADER data

Syntax

```
D = getData(X,S,F)
D = getData(X,F)
```

Description

`D = getData(X,S,F)` returns data for the fields `F` for the `xtrdr` instrument object, `S`, with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

`D = getData(X,F)` returns data for the fields `F` for all instruments associated with the `xtrdr` session object, `X`.

Examples

Return Exchange and Last Price for an Instrument

Return the exchange and last price fields for the instrument defined in `x.Instrument(1)`.

```
D = getData(X,X.Instrument(1),{'Exchange','Last'});

D =

    Exchange: {'CME'}
        Last: {'45'}
```

Return Exchange and Last Price for an Alias

Return the exchange and last price fields for the instrument defined by the alias `PriceInstrument1`.

```
D = getData(X,'PriceInstrument1',{'Exchange','Last'});

D =

    Exchange: {'CME'}
        Last: {'45'}
```

Return Exchange and Last Price for All Session Instruments

Return the exchange and last price fields for all instruments associated with the xtrdr session object, X.

```
D = getData(X,{'Exchange','Last'});

D =

    Exchange: {2x1 cell}
        Last: {2x1 cell}
```

Input Arguments

X — X_TRADER connection
connection object

X_TRADER connection, specified as a connection object created using xtrdr.

S — X_TRADER instrument
instrument object

X_TRADER instrument, specified as an instrument object created using createInstrument or aliases with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example: x.Instrument(1)

F — Fields for the instrument object
character vector | cell array of character vectors

Fields for the instrument object or aliases, S, specified as a character vector or cell array of character vectors. F without a corresponding S are fields for all instruments associated with the xtrdr session object, X.

Example: {'Exchange','Last'}

Data Types: char | cell

Output Arguments

D — X_TRADER data
character vectors

X_TRADER data, returned as character vectors. For missing data, D contains a NaN.

See Also

See Also
createInstrument | xtrdr

Topics
"Listen for X_TRADER Price Updates" on page 4-2
"Listen for X_TRADER Price Market Depth Updates" on page 4-4
"Submit X_TRADER Orders" on page 4-8
"Workflows for Trading Technologies X_TRADER" on page 2-4

External Websites
X_TRADER API

Introduced in R2013a

cqg

Create CQG connection object

Syntax

```
c = cqg
```

Description

`c = cqg` creates a CQG connection object `c`.

Examples

Create the CQG Connection Object

Create the CQG connection object using `cqg`.

```
c = cqg

c =

  cqg with properties:

      Handle: [1x1 COM.CQG_CQGCEL_4]
    APIConfig: [1x1 Interface.CQG_4.O_Type_Library_-_Revised_API.ICQGAPIConfig]
```

CQG connection object properties reflect the CQG ActiveX object `Handle` and the API configuration type library specification `APIConfig`.

Display the `Handle` property of `c`.

```
c.Handle

ans =

    COM.CQG_CQGCEL_4
```

Close the CQG connection.

```
close(c)
```

- • "Create an Order Using CQG" on page 1-12
- • "Create CQG Orders" on page 4-45
- • "Request CQG Historical Data" on page 4-51
- • "Request CQG Intraday Tick Data" on page 4-54
- • "Request CQG Real-Time Data" on page 4-58

Output Arguments

c — CQG connection
connection object

CQG connection, returned as a CQG connection object. The properties of this object are as follows:

Property	Description
Handle	CQG ActiveX object
APIConfig	API configuration type library specification

These properties are determined by the CQG API.

See Also

See Also
close | startUp

Topics
"Create an Order Using CQG" on page 1-12
"Create CQG Orders" on page 4-45
"Request CQG Historical Data" on page 4-51
"Request CQG Intraday Tick Data" on page 4-54
"Request CQG Real-Time Data" on page 4-58
"Workflow for CQG" on page 2-8
"Installation" on page 1-3

External Websites

CQG API Reference Guide

Introduced in R2013b

close

Close CQG connection

Syntax

```
close(c)
```

Description

close(c) closes CQG connection c.

Examples

Close the CQG Connection

Create the CQG connection object c using cqg.

```
c = cqg;
```

Create the CQG connection using startUp.

```
startUp(c)
```

Close the connection using the CQG connection object c.

```
close(c)
```

- "Create an Order Using CQG" on page 1-12
- "Create CQG Orders" on page 4-45
- "Request CQG Historical Data" on page 4-51
- "Request CQG Intraday Tick Data" on page 4-54
- "Request CQG Real-Time Data" on page 4-58

Input Arguments

c — CQG connection
connection object

CQG connection, specified as a CQG connection object created using `cqg`.

See Also

See Also
`cqg` | `shutDown`

Topics
"Create an Order Using CQG" on page 1-12
"Create CQG Orders" on page 4-45
"Request CQG Historical Data" on page 4-51
"Request CQG Intraday Tick Data" on page 4-54
"Request CQG Real-Time Data" on page 4-58
"Workflow for CQG" on page 2-8

External Websites
CQG API Reference Guide

Introduced in R2013b

createOrder

Create CQG order

Syntax

```
o = createOrder(c,s,1,account,quantity)
o = createOrder(c,s,2,account,quantity,limitprice)
o = createOrder(c,s,3,account,quantity,stopprice)
o = createOrder(c,s,4,account,quantity,limitprice,stopprice)
```

Description

`o = createOrder(c,s,1,account,quantity)` creates a `CQGOrder` object `o` for a market order of `quantity` shares of CQG instrument `s` using the `CQGAccount` credentials object `account` over the CQG connection `c`.

`o = createOrder(c,s,2,account,quantity,limitprice)` creates a limit order using a CQG limit price `limitprice`.

`o = createOrder(c,s,3,account,quantity,stopprice)` creates a stop order using a CQG stop price `stopprice`.

`o = createOrder(c,s,4,account,quantity,limitprice,stopprice)` creates a stop limit order using CQG limit and stop prices, `limitprice` and `stopprice`.

Examples

Create and Place a Market Order Using a CQGInstrument Object

To create and place a market order for shares of an instrument with the CQG Trader Com API using a `CQGInstrument` object to specify the instrument, create the connection c using `cqg` and `startUp`. Register an event handler for tracking events associated with the connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with the instrument subscription, order and

account. Subscribe to the instrument and create the `CQGInstrument` object `cqgInst`. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see "Create CQG Orders" on page 4-45. See *CQG API Reference Guide* to learn more about event handlers, API configuration properties, and `CQGInstrument` object.

Create a market order that buys one share of the subscribed security `cqgInst` using the account credentials `accountHandle`.

```
quantity = 1;

oMarket = createOrder(c,cqgInst,1,accountHandle,quantity);
oMarket.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oMarket` contains the order. The CQG API executes the market order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG connection.

```
shutDown(c)
```

Create and Place a Market Order Using a CQG Instrument Character Vector

To create and place a market order for shares of an instrument with the CQG Trader Com API, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with instrument subscription, order, and account. Subscribe to the instrument. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see "Create CQG Orders" on page 4-45. To learn more about the event handlers and the API configuration properties, see the *CQG API Reference Guide*.

Create a market order that buys one share of the previously subscribed security `'EZC'` using the defined account credentials `accountHandle`.

```
cqgInstrumentName = 'EZC';
quantity = 1;

oMarket = createOrder(c,cqgInstrumentName,1,accountHandle, ...
```

```
    quantity);
oMarket.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oMarket` contains the order. The CQG API executes the market order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG connection.

```
shutDown(c)
```

Create and Place a Limit Order

To create and place a limit order for shares of an instrument with the CQG Trader Com API using a `CQGInstrument` object to specify the instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with instrument subscription, order and account. Subscribe to the instrument and create the `CQGInstrument` object `cqgInst`. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see "Create CQG Orders" on page 4-45. See *CQG API Reference Guide* to learn more about the event handlers, the API configuration properties, and the `CQGInstrument` object.

To create a limit order, you can use the bid price. Extract the CQG bid object `qtBid` from the previously defined `CQGInstrument` object `cqgInst`.

```
qtBid = cqgInst.get('Bid');
```

Create a limit order that buys one share of the previously subscribed security `cqgInst` using the previously defined account credentials `accountHandle` and `qtBid` for the limit price.

```
quantity = 1;
limitprice = qtBid.get('Price');

oLimit = createOrder(c,cqgInst,2,accountHandle,quantity, ...
    limitprice);
oLimit.Place

ans =
```

```
OrderChanged
```

The `CQGOrder` object `oLimit` contains the order. The CQG API executes the limit order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG connection.

```
shutDown(c)
```

Create and Place a Stop Order

To create and place a stop order for shares of an instrument with the CQG Trader Com API using a `CQGInstrument` object to specify the instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with instrument subscription, order and account. Subscribe to the instrument and create the `CQGInstrument` object `cqgInst`. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see "Create CQG Orders" on page 4-45. See *CQG API Reference Guide* to learn more about the event handlers, the API configuration properties, and the `CQGInstrument` object.

To create a stop order, you can use the trade price. Extract the CQG trade object `qtTrade` from the previously defined `CQGInstrument` object `cqgInst`.

```
qtTrade = cqgInst.get('Trade');
```

Create a stop order that buys one share of the previously subscribed security `cqgInst` using the previously defined account credentials `accountHandle` and `qtTrade` for the stop price.

```
quantity = 1;
stopprice = qtTrade.get('Price');

oStop = createOrder(c,cqgInst,3,accountHandle,quantity, ...
    stopprice);
oStop.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oStop` contains the order. The CQG API executes the stop order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG connection.

```
shutDown(c)
```

Create and Place a Stop Limit Order

To create and place a stop limit order for shares of an instrument with the CQG Trader Com API using a `CQGInstrument` object to specify the instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with instrument subscription, order and account. Subscribe to the instrument and create the `CQGInstrument` object `cqgInst`. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see "Create CQG Orders" on page 4-45. See *CQG API Reference Guide* to learn more about the event handlers, the API configuration properties, and the `CQGInstrument` object.

To create a stop limit order, you can use the bid and trade prices. Extract the CQG bid object `qtBid` and the CQG trade object `qtTrade` from the previously defined `CQGInstrument` object `cqgInst`.

```
qtBid = cqgInst.get('Bid');
qtTrade = cqgInst.get('Trade');
```

Create a stop limit order that buys one share of the subscribed security `cqgInst` using the defined account credentials `accountHandle` and `qtBid` for the limit price and `qtTrade` for the stop price.

```
quantity = 1;
limitprice = qtBid.get('Price');
stopprice = qtTrade.get('Price');

oStopLimit = createOrder(c,cqgInst,4,accountHandle,quantity, ...
    limitprice,stopprice);
oStopLimit.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oStopLimit` contains the order. The CQG API executes the stop limit order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG connection.

```
shutDown(c)
```

Input Arguments

c — CQG connection
connection object

CQG connection, specified as a CQG connection object created using `cqg`.

s — CQG instrument name
character vector | `CQGInstrument` object

CQG instrument name, specified as a character vector or a `CQGInstrument` object, denoting the instrument or security for the order transaction. For more information about creating a `CQGInstrument` object, see the *CQG API Reference Guide*.

Data Types: `char`

`account` — CQG account credentials
`CQGAccount` object

CQG account credentials, specified as a `CQGAccount` object. This object encapsulates all data pertinent to your account. For more information about creating a `CQGAccount` object, see *CQG API Reference Guide*.

`quantity` — CQG order quantity
scalar

CQG order quantity, specified as a scalar denoting the number of shares to order. A positive number denotes a buy and a negative number denotes a sell.

Data Types: `double`

`limitprice` — CQG limit price
double

CQG limit price, specified as a double denoting the limit order price.

Data Types: `double`

`stopprice` — CQG stop price
double

CQG stop price, specified as a double denoting the stop order price.

Data Types: `double`

Output Arguments

o — CQG order
`CQGOrder` object

CQG order, returned as a `CQGOrder` object. This object encapsulates all data necessary to execute a CQG order. For more information about creating a `CQGOrder` object, see *CQG API Reference Guide*.

See Also

See Also
`cqg` | `history` | `realtime` | `timeseries`

Topics
"Create an Order Using CQG" on page 1-12
"Create CQG Orders" on page 4-45
"Request CQG Historical Data" on page 4-51
"Request CQG Intraday Tick Data" on page 4-54
"Request CQG Real-Time Data" on page 4-58
"Workflow for CQG" on page 2-8

External Websites
CQG API Reference Guide

Introduced in R2013b

history

Request CQG historical data

Syntax

```
history(c,s,startdate,enddate,period)
history(c,s,startdate,enddate,period,x)
```

Description

`history(c,s,startdate,enddate,period)` requests CQG historical data asynchronously with bar size `period` between `startdate` and `enddate` for CQG instrument name `s` with CQG connection `c`.

`history(c,s,startdate,enddate,period,x)` requests CQG historical data asynchronously with additional request properties `x`.

Examples

Request CQG Historical Data

To request daily historical data for an instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see "Request CQG Historical Data" on page 4-51. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Request historical daily data for instrument `XYZ.XYZ` for the last 10 days. `XYZ.XYZ` is a sample instrument name. To request historical data for your instrument, substitute the symbol name in `instrument`.

```
instrument = {'Close(XYZ.XYZ)','Open(XYZ.XYZ)'};
startdate = floor(now) - 10;
```

```
enddate = floor(now);
period = 'hpDaily';

history(c,instrument,startdate,enddate,period)
pause(1)
```

MATLAB writes variable `cqgHistoryData` to the Workspace browser.

Display `cqgHistoryData`.

`cqgHistoryData`

```
cqgHistoryData =
    1.0e+05 *
    7.3533    0.0063    0.0063
    7.3533    0.0064    0.0064
    7.3533    0.0065    0.0065
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0066    0.0066
    7.3534    0.0064    0.0064
```

Each row in `cqgHistoryData` represents data for 1 day. The columns in `cqgHistoryData` show the numerical representation of the timestamp, the close price, and the open price for the instrument during the day.

Close the CQG connection.

```
close(c)
```

Request CQG Historical Data with Additional Request Properties

To request daily historical data for an instrument with an additional property, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see "Request CQG Historical Data" on page 4-51. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Pass an additional optional request property by creating the structure `x` and setting the optional property.

```
x.UpdatesEnabled = false;
```

For additional optional properties you can set, see *CQG API Reference Guide*.

Request historical daily data for instrument XYZ.XYZ for the last 10 days using the additional optional request property x. XYZ.XYZ is a sample instrument name. To request historical data for your instrument, substitute the symbol name in instrument.

```
instrument = {'Close(XYZ.XYZ)','Open(XYZ.XYZ)'};
startdate = floor(now) - 10;
enddate = floor(now);
period = 'hpDaily';

history(c,instrument,startdate,enddate,period,x)
pause(1)
```

MATLAB writes the variable cqgHistoryData to the Workspace browser.

Display cqgHistoryData.

```
cqgHistoryData
```

```
cqgHistoryData =
   1.0e+05 *
    7.3533    0.0063    0.0063
    7.3533    0.0064    0.0064
    7.3533    0.0065    0.0065
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0066    0.0066
    7.3534    0.0064    0.0064
```

Each row in cqgHistoryData represents data for 1 day. The columns in cqgHistoryData show the numerical representation of the timestamp, the close price, and the open price for the instrument during the day.

Close the CQG connection.

```
close(c)
```

- "Create CQG Orders" on page 4-45
- "Request CQG Historical Data" on page 4-51

Input Arguments

c — CQG connection
connection object

CQG connection, specified as a CQG connection object created using `cqg`.

s — CQG instrument name
character vector

CQG instrument name, specified as a character vector identifying the instrument or security.

Data Types: `char`

startdate — Start date
date character vector | date scalar

Start date, specified as a starting date character vector or scalar.

Data Types: `double` | `char`

enddate — End date
date character vector | date scalar

End date, specified as an ending date character vector or scalar.

Data Types: `double` | `char`

period — Bar size
`'hpDaily'` (default) | `'hpWeekly'` | `'hpMonthly'` | `'hpQuarterly'` | `'hpSemiannual'` | `'hpYearly'`

Bar size, specified as one of the above values predetermined by the CQG API that denotes the length of time to collect data.

x — CQG request properties
request properties structure

CQG request properties, specified as a CQG request properties structure. Create this structure by writing MATLAB code to set additional optional request properties. For additional optional properties you can set, see *CQG API Reference Guide*.

Example: `x.UpdatesEnabled = false;`

Data Types: `struct`

See Also

See Also
`cqg` | `createOrder` | `realtime` | `timeseries`

Topics
"Create CQG Orders" on page 4-45
"Request CQG Historical Data" on page 4-51
"Request CQG Intraday Tick Data" on page 4-54
"Request CQG Real-Time Data" on page 4-58
"Workflow for CQG" on page 2-8

External Websites
CQG API Reference Guide

Introduced in R2013b

realtime

Subscribe to CQG instrument

Syntax

```
realtime(c,s)
```

Description

realtime(c,s) subscribes to a CQG instrument s using CQG connection c.

Examples

Subscribe to the CQG Instrument

To subscribe to the CQG instrument and get current data, create the connection c using cqg and startUp. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with instrument subscription. For an example demonstrating these activities, see "Request CQG Real-Time Data" on page 4-58. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

With the connection established, subscribe to the instrument. The instrument name must be formatted in the CQG long symbol view. For example, to subscribe to a security tied to corn, type the following.

```
instrument = 'F.US.EZC';
realtime(c,instrument)
```

MATLAB writes the structure variable cqgDataEZC to the Workspace browser.

Display cqgDataEZC.

```
cqgDataEZC(1,1)
```

```
ans =
              Price: {15x1 cell}
             Volume: {15x1 cell}
    ServerTimestamp: {15x1 cell}
          Timestamp: {15x1 cell}
               Type: {15x1 cell}
               Name: {15x1 cell}
            IsValid: {15x1 cell}
         Instrument: {15x1 cell}
          HasVolume: {15x1 cell}
```

`cqgDataEZC` returns the current quotes for the security.

Display data in the `Price` property of `cqgDataEZC`.

```
cqgDataEZC(1,1).Price
```

```
ans =
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [   660.5000]
    []
    []
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [   660.5000]
    [-2.1475e+09]
```

Close the CQG connection.

```
close(c)
```

- "Create an Order Using CQG" on page 1-12
- "Create CQG Orders" on page 4-45
- "Request CQG Historical Data" on page 4-51
- "Request CQG Intraday Tick Data" on page 4-54
- "Request CQG Real-Time Data" on page 4-58

5-183

Input Arguments

c — CQG connection
connection object

CQG connection, specified as a CQG connection object created using `cqg`.

s — CQG instrument name
character vector

CQG instrument name, specified as a character vector identifying the instrument or security.

Data Types: `char`

See Also

See Also
`cqg` | `createOrder` | `history` | `timeseries`

Topics

External Websites
CQG API Reference Guide

Introduced in R2013b

shutDown

Close CQG connection

Syntax

```
shutDown(c)
```

Description

shutDown(c) closes the CQG connection c.

Examples

Close the CQG Connection

Create the CQG connection object using cqg.

```
c = cqg;
```

Create the CQG connection using startUp.

```
startUp(c)
```

Close the CQG connection.

```
shutDown(c)
```

Alternatively, close the CQG connection using close.

```
close(c)
```

Input Arguments

c — CQG connection
connection object

CQG connection, specified as a CQG connection object created using `cqg`.

See Also

See Also
`close` | `cqg` | `startUp`

Topics
"Create CQG Orders" on page 4-45
"Request CQG Historical Data" on page 4-51
"Request CQG Intraday Tick Data" on page 4-54
"Request CQG Real-Time Data" on page 4-58
"Workflow for CQG" on page 2-8

External Websites
CQG API Reference Guide

Introduced in R2013b

startUp

Create CQG connection

Syntax

```
startUp(c)
```

Description

startUp(c) creates the CQG connection c.

Examples

Create the CQG Connection

Create the CQG connection object using cqg.

```
c = cqg;
```

Create the CQG connection.

```
startUp(c)
```

Close the CQG connection.

```
close(c)
```

- "Create an Order Using CQG" on page 1-12
- "Create CQG Orders" on page 4-45
- "Request CQG Historical Data" on page 4-51
- "Request CQG Intraday Tick Data" on page 4-54
- "Request CQG Real-Time Data" on page 4-58

Input Arguments

c — CQG connection
connection object

5-187

CQG connection, specified as a CQG connection object created using `cqg`.

See Also

See Also
`close` | `cqg` | `shutDown`

Topics
"Create an Order Using CQG" on page 1-12
"Create CQG Orders" on page 4-45
"Request CQG Historical Data" on page 4-51
"Request CQG Intraday Tick Data" on page 4-54
"Request CQG Real-Time Data" on page 4-58
"Workflow for CQG" on page 2-8

External Websites
CQG API Reference Guide

Introduced in R2013b

timeseries

Request CQG intraday tick data

Syntax

```
timeseries(c,s,startdate,enddate)
timeseries(c,s,startdate,enddate,[],x)

timeseries(c,s,startdate,enddate,intraday)
timeseries(c,s,startdate,enddate,intraday,x)
```

Description

`timeseries(c,s,startdate,enddate)` requests CQG raw intraday tick data asynchronously between `startdate` and `enddate` for CQG instrument name `s` with CQG connection `c`.

`timeseries(c,s,startdate,enddate,[],x)` requests CQG raw intraday tick data asynchronously without timed bar data using additional request properties `x`.

`timeseries(c,s,startdate,enddate,intraday)` requests CQG timed bar data asynchronously with the aggregated bar value `intraday`.

`timeseries(c,s,startdate,enddate,intraday,x)` requests CQG timed bar data asynchronously with additional request properties `x`.

Examples

Request CQG Intraday Tick Data

To request intraday tick data for an instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see "Request CQG Intraday Tick Data" on page 4-54. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Request intraday tick data for instrument XYZ.XYZ for the last 2 days. XYZ.XYZ is a sample instrument name. To request intraday tick data for your instrument, substitute the symbol name in `instrument`.

```
instrument = 'XYZ.XYZ';
startdate = now - 2;
enddate = now;

timeseries(c,instrument,startdate,enddate)
```

MATLAB writes the structure variable `cqgTickData` to the Workspace browser.

Display `cqgTickData`.

```
cqgTickData
```

```
cqgTickData =
              Timestamp: {2x1 cell}
                  Price: [2x1 double]
                 Volume: [2x1 double]
              PriceType: {2x1 cell}
         CorrectionType: {2x1 cell}
     SalesConditionLabel: {2x1 cell}
      SalesConditionCode: [2x1 double]
           ContributorId: {2x1 cell}
       ContributorIdCode: [2x1 double]
             MarketState: {2x1 cell}
```

`cqgTickData` returns intraday tick data for the specified instrument.

Display the data in the `Timestamp` property of `cqgTickData`.

```
cqgTickData.Timestamp
```

```
ans =
    '4/17/2013 2:14:00 PM'
    '4/18/2013 2:14:00 PM'
```

Close the CQG connection.

```
close(c)
```

Request CQG Intraday Tick Data with Additional Properties

To request intraday tick data for an instrument with an additional property, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events

associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see "Request CQG Intraday Tick Data" on page 4-54. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Pass an additional optional request property by creating the structure x, and setting the optional property. To see only bid tick data, for example, set TickFilter to 'tfBid'.

```
x.TickFilter = 'tfBid';
```

TickFilter and SessionsFilter are the only valid additional optional properties for calling timeseries without a timed bar request. For additional property values you can set, see *CQG API Reference Guide*.

Request intraday tick data for instrument XYZ.XYZ for the last 2 days using the additional optional request property x. XYZ.XYZ is a sample instrument name. To request intraday tick data for your instrument, substitute the symbol name in instrument.

```
instrument = 'XYZ.XYZ';
startdate = now - 2;
enddate = now;

timeseries(c,instrument,startdate,enddate,[],x)
```

MATLAB writes the variable cqgTickData to the Workspace browser.

Display cqgTickData.

```
cqgTickData

cqgTickData =
               Timestamp: {2x1 cell}
                   Price: [2x1 double]
                  Volume: [2x1 double]
               PriceType: {2x1 cell}
          CorrectionType: {2x1 cell}
      SalesConditionLabel: {2x1 cell}
       SalesConditionCode: [2x1 double]
            ContributorId: {2x1 cell}
        ContributorIdCode: [2x1 double]
              MarketState: {2x1 cell}
```

`cqgTickData` returns intraday tick data for the specified instrument.

Display the data in the `Timestamp` property of `cqgTickData`.

```
cqgTickData.Timestamp
```

```
ans =
    '4/17/2013 2:14:00 PM'
    '4/18/2013 2:14:00 PM'
```

Close the CQG connection.

```
close(c)
```

Request CQG Timed Bar Data

To request timed bar data for an instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see "Request CQG Intraday Tick Data" on page 4-54. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Request timed bar data for instrument `XYZ.XYZ` for the last fraction of a day. `XYZ.XYZ` is a sample instrument name. To request timed bar data for your instrument, substitute the symbol name in `instrument`.

```
instrument = 'XYZ.XYZ';
startdate = now - .1;
enddate = now;
intraday = 1;

timeseries(c,instrument,startdate,enddate,intraday)
```

MATLAB writes variable `cqgTimedBarData` to the Workspace browser.

Display `cqgTimedBarData`.

```
cqgTimedBarData
```

```
cqgTimedBarData =
  1.0e+09 *
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
```

```
0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
...
```

cqgTimedBarData returns timed bar data for the specified instrument. The columns of cqgTimedBarData display data corresponding to the timestamp, open price, high price, low price, close price, mid-price, HLC3, average price, and tick volume.

Close the CQG connection.

```
close(c)
```

Request CQG Timed Bar Data with Additional Properties

To request timed bar data for an instrument with an additional property, create the connection c using cqg and startUp. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see "Request CQG Intraday Tick Data" on page 4-54. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Pass an additional optional request property by creating the structure x, and setting the optional property.

```
x.UpdatesEnabled = false;
```

For additional optional properties you can set, see *CQG API Reference Guide*.

Request timed bar data for instrument XYZ.XYZ for the last fraction of a day using the additional optional request property x. XYZ.XYZ is a sample instrument name. To request timed bar data for your instrument, substitute the symbol name in instrument.

```
instrument = 'XYZ.XYZ';
startdate = now - .1;
enddate = now;
intraday = 1;

timeseries(c,instrument,startdate,enddate,intraday,x)
```

MATLAB writes the variable cqgTimedBarData to the Workspace browser.

Display cqgTimedBarData.

```
cqgTimedBarData

cqgTimedBarData =
   1.0e+09 *
      0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
      0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
      0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
      0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
      0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
      ...
```

`cqgTimedBarData` returns timed bar data for the specified instrument. The columns of `cqgTimedBarData` display data corresponding to the timestamp, open price, high price, low price, close price, mid-price, HLC3, average price, and tick volume.

Close the CQG connection.

```
close(c)
```

- "Create CQG Orders" on page 4-45
- "Request CQG Historical Data" on page 4-51
- "Request CQG Intraday Tick Data" on page 4-54
- "Request CQG Real-Time Data" on page 4-58

Input Arguments

c — CQG connection
connection object

CQG connection, specified as a CQG connection object created using `cqg`.

s — CQG instrument name
character vector

CQG instrument name, specified as a character vector identifying the instrument or security.

Data Types: `char`

startdate — Start date
date character vector | date scalar

Start date, specified as a starting date character vector or scalar.

Data Types: `double | char`

enddate — End date
date character vector | date scalar

End date, specified as an ending date character vector or scalar.

Data Types: `double | char`

intraday — Aggregated bar value
scalar | `[]`

Aggregated bar value, specified as a scalar from 1.0 to 1440.0. If you want to call `timeseries` to return intraday tick data with additional properties without timed bar data, then enter `[]` for this argument.

Data Types: `double`

x — CQG request properties
request properties structure

CQG request properties, specified as a CQG request properties structure. Create this structure by writing MATLAB code to set additional optional request properties. For additional optional properties you can set, see *CQG API Reference Guide*.

Example: `x.UpdatesEnabled = false;`

Data Types: `struct`

See Also

See Also
`cqg | createOrder | history | realtime`

Topics
"Create CQG Orders" on page 4-45
"Request CQG Historical Data" on page 4-51
"Request CQG Intraday Tick Data" on page 4-54
"Request CQG Real-Time Data" on page 4-58
"Workflow for CQG" on page 2-8

External Websites

CQG API Reference Guide

Introduced in R2013b

ibtws

Create IB Trader Workstation connection

Syntax

```
ib = ibtws(host,port)
ib = ibtws(host,port,clientid)
```

Description

`ib = ibtws(host,port)` creates a connection to IB Trader Workstation on a machine with IP address `host` and port number `port`. The function returns the IB Trader Workstation connection object `ib`.

`ib = ibtws(host,port,clientid)` specifies the client identifier `clientid`.

Examples

Connect to IB Trader Workstation℠ **on Local Machine**

Create an IB Trader Workstation℠ connection on the local machine and request current data for the IBM® security.

Connect to the IB Trader Workstation℠ using port number 7496.

```
ib = ibtws('',7496)


ib =

  ibtws with properties:

    ClientId: O
      Handle: [1×1 COM.TWS_TwsCtrl_1]
        Host: ''
        Port: 7496
```

MATLAB® returns `ib` as the IB Trader Workstation℠ connection with the Interactive Brokers® ActiveX® object, the local host, and the specified port number.

Display the `Handle` property of `ib`.

```
ib.Handle
```

```
ans =

 COM.TWS_TwsCtrl_1
```

Create the IB Trader Workstation℠ `IContract` object for IBM®. This object describes a security with these property values:

- Security symbol
- Stock security type
- Aggregate exchange
- Primary exchange
- USD currency

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'IBM';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'IEX';
ibContract.currency = 'USD';
```

Request current data using `ibContract`.

```
d = getdata(ib,ibContract)
```

```
d =

  struct with fields:

    BID_PRICE: 160.1900
     BID_SIZE: 1
    ASK_PRICE: 160.2500
     ASK_SIZE: 1
   LAST_PRICE: 160.1100
```

```
LAST_SIZE: 1
   VOLUME: 2877
```

`d` is a structure containing these fields:

- BID_PRICE -- Bid price
- BID_SIZE -- Bid size
- ASK_PRICE -- Ask price
- ASK_SIZE -- Ask size
- LAST_PRICE -- Last price
- LAST_SIZE -- Last size
- VOLUME -- Volume

Display the data in the `BID_PRICE` field of `d`.

`d.BID_PRICE`

```
ans =

  160.1900
```

Close the IB Trader Workstation℠ connection.

`close(ib)`

Connect to IB Trader Workstation on Another Machine

Note: The IP address for this example does not represent a real Interactive Brokers machine.

Connect to the IB Trader Workstation on another machine using the IP address `1111.222.333.44` and port number `7496`.

```
ib = ibtws('1111.222.333.44',7496)

ib =
```

```
  ibtws with properties:

    ClientId: 0
      Handle: [1x1 COM.TWS_TwsCtrl_1]
        Host: '1111.222.333.44'
        Port: 7496
```

MATLAB returns `ib` as the connection to the IB Trader Workstation with the Interactive Brokers ActiveX object, the IP address that you choose, and the specified port number.

Display the `Handle` property of `ib`.

`ib.Handle`

`ans =`

```
    COM.TWS_TwsCtrl_1
```

Close the IB Trader Workstation connection.

`close(ib)`

Connect to IB Trader Workstation℠ Using Client Identifier

Create an IB Trader Workstation℠ connection on the local machine and request current data for the IBM® security.

Connect to the IB Trader Workstation℠ using port number 7496 and client identifier 1.

`ib = ibtws('',7496,1)`

`ib =`

```
  ibtws with properties:

    ClientId: 1
      Handle: [1×1 COM.TWS_TwsCtrl_1]
        Host: ''
        Port: 7496
```

MATLAB® returns `ib` as the IB Trader Workstation℠ connection with the client identifier, Interactive Brokers® ActiveX® object, the local host, and the specified port number.

Display the `ClientId` property of `ib`.

```
ib.ClientId
```

```
ans =

    1
```

Create the IB Trader Workstation℠ `IContract` object for IBM®. This object describes a security with these property values:

- Security symbol
- Stock security type
- Aggregate exchange
- Primary exchange
- USD currency

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'IBM';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'IEX';
ibContract.currency = 'USD';
```

Request current data using `ibContract`.

```
d = getdata(ib,ibContract)
```

```
d =

  struct with fields:

    BID_PRICE: 160.1900
     BID_SIZE: 2
    ASK_PRICE: 160.2500
     ASK_SIZE: 2
   LAST_PRICE: 161.1100
    LAST_SIZE: 1
       VOLUME: 2877
```

d is a structure containing these fields:

- BID_PRICE -- Bid price
- BID_SIZE -- Bid size
- ASK_PRICE -- Ask price
- ASK_SIZE -- Ask size
- LAST_PRICE -- Last price
- LAST_SIZE -- Last size
- VOLUME -- Volume

Display the data in the `BID_PRICE` field of d.

```
d.BID_PRICE
```

```
ans =

  160.1900
```

Close the IB Trader Workstation℠ connection.

```
close(ib)
```

Input Arguments

`host` — IP address of machine where IB Trader Workstation is running
`''` | character vector

IP address of the machine where IB Trader Workstation is running, specified as either `''` or a character vector. `''` specifies the local machine. A character vector specifies the IP address of another machine.

Data Types: char

`port` — IB Trader Workstation port number
numeric scalar

IB Trader Workstation port number, specified as a numeric scalar designating the connection port of the machine.

Data Types: double

`clientid` — IB Trader Workstation client identifier
numeric scalar

IB Trader Workstation client identifier, specified as a numeric scalar designating the client machine. This number must be unique to the client.

Data Types: double

Output Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, returned as an IB Trader Workstation connection object. The properties of this object are as follows.

Property	Description
ClientId	Application identifier where the connection originated
Handle	Interactive Brokers ActiveX object
Host	host argument
Port	port argument

The Interactive Brokers API determines these properties.

Tips

• ibBuiltInErrMsg appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. ibBuiltInErrMsg contains messages related to:

- Connection
- Information resulting from executing functions
- Errors

See Also

See Also
close

Topics
"Create an Order Using IB Trader Workstation" on page 1-8
"Create Interactive Brokers Combination Order" on page 4-39
"Create and Manage an Interactive Brokers Order" on page 4-26
"Request Interactive Brokers Historical Data" on page 4-32
"Request Interactive Brokers Real-Time Data" on page 4-35
"Workflow for Interactive Brokers" on page 2-6

External Websites
Interactive Brokers API Reference Guide

Introduced in R2013b

close

Close IB Trader Workstation connection

Syntax

```
close(ib)
```

Description

close(ib) closes the IB Trader Workstation connection ib.

Examples

Close IB Trader Workstation℠ **Connection**

Create an IB Trader Workstation℠ connection on the local machine, request current data for a security, and close the connection.

Connect to the IB Trader Workstation℠ using port number 7496.

```
ib = ibtws('',7496)

ib =

  ibtws with properties:

    ClientId: O
      Handle: [1×1 COM.TWS_TwsCtrl_1]
        Host: ''
        Port: 7496
```

MATLAB® returns ib as the IB Trader Workstation℠ connection with the Interactive Brokers® ActiveX® object, the local host, and the specified port number.

Display the `Handle` property of `ib`.

```
ib.Handle
```

```
ans =

 COM.TWS_TwsCtrl_1
```

Create the IB Trader Workstation℠ `IContract` object for IBM®. This object describes a security with these property values:

- Security symbol
- Stock security type
- Aggregate exchange
- Primary exchange
- USD currency

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'IBM';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'IEX';
ibContract.currency = 'USD';
```

Request current data using `ibContract`.

```
d = getdata(ib,ibContract)
```

```
d =

  struct with fields:

    BID_PRICE: 160.1900
     BID_SIZE: 2
    ASK_PRICE: 160.2500
     ASK_SIZE: 2
   LAST_PRICE: 160.2200
    LAST_SIZE: 1
       VOLUME: 2877
```

`d` is a structure containing these fields:

- BID_PRICE -- Bid price
- BID_SIZE -- Bid size
- ASK_PRICE -- Ask price
- ASK_SIZE -- Ask size
- LAST_PRICE -- Last price
- LAST_SIZE -- Last size
- VOLUME -- Volume

Display the data in the `BID_PRICE` field of `d`.

```
d.BID_PRICE
```

```
ans =

  160.1900
```

Close the IB Trader Workstation℠ connection.

```
close(ib)
```

- "Create an Order Using IB Trader Workstation" on page 1-8
- "Create Interactive Brokers Combination Order" on page 4-39
- "Create and Manage an Interactive Brokers Order" on page 4-26
- "Request Interactive Brokers Historical Data" on page 4-32
- "Request Interactive Brokers Real-Time Data" on page 4-35

Input Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

See Also

See Also
`getdata` | `ibtws`

Topics
"Create an Order Using IB Trader Workstation" on page 1-8
"Create Interactive Brokers Combination Order" on page 4-39
"Create and Manage an Interactive Brokers Order" on page 4-26
"Request Interactive Brokers Historical Data" on page 4-32
"Request Interactive Brokers Real-Time Data" on page 4-35
"Workflow for Interactive Brokers" on page 2-6

External Websites
Interactive Brokers API Reference Guide

Introduced in R2013b

createOrder

Create IB Trader Workstation order

Syntax

```
d = createOrder(ib,ibContract,ibOrder,id)
d = createOrder(ib,ibContract,ibOrder,id,eventhandler)
```

Description

`d = createOrder(ib,ibContract,ibOrder,id)` creates an IB Trader Workstation order over the IB Trader Workstation connection `ib` using the IB Trader Workstation `IOrder` object `ibOrder` with a unique order identifier `id` to denote the order information. `createOrder` uses the IB Trader Workstation `IContract` object `ibContract` to signify the instrument for the transaction. `createOrder` returns the Interactive Brokers order data `d` containing data about the completed order.

`d = createOrder(ib,ibContract,ibOrder,id,eventhandler)` creates an IB Trader Workstation order using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Create an Order

To create an order, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract`. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. Then, create an IB Trader Workstation `IOrder` object `ibOrder`. An `IOrder` object is an Interactive Brokers object that contains the order conditions to place an order. For an example showing how to create these objects, see "Create and Manage an Interactive Brokers Order" on page 4-26. For details about creating these objects, see Interactive Brokers API Reference Guide.

Obtain the next valid order identification number `id` using `ib`.

```
id = orderid(ib)

id =

    54110686
```

Execute the order using `ib`, `ibContract`, `ibOrder`, and `id`. This code assumes a buy market order for two shares.

```
d = createOrder(ib,ibContract,ibOrder,id)

d =
    STATUS: 'Filled'
    FILLED: 2
    REMAINING: 0
    AVG_FILL_PRICE: 787.5600
    PERM_ID: '1979798454'
    PARENT_ID: 0
    LAST_FILL_PRICE: 787.5600
    CLIENT_ID: 0
    WHY_HELD: ''
```

`d` contains these fields:

- Status
- Filled
- Remaining
- Average fill price
- Permanent identifier
- Parent identifier
- Last fill price
- Client identifier
- Why held

Display the data in the STATUS property of `d`.

```
d(1,1).STATUS

ans =
```

```
    Filled
```

Close the IB Trader Workstation connection.

```
 close(ib)
```

Create an Order Using an Event Handler

To create an order, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract`. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. Then, create an IB Trader Workstation `IOrder` object `ibOrder`. An `IOrder` object is an Interactive Brokers object that contains the order conditions to place an order. For an example showing how to create these objects, see "Create and Manage an Interactive Brokers Order" on page 4-26. For details about creating these objects, see Interactive Brokers API Reference Guide.

Obtain the next valid order identification number `id` using `ib`.

```
id = orderid(ib)

id =

   768409.00
```

Execute the order using `ib`, `ibContract`, `ibOrder`, and `id`. This code assumes a buy market order for two shares. Use the sample event handler function `ibExampleEventHandler` or write a custom event handler function.

```
d = createOrder(ib,ibContract,ibOrder,id,@ibExampleEventHandler)

d =

    768409.00

  Columns 1 through 5

   [1x1 COM.TWS_TwsCtrl_1]    [13.00]    [768409.00]    'Submitted'    [0]

  Columns 6 through 12

   [2.00]    [0]    [1679681704.00]    [0]    [0]    [0]    ''
```

```
Columns 13 through 14

  [1x1 struct]    'orderStatus'
...
```

`d` contains the unique order identifier `id`.

`ibExampleEventHandler` displays order status data in the Command Window. The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Unique order identifier
- Order status
- Filled
- Remaining
- Average fill price
- Permanent identifier
- Parent identifier
- Last fill price
- Client identifier
- Why held
- Structure that repeats the contents of the columns
- Event type

For details about this data, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

- "Create an Order Using IB Trader Workstation" on page 1-8
- "Create Interactive Brokers Combination Order" on page 4-39
- "Create and Manage an Interactive Brokers Order" on page 4-26
- "Request Interactive Brokers Historical Data" on page 4-32

- "Request Interactive Brokers Real-Time Data" on page 4-35

Input Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

`ibContract` — IB Trader Workstation contract
`IContract` object

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object. This object is the instrument or security used in the order transaction. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that you can set, see Interactive Brokers API Reference Guide.

`ibOrder` — IB Trader Workstation order
`IOrder` object

IB Trader Workstation order, specified as an IB Trader Workstation `IOrder` object. This object contains the order conditions, which are: the action of the order, for example, buy or sell; the order quantity; and the type of order, for example, market or limit. Create this object by calling the Interactive Brokers API function `createOrder`. For details about the attributes that you can set and `createOrder`, see Interactive Brokers API Reference Guide.

`id` — IB Trader Workstation order unique identifier
scalar

IB Trader Workstation order unique identifier, specified as a scalar.

Data Types: `double`

`eventhandler` — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write

a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

Output Arguments

d — Interactive Brokers order data
structure | double

Interactive Brokers order data, returned as a structure containing these fields:

- Status
- Filled
- Remaining
- Average fill price
- Permanent identifier
- Parent identifier
- Last fill price
- Client identifier
- Why held

When using an event handler function, `d` is a double containing the unique order identifier.

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

  - Connection
  - Information resulting from executing functions
  - Errors

See Also

See Also
close | getdata | history | ibtws | orderid | realtime | timeseries

Topics
"Create an Order Using IB Trader Workstation" on page 1-8
"Create Interactive Brokers Combination Order" on page 4-39
"Create and Manage an Interactive Brokers Order" on page 4-26
"Request Interactive Brokers Historical Data" on page 4-32
"Request Interactive Brokers Real-Time Data" on page 4-35
"Workflow for Interactive Brokers" on page 2-6
"Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites
Interactive Brokers API Reference Guide

Introduced in R2013b

getdata

Request current Interactive Brokers data

Syntax

```
d = getdata(ib,ibContract)
d = getdata(ib,ibContract,eventhandler)
```

Description

`d = getdata(ib,ibContract)` requests Interactive Brokers current data over the IB Trader Workstation connection `ib` using the IB Trader Workstation `IContract` object `ibContract` to signify the instrument.

`d = getdata(ib,ibContract,eventhandler)` requests Interactive Brokers current data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Request Current Data

To request Interactive Brokers current data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Real-Time Data" on page 4-35. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request current data using `ib` and `ibContract`.

```
d = getdata(ib,ibContract)

d =
```

```
 LAST_PRICE: 6.85
  LAST_SIZE: 1.00
     VOLUME: 187.00
  BID_PRICE: 6.84
   BID_SIZE: 14.00
  ASK_PRICE: 6.86
   ASK_SIZE: 13.00
```

d contains these fields:

- Last price
- Last size
- Volume
- Bid price
- Bid size
- Ask price
- Ask size

Display the data in the BID_PRICE field of d.

```
d.BID_PRICE
```

```
ans =
    6.84
```

Close the IB Trader Workstation connection.

```
 close(ib)
```

Request Current Data Using an Event Handler

To request Interactive Brokers current data, set up the IB Trader Workstation connection ib using ibtws. Create an IB Trader Workstation IContract object ibContract as shown in "Request Interactive Brokers Real-Time Data" on page 4-35. An IContract object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request current data using ib, ibContract, and sample event handler function ibExampleEventHandler. Use ibExampleEventHandler or write a custom event handler function.

```
d = getdata(ib,ibContract,@ibExampleEventHandler)

d =

      1418.00

  Columns 1 through 5

    [1x1 COM.TWS_TwsCtrl_1]    [2.00]    [1418.00]    [0]    [5.00]

  Columns 6 through 7

    [1x1 struct]    'tickSize'
...
```

d is the request identifier.

After d, `ibExampleEventHandler` streams current data to the Command Window. Each column set is a type of tick.

For a size tick, the columns are:

· Interactive Brokers ActiveX object
· Event identifier
· Request identifier
· Tick type
· Size
· Structure that repeats the contents of the columns
· Event type

Close the IB Trader Workstation connection.

```
close(ib)
```

Input Arguments

ib — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using ibtws.

ibContract — IB Trader Workstation contract
IContract object

IB Trader Workstation contract, specified as an IB Trader Workstation IContract object. This object is the instrument or security used in the order transaction. Create this object by calling the Interactive Brokers API function createContract. For details about createContract and the attributes that you can set, see Interactive Brokers API Reference Guide.

eventhandler — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: @eventhandler

Data Types: function_handle | char

Output Arguments

d — Interactive Brokers current data
structure | double

Interactive Brokers current data, returned as a structure containing these tick types:

- Last price
- Last size
- Volume
- Bid price

- Bid size
- Ask price
- Ask size

When using an event handler function, `d` is a double denoting the request identifier.

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

  - Connection
  - Information resulting from executing functions
  - Errors

See Also

See Also

`close` | `createOrder` | `history` | `ibtws` | `realtime` | `timeseries`

Topics

"Create an Order Using IB Trader Workstation" on page 1-8
"Create Interactive Brokers Combination Order" on page 4-39
"Create and Manage an Interactive Brokers Order" on page 4-26
"Request Interactive Brokers Historical Data" on page 4-32
"Request Interactive Brokers Real-Time Data" on page 4-35
"Workflow for Interactive Brokers" on page 2-6
"Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites

Interactive Brokers API Reference Guide

Introduced in R2013b

history

Request Interactive Brokers historical data

Syntax

```
d = history(ib,ibContract,startdate,enddate)
d = history(ib,ibContract,startdate,enddate,ticktype,period)
d = history(ib,ibContract,startdate,enddate,ticktype,period,
tradehours)
d = history(ib,ibContract,startdate,enddate,ticktype,period,
tradehours,eventhandler)
```

Description

`d = history(ib,ibContract,startdate,enddate)` requests Interactive Brokers historical data using the IB Trader Workstation connection `ib` and IB Trader Workstation `IContract` object `ibContract` to signify the instrument. `history` requests data from `startdate` through `enddate`. The default tick type is `'TRADES'` and default period is `'1 day'`.

`d = history(ib,ibContract,startdate,enddate,ticktype,period)` requests Interactive Brokers historical data for a specific type of market data tick `ticktype` and bar size `period`.

`d = history(ib,ibContract,startdate,enddate,ticktype,period,
tradehours)` requests Interactive Brokers historical data using the flag `tradehours` to include all data or only data within regular trading hours.

`d = history(ib,ibContract,startdate,enddate,ticktype,period,
tradehours,eventhandler)` requests Interactive Brokers historical data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Request Interactive Brokers Historical Data with TRADES Default Tick Type and 1-Day Default Period

To request historical data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Historical Data" on page 4-32. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request the last 5 days of historical data using `ib` and `ibContract`.

```
startdate = floor(now)-5;
enddate = floor(now);

d = history(ib,ibContract,startdate,enddate)

d =

  Columns 1 through 5

    736308.00        751.83        755.85        743.83        749.46
    736309.00        742.69        745.71        736.75        738.20
    736312.00        743.08        748.73        724.17        748.48
    736313.00        752.50        758.08        744.43        747.65

  Columns 6 through 9

     12513.00       9107.00        751.28             0
     15984.00      11121.00        740.39             0
     17125.00      11355.00        736.61             0
      2139.00       2568.00        751.29             0
```

`d` returns the historical data for 5 days. When `ticktype` and `period` are not specified as input arguments, `history` returns historical data using the default `ticktype` of `'TRADES'` and the default `period` of `'1 day'`.

Each row of `d` contains historical data for 1 day. The columns in matrix `d` are:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume

- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the open price for the most recent record in matrix d.

```
d(1,2)
```

```
ans =

    751.83
```

Close the IB Trader Workstation connection.

```
 close(ib)
```

Request Interactive Brokers Historical Data with BID Tick Type and 1-Week Period

To request historical data, set up the IB Trader Workstation connection ib using ibtws. Create an IB Trader Workstation IContract object ibContract as shown in "Request Interactive Brokers Historical Data" on page 4-32. An IContract object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request the last 50 days of historical data using ib, ibContract, and these arguments:

- Start date is 50 days ago.
- End date is the current moment.
- Tick type is 'BID'.
- Bar size is '1W'.

```
startdate = floor(now)-50;
enddate = floor(now);
ticktype = 'BID';
period = '1W';

d = history(ib,ibContract,startdate,enddate,ticktype,period)
d =

  Columns 1 through 5

    736267.00       699.28        720.36        695.10        710.50
    736274.00       710.35        739.20        703.18        732.77
    736281.00       730.00        740.92        711.99        711.99
    736288.00       713.05        757.73        706.00        756.35
    736295.00       755.30        762.70        737.52        748.56
```

```
       736302.00          749.33          775.81          740.00          766.15
       736309.00          765.00          768.18          735.57          738.20
       736312.00          738.87          757.77          700.00          747.84

   Columns 6 through 9

          -1.00           -1.00           -1.00               0
          -1.00           -1.00           -1.00               0
          -1.00           -1.00           -1.00               0
          -1.00           -1.00           -1.00               0
          -1.00           -1.00           -1.00               0
          -1.00           -1.00           -1.00               0
          -1.00           -1.00           -1.00               0
          -1.00               0           -1.00               0
```

d returns the historical data for 50 days.

Each row of d contains historical data for 1 week.

The columns in matrix d are:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the high price for the most recent record in matrix d.

```
d(1,3)
```

```
ans =

    720.36
```

Close the IB Trader Workstation connection.

```
 close(ib)
```

Request Interactive Brokers Historical Data with TRADES Default Tick Type and 1-Month Period

To request historical data, set up the IB Trader Workstation connection ib using ibtws. Create an IB Trader Workstation IContract object ibContract as shown in "Request

Interactive Brokers Historical Data" on page 4-32. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request the last 50 days of historical data using `ib`, `ibContract`, and these arguments:

- Start date is 50 days ago.
- End date is the current moment.
- The empty character vector denotes the default tick type `'TRADES'`.
- Bar size is `'1M'`.

```
startdate = floor(now)-50;
enddate = floor(now);
ticktype = '';
period = '1M';

d = history(ib,ibContract,startdate,enddate,ticktype,period)

d =

  Columns 1 through 5

      736267.00         661.18         738.42         641.64         710.85
      736298.00         712.00         762.71         705.85         742.60
      736312.00         745.50         775.96         724.17         748.73

  Columns 6 through 9

      186268.00      127222.00         692.28              0
      234490.00      160672.00         734.32              0
      151754.00      102702.00         754.11              0
```

`d` returns the historical data for 50 days.

Each row of `d` contains historical data for 1 month.

The columns in matrix `d` are:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count

- Weighted average price
- Flag indicating if there are gaps in the bar

Display the low price for the most recent record in matrix d.

```
d(1,4)
```

```
ans =

    641.64
```

Close the IB Trader Workstation connection.

```
 close(ib)
```

Request Interactive Brokers Historical Data Within Regular Trading Hours

To request historical data, set up the IB Trader Workstation connection ib using ibtws. Create an IB Trader Workstation IContract object ibContract as shown in "Request Interactive Brokers Historical Data" on page 4-32. An IContract object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request the last 50 days of historical data using ib, ibContract, and these arguments:

- Start date is 50 days ago.
- End date is the current moment.
- The empty character vector denotes the default tick type 'TRADES'.
- Bar size is '1M'.
- Within regular trading hours.

```
startdate = floor(now)-50;
enddate = floor(now);
ticktype = '';
period = '1M';
tradehours = true;

d = history(ib,ibContract,startdate,enddate,ticktype,period,...
            tradehours)
d =

  Columns 1 through 5
```

```
    736267.00         661.18         730.00         641.73         710.81
    736298.00         711.21         762.71         705.85         742.60
    736312.00         747.11         775.96         724.17         748.73

Columns 6 through 9

    169656.00      125271.00         691.49              0
    210536.00      160260.00         734.41              0
    135075.00      102377.00         753.82              0
```

d returns the historical data for 50 days.

Each row of d contains historical data for 1 month.

The columns in matrix d are:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the low price for the most recent record in matrix d.

```
d(1,4)
```

```
ans =

    641.73
```

Close the IB Trader Workstation connection.

```
 close(ib)
```

Request Interactive Brokers Historical Data Using an Event Handler

To request historical data, set up the IB Trader Workstation connection ib using ibtws. Create an IB Trader Workstation IContract object ibContract as shown in "Request Interactive Brokers Historical Data" on page 4-32. An IContract object is an Interactive

Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request the last 50 days of historical data using `ib`, `ibContract`, and these arguments:

• Start date is 50 days ago.
• End date is the current moment.
• The empty character vector denotes the default tick type `'TRADES'`.
• Bar size is `'1M'`.
• Within regular trading hours.
• Sample event handler function `ibExampleEventHandler`.

Use `ibExampleEventHandler` or write a custom event handler function.

```
startdate = floor(now)-50;
enddate = floor(now);
ticktype = '';
period = '1M';
tradehours = true;
eventhandler = 'ibExampleEventHandler';

d = history(ib,ibContract,startdate,enddate,ticktype,period,...
            tradehours,eventhandler)

d =

     9157.00

 Columns 1 through 4

   [1x1 COM.TWS_TwsCtrl_1]   [22.00]   [9157.00]   '20151030'

 Columns 5 through 9

   [661.18]   [730.00]   [641.73]   [710.81]   [169656.00]

 Columns 10 through 14

   [125271.00]   [691.49]   [0]   [1x1 struct]   'historicalData'
...
```

`d` is the request identifier.

After `d`, `ibExampleEventHandler` streams historical data to the Command Window. The columns are:

• Interactive Brokers ActiveX object

- Event identifier
- Request identifier
- Date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar
- Structure that repeats the contents of the columns
- Event type

Close the IB Trader Workstation connection.

```
close(ib)
```

- "Create an Order Using IB Trader Workstation" on page 1-8
- "Create Interactive Brokers Combination Order" on page 4-39
- "Create and Manage an Interactive Brokers Order" on page 4-26
- "Request Interactive Brokers Historical Data" on page 4-32
- "Request Interactive Brokers Real-Time Data" on page 4-35

Input Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

`ibContract` — IB Trader Workstation contract
IContract object

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object. This object is the instrument or security used in the order transaction. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that you can set, see Interactive Brokers API Reference Guide.

startdate — Start date
date character vector | date scalar

Start date, specified as a starting date character vector or scalar.

Data Types: `double` | `char`

enddate — End date
date character vector | date scalar

End date, specified as an ending date character vector or scalar.

Data Types: `double` | `char`

ticktype — Types of market data ticks
`'TRADES'` (default) | `'MIDPOINT'` | `'BID'` | `'ASK'` | `'BID_ASK'` | `'HISTORICAL_VOLATILITY'` | `'OPTION_IMPLIED_VOLATILITY'`

Types of market data ticks, specified as one of the preceding values predetermined by the Interactive Brokers API that denote tick values to collect.

period — Bar size
`'1 day'` (default) | `'1W'` | `'1M'`

Bar size, specified as one of the preceding values predetermined by the Interactive Brokers API that denotes the periodicity for collecting data.

tradehours — Trading hours
false (default) | true

Trading hours, specified as the logical `true` or `false`. When this flag is set to `true`, this function returns data only within regular trading hours. Otherwise, this function returns all data.

Data Types: `logical`

eventhandler — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

Output Arguments

d — Interactive Brokers historical data
matrix | double

Interactive Brokers historical data, returned as a matrix with these columns:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

When using an event handler function, `d` is a double denoting the request identifier.

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

    - Connection
    - Information resulting from executing functions

- Errors

See Also

See Also
`close` | `createOrder` | `getdata` | `ibtws` | `realtime` | `timeseries`

Topics
"Create an Order Using IB Trader Workstation" on page 1-8
"Create Interactive Brokers Combination Order" on page 4-39
"Create and Manage an Interactive Brokers Order" on page 4-26
"Request Interactive Brokers Historical Data" on page 4-32
"Request Interactive Brokers Real-Time Data" on page 4-35
"Workflow for Interactive Brokers" on page 2-6
"Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites
Interactive Brokers API Reference Guide

Introduced in R2013b

timeseries

Request Interactive Brokers aggregated intraday data

Syntax

```
d = timeseries(ib,ibContract,startdate,enddate,barsize)
d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype)
d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype,
tradehours)
d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype,
tradehours,eventhandler)
```

Description

`d = timeseries(ib,ibContract,startdate,enddate,barsize)` requests Interactive Brokers aggregated intraday data using the IB Trader Workstation connection `ib` and IB Trader Workstation `IContract` object `ibContract` to signify the instrument. Request data between `startdate` and `enddate` using the tick aggregation interval `barsize` for default tick type `'TRADES'`.

`d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype)` requests Interactive Brokers aggregated intraday data for a specific type of market data tick `ticktype`.

`d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype, tradehours)` requests Interactive Brokers aggregated intraday data using the flag `tradehours` to include all data or only data within regular trading hours.

`d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype, tradehours,eventhandler)` requests Interactive Brokers aggregated intraday data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Request Interactive Brokers Intraday Data Aggregated Every 5 Minutes with TRADES Default Tick Type

To request intraday data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Real-Time Data" on page 4-35. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request intraday data aggregated every 5 minutes using `ib` and `ibContract`.

```
startdate = floor(now);
enddate = now;
barsize = '5 mins';

d = timeseries(ib,ibContract,startdate,enddate,barsize)

d =

    735329.40        6.91        6.91        6.85        6.85       158.00        13.00        6.87
    735329.40        6.85        6.87        6.85        6.87        29.00        24.00        6.86
    735329.40        6.87        6.89        6.87        6.87        13.00        13.00        6.88
...
```

`d` returns the aggregated 5-minute data with default tick type `'TRADES'`.

Each row in matrix `d` represents a 5-minute interval.

The columns in matrix `d` are:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the open price for the most recent bar in matrix `d`.

```
d(1,2)

ans =
    6.91
```

Close the IB Trader Workstation connection.

```
 close(ib)
```

Request Interactive Brokers Intraday Data Aggregated Every 10 Minutes with a BID Tick Type

To request intraday data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Real-Time Data" on page 4-35. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request intraday data aggregated every 10 minutes using `ib`, `ibContract`, and `'BID'` tick type.

```
startdate = floor(now);
enddate = now;
barsize = '10 mins';
ticktype = 'BID';

d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype)

d =

    735329.17        6.38        6.38        6.38        6.38       -1.00       -1.00       -1.00
    735329.17        6.38        6.38        6.38        6.38       -1.00       -1.00       -1.00
    735329.18        6.38        6.38        6.38        6.38       -1.00       -1.00       -1.00
...
```

`d` returns the aggregated 10-minute data for `'BID'` tick type.

Each row in matrix `d` represents a 10-minute interval.

The columns in matrix `d` are:

- Numeric representation of a date
- Open price

- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the high price for the most recent bar in matrix `d`.

```
d(1,3)
```

```
ans =
    6.38
```

Close the IB Trader Workstation connection.

```
close(ib)
```

Request Interactive Brokers Intraday Data Within Regular Trading Hours

To request intraday data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Real-Time Data" on page 4-35. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request intraday data using `ib`, `ibContract`, and these arguments:

- Start date is this morning.
- End date is the current moment.
- Aggregated every 10 minutes.
- Tick type is `'BID'`.
- Within regular trading hours.

```
startdate = floor(now);
enddate = now;
barsize = '10 mins';
```

```
ticktype = 'BID';
tradehours = true;

d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype,...
               tradehours)

d =

  Columns 1 through 5

     735852.40        580.70        582.12        580.12        580.27
     735852.40        580.27        580.75        579.70        579.80
     735852.40        579.80        579.88        578.33        579.44
     ...

  Columns 6 through 9

        -1.00         -1.00         -1.00             0
        -1.00         -1.00         -1.00             0
        -1.00         -1.00         -1.00             0
        ...
```

d returns the aggregated 10-minute data for 'BID' tick type.

Each row in matrix d represents a 10-minute interval.

The columns in matrix d are:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the high price for the most recent bar in matrix d.

```
d(1,3)

ans =
     582.12
```

Close the IB Trader Workstation connection.

```
close(ib)
```

Request Interactive Brokers Intraday Data Using an Event Handler

To request intraday data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Real-Time Data" on page 4-35. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request intraday data using `ib`, `ibContract`, and these arguments:

- Start date is this morning.
- End date is the current moment.
- Aggregated every 10 minutes.
- Tick type is `'BID'`.
- Within regular trading hours.
- Sample event handler function `ibExampleEventHandler`.

Use `ibExampleEventHandler` or write a custom event handler function.

```
startdate = floor(now);
enddate = now;
barsize = '10 mins';
ticktype = 'BID';
tradehours = true;
eventhandler = 'ibExampleEventHandler';

d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype,...
                tradehours,eventhandler)

d =

      4853.00

  Columns 1 through 3

    [1x1 COM.TWS_TwsCtrl_1]    [22.00]    [4853.00]

  Columns 4 through 7

    '20140909  15:55:00'    [580.20]    [581.40]    [580.09]

  Columns 8 through 13

    [581.01]    [-1.00]    [-1.00]    [-1.00]    [0]    [1x1 struct]
```

```
   Column 14

     'historicalData'
...
```

d is the request identifier.

After d, `ibExampleEventHandler` streams intraday data to the Command Window. The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Request identifier
- Date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar
- Structure that repeats the contents of the columns
- Event type

Close the IB Trader Workstation connection.

```
 close(ib)
```

- "Request Interactive Brokers Real-Time Data" on page 4-35

Input Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

`ibContract` — IB Trader Workstation contract
`IContract` object

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object. This object is the instrument or security used in the order transaction. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that you can set, see Interactive Brokers API Reference Guide.

`startdate` — Start date
date character vector | date scalar

Start date, specified as a starting date character vector or scalar.

Data Types: `double` | `char`

`enddate` — End date
date character vector | date scalar

End date, specified as an ending date character vector or scalar.

Data Types: `double` | `char`

`barsize` — Tick aggregation interval
`'10 secs'` | `'15 secs'` | `'30 secs'` | `'1 min'` | `'2 mins'` | `'3 mins'` | …

Tick aggregation interval, specified as one of the following values predetermined by the Interactive Brokers API that denotes the size of aggregated bars for collecting data.

- `'10 secs'`
- `'15 secs'`
- `'30 secs'`
- `'1 min'`
- `'2 mins'`
- `'3 mins'`
- `'5 mins'`
- `'10 mins'`
- `'15 mins'`
- `'20 mins'`

- `'30 mins'`
- `'1 hour'`
- `'2 hours'`
- `'3 hours'`
- `'4 hours'`
- `'8 hours'`

`ticktype` — Types of market data ticks
`'TRADES'` (default) | `'MIDPOINT'` | `'BID'` | `'ASK'` | `'BID_ASK'` |
`'HISTORICAL_VOLATILITY'` | `'OPTION_IMPLIED_VOLATILITY'`

Types of market data ticks, specified as one of the preceding values predetermined by the Interactive Brokers API that denote tick values to collect.

`tradehours` — Trading hours
false (default) | true

Trading hours, specified as the logical `true` or `false`. When this flag is set to `true`, this function returns data only within regular trading hours. Otherwise, this function returns all data.

Data Types: `logical`

`eventhandler` — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

Output Arguments

d — Interactive Brokers aggregated intraday data
matrix | double

Interactive Brokers aggregated intraday data, returned as a matrix with these columns:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

When using an event handler function, `d` is a double denoting the request identifier.

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

  - Connection
  - Information resulting from executing functions
  - Errors

See Also

See Also
`close` | `createOrder` | `getdata` | `history` | `ibtws` | `realtime`

Topics
"Request Interactive Brokers Real-Time Data" on page 4-35
"Workflow for Interactive Brokers" on page 2-6
"Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites

Interactive Brokers API Reference Guide

Introduced in R2013b

accounts

Retrieve Interactive Brokers account information

Syntax

```
d = accounts(ib,acctno)
d = accounts(ib,acctno,eventhandler)
```

Description

`d = accounts(ib,acctno)` retrieves account information using Interactive Brokers connection `ib` and account number `acctno`.

`d = accounts(ib,acctno,eventhandler)` retrieves account information using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Retrieve Account Information

Create the IB Trader Workstation℠ connection `ib` on the local machine using port number `7496`.

```
ib = ibtws('',7496);
```

Retrieve account information for account number `acctno` using `ib`.

```
acctno = 'AB123456';
```

```
d = accounts(ib,acctno);
```

`d` is a structure with fields containing the account information.

Display the account code.

```
d.AccountCode
```

```
ans =

  1×7 char array

DU15066
```

For details about this data and the other fields, see the Interactive Brokers® API Reference Guide.

Close the IB Trader Workstation℠ connection.

```
close(ib)
```

Retrieve Account Information Using an Event Handler

Create the IB Trader Workstation connection `ib` on the local machine using port number `7496`.

```
 ib = ibtws('',7496);
```

Retrieve account information for account number `acctno` using `ib`. Use the sample event handler `ibExampleEventHandler` to display the IB Trader Workstation account information in the Command Window. Use `ibExampleEventHandler` or write a custom event handler function.

```
acctno = 'AB123456';

d = accounts(ib,acctno,@ibExampleEventHandler)

d =

    []

  Columns 1 through 7

    [1x1 COM.TWS_TwsCtrl_1]    [7]    'AccountCode'    'AB123456'    ''    'AB123456'

  Column 8

    'updateAccountValue'
  ...
```

`d` is an empty double.

The sample event handler `ibExampleEventHandler` displays the account information in the Command Window. The columns are:

· Interactive Brokers ActiveX object

· Event identifier

· Account code

· Event key

· Currency

· Account name

· Structure that repeats the contents of the columns

· Request type

For details about this data, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

Input Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

`acctno` — Account number
character vector

Account number, specified as a character vector that identifies the Interactive Brokers account number.

Example:

Data Types: `char`

`eventhandler` — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

Output Arguments

d — Account information
structure | double

Account information, returned as a structure containing fields with the Interactive Brokers account information. When using an event handler function, `d` is an empty double.

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

  - Connection
  - Information resulting from executing functions
  - Errors

See Also

See Also
`close` | `createOrder` | `history` | `ibtws` | `timeseries`

Topics
"Create and Manage an Interactive Brokers Order" on page 4-26
"Workflow for Interactive Brokers" on page 2-6

"Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites

Interactive Brokers API Reference Guide

Introduced in R2015a

contractdetails

Request Interactive Brokers contract details

Syntax

```
[d,reqid] = contractdetails(ib,ibContract)
[d,reqid] = contractdetails(ib,ibContract,eventhandler)
```

Description

`[d,reqid] = contractdetails(ib,ibContract)` requests Interactive Brokers contract details using IB Trader Workstation connection `ib` and IB Trader Workstation `IContract` object `ibContract`.

`[d,reqid] = contractdetails(ib,ibContract,eventhandler)` requests Interactive Brokers contract details using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Request Interactive Brokers® Contract Details

Create the IB Trader Workstation℠ connection `ib` on the local machine using port number `7496`.

```
ib = ibtws('',7496);
```

Create the IB Trader Workstation℠ `IContract` object `ibContract`. This object describes a security with these property values:

- Google® symbol
- Stock security type
- Aggregate exchange
- Primary exchange

- USD currency

`IEX` is a sample primary exchange name. Substitute your primary exchange name for `ibContract.primaryExchange`.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'GOOG';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'IEX';
ibContract.currency = 'USD';
```

For details about the `IContract` object, see the Interactive Brokers® API Reference Guide.

Request contract details data using `ib` and `ibContract`.

```
[d,reqid] = contractdetails(ib,ibContract);
```

`d` is a structure containing the contract details data. For details about this data, see the Interactive Brokers® API Reference Guide.

`reqid` is a number that Interactive Brokers® uses to track this contract details data request.

Display the market name from the contract details data.

```
d.marketName
```

```
ans =

  1×3 char array

NMS
```

Display the request identifier.

```
reqid
```

```
reqid =

      8147
```

Close the IB Trader Workstation℠ connection.

```
close(ib)
```

Request Interactive Brokers Contract Details Using an Event Handler

Create the IB Trader Workstation connection `ib` on the local machine using port number `7496`.

```
 ib = ibtws('',7496);
```

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- Google symbol
- Stock security type
- Aggregate exchange
- Primary exchange
- USD currency

`EX` is a sample primary exchange name. Substitute your primary exchange name in `ibContract.primaryExchange`.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'GOOG';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'EX';
ibContract.currency = 'USD';
```

For details about the `IContract` object, see Interactive Brokers API Reference Guide.

Request contract details data using `ib`, `ibContract`, and sample event handler function `ibExampleEventHandler`. Use `ibExampleEventHandler` or write a custom event handler function.

```
[d,reqid] = contractdetails(ib,ibContract,@ibExampleEventHandler)

d =

       1269

reqid =
```

```
      1269

Columns 1 through 4

  [1x1 COM.TWS_TwsCtrl_1]    [100]    [1269]    [1x1 Interface.Tws_ActiveX_Control_m

Columns 5 through 6

  [1x1 struct]     'contractDetailsEx'
```

d and `reqid` return a number that Interactive Brokers uses to track this contract details data request.

After these variables, `ibExampleEventHandler` returns contract details data to the Command Window. The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Request identifier
- Contract details ActiveX object
- Structure that repeats the contents of the columns
- Request type

For details about this data, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

Input Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

`ibContract` — IB Trader Workstation contract
IContract object

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object. This object is the instrument or security used in the order transaction. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that you can set, see Interactive Brokers API Reference Guide.

`eventhandler` — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

Output Arguments

`d` — Interactive Brokers contract details data
structure | scalar

Interactive Brokers contract details data, returned as a structure. When using an event handler function, `d` is a scalar that denotes the contract detail data request identifier.

`reqid` — Contract detail data request identifier
scalar

Contract detail data request identifier, returned as a scalar. Interactive Brokers uses this number to match responses to the correct data request when multiple data requests are present.

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

  - Connection

- Information resulting from executing functions
- Errors

See Also

See Also
close | createOrder | history | ibtws | timeseries

Topics
"Create and Manage an Interactive Brokers Order" on page 4-26
"Workflow for Interactive Brokers" on page 2-6
"Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites
Interactive Brokers API Reference Guide

Introduced in R2015a

executions

Request Interactive Brokers execution data

Syntax

```
d = executions(ib,filter)
d = executions(ib,filter,eventhandler)
```

Description

`d = executions(ib,filter)` requests Interactive Brokers execution data using the IB Trader Workstation connection `ib` and the Interactive Brokers execution filter `filter`.

`d = executions(ib,filter,eventhandler)` requests Interactive Brokers execution data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Request Execution Filter Data

Create the IB Trader Workstation℠ connection `ib` on the local machine using port number `7496`.

```
ib = ibtws('',7496);
```

Create the IB Trader Workstation℠ execution filter `IExecutionFilter` object `filter`. This object specifies these property values:

- Buy side
- Stock security type
- Aggregate exchange
- Google® symbol

```
filter = ib.Handle.createExecutionFilter;
filter.side = 'BUY';
filter.secType = 'STK';
filter.exchange = 'SMART';
filter.symbol = 'GOOG';
```

For details about the `IExecutionFilter` object, see the Interactive Brokers® API Reference Guide.

Request IB Trader Workstation℠ execution filter data using `ib` and `filter`.

```
d = executions(ib,filter)

d =

  struct with fields:

    enddetails: [1×1 struct]
```

`d` is a structure containing the execution filter data in the structure `enddetails`.

Display the execution filter data.

```
d.enddetails

ans =

  struct with fields:

       Type: 'execDetailsEnd'
     Source: [1×1 COM.TWS_TwsCtrl_1]
    EventID: 38
      reqId: 1
```

The structure `enddetails` contains these fields:

• Type -- Data request type
• Source -- Interactive Brokers® ActiveX® object
• EventID -- Event identifier
• reqId -- Execution filter data request identifier

Close the IB Trader Workstation℠ connection.

```
close(ib)
```

Request Execution Filter Data Using an Event Handler

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

```
 ib = ibtws('',7496);
```

Create the IB Trader Workstation execution filter `IExecutionFilter` object `filter`. Here, this object specifies these property values:

- Buy side
- Stock security type
- Aggregate exchange
- Google symbol

```
filter = ib.Handle.createExecutionFilter;
filter.side = 'BUY';
filter.secType = 'STK';
filter.exchange = 'SMART';
filter.symbol = 'GOOG';
```

For details about the `IExecutionFilter` object, see Interactive Brokers API Reference Guide.

Request IB Trader Workstation execution filter data using `ib` and `filter`. Use the sample event handler `ibExampleEventHandler` to display the IB Trader Workstation execution filter data in the Command Window. Use `ibExampleEventHandler` or write a custom event handler function.

```
d = executions(ib,filter,@ibExampleEventHandler)

d =

    []

    [1x1 COM.TWS_TwsCtrl_1]    [38]    [1]    [1x1 struct]    'execDetailsEnd'
```

`d` is an empty double.

`ibExampleEventHandler` displays the data in the Command Window. The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Execution filter data request identifier
- Structure that repeats the contents of the columns
- Data request type

For details, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

- "Create and Manage an Interactive Brokers Order" on page 4-26

Input Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

`filter` — IB Trader Workstation execution filter
`IExecutionFilter` object

IB Trader Workstation execution filter, specified as a `IExecutionFilter` object. For details about this object, see Interactive Brokers API Reference Guide.

Example:

Data Types: `struct`

`eventhandler` — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

Output Arguments

d — IB Trader Workstation execution filter data
structure | double

IB Trader Workstation execution filter data, returned as a structure. When using an event handler function, `d` is an empty double.

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

  - Connection
  - Information resulting from executing functions
  - Errors

See Also

See Also
`close` | `createOrder` | `getdata` | `history` | `ibtws` | `timeseries`

Topics
"Create and Manage an Interactive Brokers Order" on page 4-26
"Workflow for Interactive Brokers" on page 2-6
"Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites
Interactive Brokers API Reference Guide

Introduced in R2015a

marketdepth

Request Interactive Brokers market depth data

Syntax

```
d = marketdepth(ib,ibContract,depth)
d = marketdepth(ib,ibContract,depth,eventhandler)
```

Description

`d = marketdepth(ib,ibContract,depth)` requests Interactive Brokers market depth data using the IB Trader Workstation connection `ib`, IB Trader Workstation `IContract` object `ibContract`, and price level `depth`.

`d = marketdepth(ib,ibContract,depth,eventhandler)` requests Interactive Brokers market depth data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Request Market Depth Data

To request Interactive Brokers market depth data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Real-Time Data" on page 4-35. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request market depth data using `ib` and `ibContract`. Specify five price levels for the bid and offer sides for `depth`. This code assumes `ibContract` is an E-mini S&P 500 futures contract with an expiry of December 2014 that trades on the CME Globex exchange.

```
depth = 5;
```

```
d = marketdepth(ib,ibContract,depth)

d =

     bid: [5x2 double]
   offer: [5x2 double]
```

d is a structure that contains the fields for bid and offer price levels.

Display the bid prices for five levels of market depth.

```
d.bid

ans =

      1992.5                      495
     1992.25                     1479
        1992                     1950
     1991.75                     1763
      1991.5                     2117
```

The first column contains the bid price and the second column contains the bid size.

Close the IB Trader Workstation connection.

```
 close(ib)
```

Request Market Depth Data Using an Event Handler

To request Interactive Brokers market depth data, set up the IB Trader Workstation connection ib using ibtws. Create an IB Trader Workstation IContract object ibContract as shown in "Request Interactive Brokers Real-Time Data" on page 4-35. An IContract object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request market depth data using ib and ibContract. Specify five price levels for the bid and offer sides for depth. This code assumes ibContract is an E-mini S&P 500 futures contract with an expiry of December 2014 that trades on the CME Globex exchange. Use the sample event handler function ibExampleEventHandler or write a custom event handler function.

```
depth = 5;
```

```
d = marketdepth(ib,ibContract,depth,@ibExampleEventHandler)

d =

      8147

  [1x1 COM.TWS_TwsCtrl_1]   [16.00]   [8147.00]   [0]   [0]   [1.00]   [1988.7
  ...
```

d is the request identifier.

After d, `ibExampleEventHandler` streams market depth data to the Command Window.

The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Request identifier
- Position
- Operation
- Side
- Price
- Size
- Structure that repeats the contents of the columns
- Event type

Close the IB Trader Workstation connection.

```
 close(ib)
```

- "Create and Manage an Interactive Brokers Order" on page 4-26

Input Arguments

ib — IB Trader Workstation connection
connection object

5-263

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

`ibContract` — IB Trader Workstation contract
`IContract` object

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object. This object is the instrument or security used in the order transaction. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that you can set, see Interactive Brokers API Reference Guide.

`depth` — IB Trader Workstation market depth
1 | 2 | 3 | ...

IB Trader Workstation market depth, specified as a scalar from one through 10. This number denotes the depth of the active book.

Data Types: `double`

`eventhandler` — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

Output Arguments

`d` — IB Trader Workstation market depth data
structure | double

IB Trader Workstation market depth data, returned as a structure containing the price level data for the bid and offer prices. Price level data consists of the price and size. When using an event handler function, `d` is a double denoting the request identifier.

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

    - Connection
    - Information resulting from executing functions
    - Errors

See Also

See Also

`close` | `createOrder` | `history` | `ibtws` | `realtime` | `timeseries`

Topics

"Create and Manage an Interactive Brokers Order" on page 4-26
"Workflow for Interactive Brokers" on page 2-6
"Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites

Interactive Brokers API Reference Guide

Introduced in R2015a

orderid

Obtain next valid order identification number

Syntax

```
id = orderid(ib)
```

Description

`id = orderid(ib)` obtains the next valid order identification number for Interactive Brokers connection `ib`.

Examples

Obtain Next Valid Order Identification Number

Create an IB Trader Workstation℠ connection on the local machine using port number 7496.

```
ib = ibtws('',7496);
```

Obtain the next valid order identification number using `ib`.

```
id = orderid(ib)
```

```
id =

    1
```

`id` contains the next valid order identification number. To create an order, use this number in createOrder.

Close the IB Trader Workstation℠ connection.

```
close(ib)
```

- "Create and Manage an Interactive Brokers Order" on page 4-26

Input Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

Output Arguments

`id` — Next valid order identification number
scalar

Next valid order identification number, returned as a scalar.

Tips

*   `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

    *   Connection
    *   Information resulting from executing functions
    *   Errors

See Also

See Also
`close` | `createOrder` | `getdata` | `history` | `ibtws` | `timeseries`

Topics
"Create and Manage an Interactive Brokers Order" on page 4-26
"Workflow for Interactive Brokers" on page 2-6

External Websites

Interactive Brokers API Reference Guide

Introduced in R2015a

orders

Request Interactive Brokers open order data

Syntax

```
o = orders(ib)
o = orders(ib,client)
o = orders(ib,client,eventhandler)
```

Description

`o = orders(ib)` requests Interactive Brokers open order data using IB Trader Workstation connection `ib` for the current client only.

`o = orders(ib,client)` requests Interactive Brokers open order data using IB Trader Workstation connection `ib` and a client flag. `client` denotes requesting data from the current client or all clients.

`o = orders(ib,client,eventhandler)` requests Interactive Brokers open order data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Request Open Order Data

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

```
ib = ibtws('',7496);
```

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- Google symbol
- Stock security type

5-269

- Aggregate exchange
- Primary exchange
- USD currency

EX is a sample primary exchange name. Substitute your primary exchange name in `ibContract.primaryExchange`.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'GOOG';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'EX';
ibContract.currency = 'USD';
```

Create the IB Trader Workstation `IOrder` object `ibOrder`. Here, this object describes a limit order to sell two shares with a limit price of $590.

```
ibOrder = ib.Handle.createOrder;
ibOrder.action = 'SELL';
ibOrder.totalQuantity = 2;
ibOrder.orderType = 'LMT'
ibOrder.lmtPrice = 590;
```

For details about the `IContract` and `IOrder` objects, see Interactive Brokers API Reference Guide.

Create a unique order identifier `id`.

```
id = orderid(ib);
```

Execute the order using:

- IB Trader Workstation connection `ib`
- IB Trader Workstation `IContract` object `ibContract`
- IB Trader Workstation `IOrder` object `ibOrder`
- Unique order identifier `id`

```
d = createOrder(ib,ibContract,ibOrder,id);
```

Retrieve order information `o`.

```
o = orders(ib)

o =
```

```
1x2 struct array with fields:

    Type
    EventID
    orderId
    contract
    order
    orderState
```

o contains a structure array. The array contains a structure with data for each open order. The structure fields are:

- Order type
- Event identifier
- Order identifier
- Contract data
- Order data
- Order status

Retrieve the current status of the order.

```
o.orderState
```

```
ans =

            status: 'Submitted'
        initMargin: '1.7976931348623157E308'
       maintMargin: '1.7976931348623157E308'
        ...
```

orderState is a structure with fields corresponding to the status of the order. The fields are order status, initial margin, and maintenance margin. For details on these fields and the additional fields in orderState, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

Request Open Order Data From All Clients

Create the IB Trader Workstation connection ib on the local machine using port number 7496.

```
 ib = ibtws('',7496);
```

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- Google symbol
- Stock security type
- Aggregate exchange
- Primary exchange
- USD currency

`EX` is a sample primary exchange name. Substitute your primary exchange name in `ibContract.primaryExchange`.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'GOOG';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'EX';
ibContract.currency = 'USD';
```

Create the IB Trader Workstation `IOrder` object `ibOrder`. Here, this object describes a limit order to sell two shares with a limit price of $590.

```
ibOrder = ib.Handle.createOrder;
ibOrder.action = 'SELL';
ibOrder.totalQuantity = 2;
ibOrder.orderType = 'LMT'
ibOrder.lmtPrice = 590;
```

For details about the `IContract` and `IOrder` objects, see Interactive Brokers API Reference Guide.

Create a unique order identifier `id`.

```
id = orderid(ib);
```

Execute the order using:

- IB Trader Workstation connection `ib`
- IB Trader Workstation `IContract` object `ibContract`
- IB Trader Workstation `IOrder` object `ibOrder`

- Unique order identifier `id`

```
d = createOrder(ib,ibContract,ibOrder,id);
```

Retrieve order information `o` from all clients by setting `client` to `false`.

```
o = orders(ib,false)

o =

1x2 struct array with fields:

    Type
    EventID
    orderId
    contract
    order
    orderState
```

`o` contains a structure array. The array contains a structure with data for each open order. The structure fields are:

- Order type
- Event identifier
- Order identifier
- Contract data
- Order data
- Order status

Retrieve the current status of the order.

```
o.orderState

ans =

            status: 'Submitted'
        initMargin: '1.7976931348623157E308'
       maintMargin: '1.7976931348623157E308'
        ...
```

`orderState` is a structure with fields corresponding to the status of the order. The fields are order status, initial margin, and maintenance margin. For details on these fields and the additional fields in `orderState`, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

Request Open Order Data Using an Event Handler

Create the IB Trader Workstation connection `ib` on the local machine using port number `7496`.

```
 ib = ibtws('',7496);
```

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- Google symbol
- Stock security type
- Aggregate exchange
- Primary exchange
- USD currency

`EX` is a sample primary exchange name. Substitute your primary exchange name in `ibContract.primaryExchange`.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'GOOG';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.primaryExchange = 'EX';
ibContract.currency = 'USD';
```

Create the IB Trader Workstation `IOrder` object `ibOrder`. Here, this object describes a limit order to sell two shares with a limit price of $590.

```
ibOrder = ib.Handle.createOrder;
ibOrder.action = 'SELL';
ibOrder.totalQuantity = 2;
ibOrder.orderType = 'LMT'
ibOrder.lmtPrice = 590;
```

For details about the `IContract` and `IOrder` objects, see Interactive Brokers API Reference Guide.

Create a unique order identifier `id`.

```
id = orderid(ib);
```

Execute the order using:

- IB Trader Workstation connection `ib`
- IB Trader Workstation `IContract` object `ibContract`
- IB Trader Workstation `IOrder` object `ibOrder`
- Unique order identifier `id`

```
d = createOrder(ib,ibContract,ibOrder,id);
```

Retrieve order information from all clients by setting `client` to `false` and using the sample event handler function `ibExampleEventHandler`. Use `ibExampleEventHandler` or write a custom event handler function.

```
o = orders(ib,false,@ibExampleEventHandler)

o =

    []

  Columns 1 through 4

    [1x1 COM.TWS_TwsCtrl_1]    [101]    [56947638]    [1x1 Interface.Tws_ActiveX_Contro

  Columns 5 through 6

    [1x1 Interface.Tws_ActiveX_Control_module.IOrder]    [1x1 Interface.Tws_ActiveX_Cor

  Columns 7 through 8

    [1x1 struct]    'openOrderEx'
```

`o` contains an empty double because the event handler `ibExampleEventHandler` processes the output data.

`ibExampleEventHandler` displays the output data in the Command Window. Here, IB Trader Workstation returns:

- Interactive Brokers ActiveX object
- Event identifier
- Unique order identifier

- IB Trader Workstation `IContract` object
- IB Trader Workstation `IOrder` object
- IB Trader Workstation `IOrderState` object
- Structure that repeats the contents of the columns
- Request type

For details about this data, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

- "Create and Manage an Interactive Brokers Order" on page 4-26

Input Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

`client` — Client flag
`true` (default) | `false`

Client flag, specified as a logical. `true` denotes returning data from the current client only. `false` denotes returning data from all clients.

Data Types: `logical`

`eventhandler` — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

Output Arguments

o — Interactive Brokers open order data
structure | double

Interactive Brokers open order data, returned as a structure or an empty double. The structure contains these fields:

- Order type
- Event identifier
- Order identifier
- Contract data
- Order data
- Order status

When using an event handler function, `o` is an empty double.

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

  - Connection
  - Information resulting from executing functions
  - Errors

- Executing `orders` multiple times using the same IB Trader Workstation connection can cause this kind of warning message: Warning: Cannot unregister 'openOrderEx'. Invalid event name/handler combination. To fix this warning, close the IB Trader Workstation connection and create a new connection using `ibtws`.

See Also

See Also
`close` | `createOrder` | `executions` | `getdata` | `history` | `ibtws` | `orderid` | `timeseries`

Topics

External Websites
Interactive Brokers API Reference Guide

Introduced in R2015a

portfolio

Retrieve current Interactive Brokers portfolio data

Syntax

```
p = portfolio(ib)
p = portfolio(ib,acctno)
p = portfolio(ib,acctno,eventhandler)
```

Description

`p = portfolio(ib)` retrieves current Interactive Brokers portfolio data for the active account number using the IB Trader Workstation connection `ib`.

`p = portfolio(ib,acctno)` retrieves current Interactive Brokers portfolio data using the account number `acctno`.

`p = portfolio(ib,acctno,eventhandler)` retrieves current Interactive Brokers portfolio data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Retrieve Current Portfolio Data

Create the IB Trader Workstation connection `ib` on the local machine using port number `7496`.

```
 ib = ibtws('',7496);
```

Retrieve current Interactive Brokers portfolio data using `ib`.

```
p = portfolio(ib)

p =
```

```
         Type: {5x1 cell}
       Source: {5x1 cell}
      EventID: {5x1 cell}
     contract: {5x1 cell}
     position: {5x1 cell}
  marketPrice: {5x1 cell}
  marketValue: {5x1 cell}
  averageCost: {5x1 cell}
unrealizedPNL: {5x1 cell}
  realizedPNL: {5x1 cell}
  accountName: {5x1 cell}
```

p is a structure that contains these fields:

- Event type
- Interactive Brokers ActiveX object
- Event identifier
- Contract details
- Number of shares for each contract
- Price of the shares for each contract
- Number of shares multiplied by the price of the shares for each contract
- Average price when the shares are purchased for each contract
- Unrealized profit and loss for each contract
- Actual profit and loss for each contract
- Account number

5x1 means there are five contracts in this portfolio. For details about this data, see Interactive Brokers API Reference Guide.

Display the market price for each contract in the portfolio.

```
p.marketPrice
```

```
ans =

    [  8.60]
    [582.95]
    [591.79]
    [188.44]
    [ 42.24]
```

Close the IB Trader Workstation connection.

```
close(ib)
```

Retrieve Current Portfolio Data Using the Account Number

Create the IB Trader Workstation connection `ib` on the local machine using port number `7496`.

```
 ib = ibtws('',7496);
```

Retrieve current Interactive Brokers portfolio data using `ib` and account number `acctno`.

```
acctno = 'DU111111';

p = portfolio(ib,acctno)

p =

             Type: {5x1 cell}
           Source: {5x1 cell}
          EventID: {5x1 cell}
         contract: {5x1 cell}
         position: {5x1 cell}
      marketPrice: {5x1 cell}
      marketValue: {5x1 cell}
      averageCost: {5x1 cell}
    unrealizedPNL: {5x1 cell}
      realizedPNL: {5x1 cell}
      accountName: {5x1 cell}
```

`p` is a structure that contains these fields:

- Event type
- Interactive Brokers ActiveX object
- Event identifier
- Contract details
- Number of shares for each contract
- Price of the shares for each contract
- Number of shares multiplied by the price of the shares for each contract
- Average price when the shares are purchased for each contract

- Unrealized profit and loss for each contract
- Actual profit and loss for each contract
- Account number

`5x1` means there are five contracts in this portfolio. For details about this data, see Interactive Brokers API Reference Guide.

Display the market price for each contract in the portfolio.

```
p.marketPrice

ans =

    [  8.60]
    [582.95]
    [591.79]
    [188.44]
    [ 42.24]
```

Close the IB Trader Workstation connection.

```
close(ib)
```

Retrieve Current Portfolio Data Using an Event Handler

Create the IB Trader Workstation connection `ib` on the local machine using port number `7496`.

```
 ib = ibtws('',7496);
```

Retrieve current Interactive Brokers portfolio data using `ib`, account number `acctno`, and sample event handler `ibExampleEventHandler`. Use `ibExampleEventHandler` or write a custom event handler function.

```
acctno = 'DU111111';

p = portfolio(ib,acctno,@ibExampleEventHandler)

p =

    []

  Columns 1 through 5
```

```
   [1x1 COM.TWS_TwsCtrl_1]    [103]    [1x1 Interface.Tws_ActiveX_Control_module.ICont

  Columns 6 through 12

   [515.10]    [8.22]    [21.68]    [0]    'DU111111'    [1x1 struct]    'updatePortfo
  ...
```

p is an empty double because `ibExampleEventHandler` displays the current Interactive Brokers portfolio data for each security in the Command Window.

The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- IB Trader Workstation `IContract` object
- Number of shares
- Price of the shares
- Number of shares multiplied by the price of the shares
- Average price when the shares are purchased
- Unrealized profit and loss
- Actual profit and loss
- Account number
- Structure that repeats the contents of the columns
- Event type

For details about this data, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

- "Create and Manage an Interactive Brokers Order" on page 4-26

Input Arguments

ib — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using ibtws.

acctno — Account number
character vector

Account number, specified as a character vector that identifies the Interactive Brokers account number.

Example:

Data Types: char

eventhandler — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: @eventhandler

Data Types: function_handle | char

Output Arguments

p — Interactive Brokers portfolio data
structure | double

Interactive Brokers portfolio data, returned as a structure. The structure contains these fields. When using an event handler function, p is an empty double.

Field	Description
Type	Interactive Brokers event type name
Source	Interactive Brokers ActiveX object
EventID	Number that identifies the event type
contract	Structure that contains details for each contract in the portfolio

Field	Description
position	Number of shares for each contract in the portfolio
marketPrice	Price of the shares for each contract in the portfolio
marketValue	Number of shares multiplied by the price of the shares for each contract in the portfolio
averageCost	Average price when the shares are purchased for each contract in the portfolio
unrealizedPNL	Unrealized profit and loss for each contract in the portfolio
realizedPNL	Actual profit and loss for each contract in the portfolio
accountName	Account number

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

  - Connection
  - Information resulting from executing functions
  - Errors

See Also

See Also
`close` | `createOrder` | `executions` | `getdata` | `history` | `ibtws` | `marketdepth` | `timeseries`

Topics
"Create and Manage an Interactive Brokers Order" on page 4-26

"Workflow for Interactive Brokers" on page 2-6
"Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites

Interactive Brokers API Reference Guide

Introduced in R2015a

realtime

Request Interactive Brokers real-time data

Syntax

```
tickerid = realtime(ib,ibContract,f)
tickerid = realtime(ib,ibContract,f,eventhandler)
```

Description

`tickerid = realtime(ib,ibContract,f)` requests Interactive Brokers real-time data using IB Trader Workstation connection `ib`, IB Trader Workstation `IContract` object `ibContract`, and Interactive Brokers fields `f`.

`tickerid = realtime(ib,ibContract,f,eventhandler)` requests Interactive Brokers real-time data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

Examples

Request Real-Time Data

To request real-time data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Real-Time Data" on page 4-35. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Set the Interactive Brokers field `f` to `233` to denote the tick type for RTVolume. RTVolume contains these fields:

- Last trade price
- Last trade size

- Last trade time
- Total volume
- Volume weighted average price (VWAP)
- Single trade flag

For details about RTVolume, see Interactive Brokers API Reference Guide.

Request real-time data using `ib`, `ibContract`, and `f`.

```
f = '233';

tickerid = realtime(ib,ibContract,f)

tickerid =

    1
```

`tickerid` returns a number for tracking the real-time data request.

The real-time data is returned in the MATLAB workspace variable `ibBuiltInRealtimeData`.

Display this real-time data.

```
ibBuiltInRealtimeData

ibBuiltInRealtimeData =

         id: 1
  BID_PRICE: 584.65
   BID_SIZE: 1
  ASK_PRICE: 585.80
   ASK_SIZE: 1
 LAST_PRICE: 585
  LAST_SIZE: 1
     VOLUME: 11611
```

The structure `ibBuiltInRealtimeData` contains these fields:

- Real-time request identifier
- Bid price
- Bid size

- Ask price
- Ask size
- Last price
- Last size
- Volume

The `id` field is a number that tracks the real-time data request for IB Trader Workstation `IContract` object `ibContract`. When you create multiple contracts, each real-time data display has a different value for the `id` field that corresponds to a specific contract.

Cancel the real-time market data request using `tickerid`.

```
ib.Handle.cancelMktData(tickerid)
```

Remove event handler assignments for each real-time event.

```
evtListeners = ib.Handle.eventlisteners;
eventTypes = {'tickSize','tickPrice','tickSnapshotEnd', ...
    'tickOptionComputation','tickGeneric','tickString', ...
    'tickEFP','marketDataType'};
for j = 1:length(eventTypes)
    i = strcmp(evtListeners(:,1),eventTypes{j});
    ib.Handle.unregisterevent([{evtListeners{i,1}}' {evtListeners{i,2}}']);
end
```

Close the IB Trader Workstation connection.

```
close(ib)
```

Request Real-Time Data Using an Event Handler

To request real-time data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Real-Time Data" on page 4-35. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Set the field `f` to the tick type for RTVolume `233`. RTVolume contains:

- Last trade price

- Last trade size
- Last trade time
- Total volume
- Volume weighted average price (VWAP)
- Single trade flag

For details about RTVolume, see Interactive Brokers API Reference Guide.

```
f = '233';
```

Request real-time data using `ib`, `ibContract`, and `f`. Use the sample event handler `ibExampleEventHandler` to display the real-time data in the Command Window.

```
tickerid = realtime(ib,ibContract,f,...
                    @ibExampleEventHandler)

tickerid =

     1

    [1x1 COM.TWS_TwsCtrl_1]    [1]    [1]    [1]    [585.50]    [1]    [1x1 struct]

    [1x1 COM.TWS_TwsCtrl_1]    [2]    [1]    [0]    [1]    [1x1 struct]    'tickSize'

...
```

`tickerid` returns a number for tracking the real-time data request.

After the `tickerid`, `ibExampleEventHandler` streams real-time data to the Command Window. Each line is a type of tick. Here, there is a price tick and size tick.

For a price tick, the IB Trader Workstation returns:

- Interactive Brokers ActiveX object
- Event identifier
- Request identifier
- Tick type
- Price
- Automatic execution flag
- Structure that repeats the contents of the columns

- Event type

For details about this data, see Interactive Brokers API Reference Guide.

Cancel the real-time market data request using `tickerid`.

```
ib.Handle.cancelMktData(tickerid)
```

Remove event handler assignments for each real-time event.

```
evtListeners = ib.Handle.eventlisteners;
eventTypes = {'tickSize','tickPrice','tickSnapshotEnd', ...
    'tickOptionComputation','tickGeneric','tickString', ...
    'tickEFP','marketDataType'};
for j = 1:length(eventTypes)
    i = strcmp(evtListeners(:,1),eventTypes{j});
    ib.Handle.unregisterevent([{evtListeners{i,1}}' {evtListeners{i,2}}']);
end
```

Close the IB Trader Workstation connection.

```
close(ib)
```

- "Request Interactive Brokers Real-Time Data" on page 4-35

Input Arguments

`ib` — IB Trader Workstation connection
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

`ibContract` — IB Trader Workstation contract
IContract object | cell array

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object or a cell array for multiple IB Trader Workstation `IContract` objects. This object is the instrument or security used in the order transaction. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that you can set, see Interactive Brokers API Reference Guide.

Data Types: `cell`

`f` — Interactive Brokers fields
character vector | cell array of character vectors

Interactive Brokers fields, specified as a character vector or a cell array of character vectors. These fields correspond to numeric identifiers that specify the Interactive Brokers generic market data tick types. For details, see Interactive Brokers API Reference Guide.

Data Types: `char` | `cell`

`eventhandler` — Event handler
function handle | character vector

Event handler, specified as a function handle or a character vector to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see "Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

Output Arguments

`tickerid` — Interactive Brokers market request identifier
double

Interactive Brokers market request identifier, specified as a double for tracking and canceling the market data request. `tickerid` is a scalar for one Interactive Brokers contract and a vector of scalars for multiple contracts.

Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:

  - Connection

- Information resulting from executing functions
- Errors

See Also

See Also
close | createOrder | history | ibtws | timeseries

Topics
"Request Interactive Brokers Real-Time Data" on page 4-35
"Workflow for Interactive Brokers" on page 2-6
"Writing and Running Custom Event Handler Functions with Interactive Brokers" on page 1-28

External Websites
Interactive Brokers API Reference Guide

Introduced in R2015a

fixflyer

FIX Flyer connection

Description

The `fixflyer` function creates a `fixflyer` object, which represents a FIX Flyer Engine connection. After you create a `fixflyer` object, you can send FIX messages and retrieve order status and information using the object functions.

Create Object

Syntax

Description

`c = fixflyer(username,password,ipaddress,port)` creates a connection `c` to the FIX Flyer Engine with user name, password, IP address, and port number.

`c = fixflyer(username,password,ipaddress,port,orderport)` also specifies a port number `orderport` for order information retrieval.

Input Arguments

`username` — FIX Flyer user name
character vector

FIX Flyer user name, specified as a character vector.

Example: `'guest'`

Data Types: `char`

`password` — FIX Flyer password
character vector

FIX Flyer password, specified as a character vector.

Data Types: char

`ipaddress` — IP address
character vector

IP address, specified as a character vector to indicate the IP address of the computer where the FIX Flyer Engine is running.

Example: `'127.0.0.1'`

Data Types: char

`port` — Port number
scalar

Port number, specified as a scalar to indicate the port number on the computer where the FIX Flyer Engine is running.

Example: `12001`

Data Types: double

`orderport` — Order information port number
scalar

Order information port number, specified as a scalar to indicate the port number on the computer where the FIX Flyer Engine is running. You can retrieve order information for active and closed orders by creating a FIX Flyer Engine connection using `orderport`.

Example: `13001`

Data Types: double

Properties

`User` — FIX Flyer user name
character vector

FIX Flyer user name, specified as a character vector.

Example: `'guest'`

Data Types: char

`Ipaddress` — IP address
character vector

IP address of the computer where the FIX Flyer Engine is running, specified as a character vector.

Example: `'127.0.0.1'`

Data Types: `char`

`FIXPort` — Port number
double

Port number of the computer where the FIX Flyer Engine is running, specified as a double.

Example: `12001`

Data Types: `double`

`RestPort` — Order information port number
double

Order information port number of the computer where the FIX Flyer Engine is running, specified as a double. This property appears only when you run `fixflyer` with the `orderport` input argument.

Example: `13001`

Data Types: `double`

`FlyerApplicationManager` — FIX Flyer application
FIX Flyer application manager object

FIX Flyer application, specified as a FIX Flyer application manager object.

`SessionID` — FIX Flyer session identifier
double

FIX Flyer session identifier, specified as a double.

Data Types: `double`

Object Functions

close   Close FIX Flyer connection

sendMessage	Send FIX message to FIX Flyer Engine
orderInfo	Retrieve FIX Flyer order status and information
addListener	Add event handling listener to FIX Flyer connection

Examples

Create FIX Flyer Engine Connection

Note: To create a FIX Flyer connection for the first time, add the JAR file `fix-flyer.jar` to the static Java class path. For details, see "Installation" on page 1-3.

Import the FIX Flyer Java libraries.

```
import flyer.apps.*;
import flyer.apps.FlyerApplicationManagerFactory.*;
import flyer.core.session.*;
```

Use these arguments to create the FIX Flyer Engine connection and display the connection properties:

- User name `username`
- Password `password`
- IP address `ipaddress`
- Port number `port`

```
username = 'user';
password = 'pwd';
ipaddress = '127.0.0.1';
port = 7002;

c = fixflyer(username,password,ipaddress,port)

c =

  fixflyer with properties:

                     User: 'user'
```

```
                    Ipaddress: '127.0.0.1'
                      FIXPort: 7002.00
                     RestPort: []
    FlyerApplicationManager: [1x1 flyer.apps.FlyerApplicationManager]
                    SessionID: []
```

`c` is the FIX Flyer Engine connection object with these properties:

- User name
- IP address
- Port number
- FIX Flyer application manager instance
- FIX Flyer session identifier

Close the FIX Flyer connection.

```
close(c)
```

Create FIX Flyer Engine Connection for Order Information Retrieval

Note: To create a FIX Flyer connection for the first time, add the JAR file `fix-flyer.jar` to the static Java class path. For details, see "Installation" on page 1-3.

Import the FIX Flyer Java libraries.

```
import flyer.apps.*;
import flyer.apps.FlyerApplicationManagerFactory.*;
import flyer.core.session.*;
```

Use these arguments to create the FIX Flyer Engine connection and display the connection properties:

- User name `username`
- Password `password`
- IP address `ipaddress`
- Port number `port`
- Order information port number `orderport`

```
username = 'guest';
```

```
password = 'guest';
ipaddress = 'example.fixcomputeserver.com';
port = 12001;
orderport = 13001;

c = fixflyer(username,password,ipaddress,port,orderport)

c =

  fixflyer with properties:

                        User: 'guest'
                   Ipaddress: 'example.fixcomputeserver.com'
                     FIXPort: 12001
                    RestPort: 13001
    FlyerApplicationManager: [1x1 flyer.apps.FlyerApplicationManager]
                   SessionID: []
```

c is the FIX Flyer Engine connection object with these properties:

- User name
- IP address
- Port number
- Order information port number
- FIX Flyer application manager instance
- FIX Flyer session identifier

To retrieve order information for active and closed orders, see `orderInfo`.

Close the FIX Flyer connection.

```
close(c)
```

- "Create an Order Using FIX Flyer" on page 1-20

See Also

Topics
"Create an Order Using FIX Flyer" on page 1-20

External Websites

FIX Flyer Download Portal
FIX Trading Community

Introduced in R2015b

addListener

Add event handling listener to FIX Flyer connection

Syntax

```
lh = addListener(c,listener)
```

Description

`lh = addListener(c,listener)` adds the event handling listener `listener` to the FIX Flyer Engine connection `c`. Use the sample event handling listener `fixExampleListener` or write a custom event handling listener function.

Examples

Listen for FIX Flyer Event Data

First, create a FIX Flyer Engine connection. Then, add a FIX Flyer event listener to the FIX Flyer Engine connection, and listen for and display the event data in the Workspace browser.

Create the FIX Flyer Engine connection `c` using these arguments:

- User name `username`
- Password `password`
- IP address `ipaddress`
- Port number `port`

```
username = 'user';
password = 'pwd';
ipaddress = '127.0.0.1';
port = 7002;

c = fixflyer(username,password,ipaddress,port);
```

Add the FIX Flyer event listener to the FIX Flyer Engine connection. Use the sample event handling listener `fixExampleListener` to listen for and display the FIX Flyer

Engine event data in the Workspace browser. To access the code for the listener, enter `edit fixExampleListener.m`. Or, to process the event data in another way, you can write a custom event handling listener function. For details, see "Create Functions in Files" (MATLAB).

Process the FIX Flyer Engine events `e` using the sample event handling listener `fixExampleListener`. Specify `e` as any letter. `fixExampleListener` returns a handle to the listener `lh`.

```
lh = addListener(c,@(~,e)fixExampleListener(e,c));
```

When events occur, `fixExampleListener` returns event data to objects in the MATLAB Workspace. To view event data, double-click the object. The Variables dialog box displays the data in the object.

Close the FIX Flyer Engine connection.

```
close(c)
```

Input Arguments

c — FIX Flyer Engine connection
fixflyer object

FIX Flyer Engine connection, specified as a `fixflyer` object.

listener — Listener event handler
function handle

Listener event handler, specified as a function handle to listen for FIX Flyer Engine event data. You can modify the existing listener function or define your own. The code for the existing listener function is in the `fixExampleListener.m` file.

Data Types: `function_handle`

Output Arguments

lh — Listener handle
object handle

Listener handle, returned as a handle to a FIX Flyer listener object.

See Also

See Also
`close` | `fixflyer` | `sendMessage`

Topics
"Create an Order Using FIX Flyer" on page 1-20

Introduced in R2015b

sendMessage

Send FIX message to FIX Flyer Engine

Syntax

```
status = sendMessage(c,fixmsg)
```

Description

`status = sendMessage(c,fixmsg)` sends the FIX message `fixmsg` using the FIX Flyer Engine connection `c`.

Examples

Send FIX Message

First, create a FIX Flyer Engine connection. Then, add a FIX Flyer event listener to the FIX Flyer Engine connection. Subscribe to FIX sessions. Create and send two FIX messages.

Create the FIX Flyer Engine connection `c` using these arguments:

- User name `username`
- Password `password`
- IP address `ipaddress`
- Port number `port`

```
username = 'user';
password = 'pwd';
ipaddress = '127.0.0.1';
port = 7002;

c = fixflyer(username,password,ipaddress,port);
```

Add the FIX Flyer event listener to the FIX Flyer Engine connection. Use the sample event handling listener `fixExampleListener` to listen for and display the FIX Flyer Engine event data in the Workspace browser. To access the code for the listener, enter

edit `fixExampleListener.m`. Or, to process the event data in another way, you can write a custom event handling listener function. For details, see "Create Functions in Files" (MATLAB).

Process the FIX Flyer Engine events `e` using the sample event handling listener `fixExampleListener`. Specify `e` as any letter. `fixExampleListener` returns a handle to the listener `lh`.

```
lh = addListener(c,@(~,e)fixExampleListener(e,c));
```

Subscribe to FIX sessions and set up the FIX Flyer Application Manager. Register with the FIX Flyer session. Connect the FIX Flyer Application Manager to the FIX Flyer Engine and start the internal receiving thread.

```
c.SessionID = flyer.core.session.SessionID('Alpha',...
                                           'Beta','FIX.4.4');
c.FlyerApplicationManager.setLoadDefaultDataDictionary(false);
c.FlyerApplicationManager.registerFIXSession(...
                                  flyer.apps.FixSessionSubscription(...
                                  c.SessionID,true,0));
c.FlyerApplicationManager.connect;
c.FlyerApplicationManager.start;
```

Create a FIX message using table `fixtable`. This table contains two FIX messages. The first row in the table represents a sell side transaction for 100 shares of symbol ABC. The order type is a previously quoted order. The order handling instruction is a private automated execution. The order transaction time is the current moment. The second row in the table has the same order field variables, except that the order identifier is unique across orders. The FIX protocol version is 4.4.

```
fixtable = table({'FIX.4.4';'FIX.4.4'},...
                 {'338';'339'},{'2';'2'},...
                 {datestr(now);datestr(now)},...
                 {'D';'D'},{'ABC';'ABC'},...
                 {'1';'1'},{'D';'D'},{'100';'100'},...
                 'VariableNames',{'BeginString' ...
                 'CLOrdId' 'Side' 'TransactTime' ...
                 'OrdType' 'Symbol' ...
                 'HandlInst' 'MsgType' 'OrderQty'});
```

Send the FIX message using the FIX message `fixtable`.

`status` contains the FIX Flyer Engine message status for each FIX message sent. If the FIX message is sent successfully, `status` contains a logical zero. `status` has an entry for each FIX message in `fixtable`.

```
status = sendMessage(c,fixtable)
```

```
status =

     0
     0
```

The MATLAB Workspace variable `fixResponseStruct` contains the returned FIX messages from the FIX Flyer Engine.

Close the FIX Flyer Engine connection.

```
close(c)
```

- "Create an Order Using FIX Flyer" on page 1-20

Input Arguments

`c` — FIX Flyer Engine connection
`fixflyer` object

FIX Flyer Engine connection, specified as a `fixflyer` object.

`fixmsg` — FIX message
table | structure

FIX message, specified as a table or structure.

Example: `fixtable = table({'FIX.4.4';'FIX.4.4'},...`
`{'338';'339'},{'2';'2'},...`
`{datestr(now);datestr(now)},...`
`{'D';'D'},{'ABC';'ABC'},...`
`{'1';'1'},{'D';'D'},{'100';'100'},...`
`'VariableNames',{'BeginString' ...`
`'CLOrdId' 'Side' 'TransactTime' ...`
`'OrdType' 'Symbol' ...`
`'HandlInst' 'MsgType' 'OrderQty'});`

Data Types: `table` | `struct`

Output Arguments

`status` — Sent message status
logical

Sent message status, returned as an array of logical zeroes or ones. The array contains an entry for each FIX message in `fixmsg`. If a FIX message is sent successfully, `status` contains a zero. Otherwise, `status` contains a `1`.

See Also

See Also
addListener | close | fixflyer

Topics
"Create an Order Using FIX Flyer" on page 1-20

External Websites
FIX Trading Community

Introduced in R2015b

orderInfo

Retrieve FIX Flyer order status and information

Syntax

```
o = orderInfo(c)
o = orderInfo(c,status)
o = orderInfo(c,infoterm,infovalue)
```

Description

`o = orderInfo(c)` returns order information for all orders associated with the FIX Flyer connection `c`.

`o = orderInfo(c,status)` filters orders by the order status.

`o = orderInfo(c,infoterm,infovalue)` filters orders by a specified term `infoterm` and value `infovalue`.

Examples

Return Order Information for All Orders

First, create a FIX Flyer Engine connection, add a FIX Flyer event listener, and subscribe to FIX sessions as in "Create an Order Using FIX Flyer" on page 1-20. Then, create and send a FIX message for a new order. Display the order information for all orders.

Create structure `orderStruct` to contain the FIX message for a new order. This order is a market order to sell 1000 IBM shares.

```
orderStruct.BeginString{1,1} = 'FIX.4.4';
orderStruct.CLOrdId{1,1} = '338';
orderStruct.Side{1,1} = '2';
orderStruct.TransactTime{1,1} = datestr(now);
orderStruct.OrdType{1,1} = 'D';
```

```matlab
orderStruct.Symbol{1,1} = 'IBM';
orderStruct.HandlInst{1,1} = '1';
orderStruct.MsgType{1,1} = 'D';
orderStruct.OrderQty{1,1} = '1000';
orderStruct.HeaderFields{1,1} = {'OnBehalfOfCompID','TRADER'};
orderStruct.BodyFields{1,1} = {'NoPartyIDs','3'; ...
                               'PartyID','1'; ...
                               'PartyRole','BBVA'; ...
                               'PartyID','1'; ...
                               'PartyRole','CVGX'; ...
                               'PartyID','1'; ...
                               'PartyRole','GSAM'};
```

Send FIX message using the FIX Flyer connection and the FIX message.

```matlab
status = sendMessage(c,orderStruct);
```

Return and display the order information **o** for all orders. The Variables editor displays the contents of **o**.

```matlab
o = orderInfo(c);
openvar('o')
```

Close the FIX Flyer Engine connection.

```matlab
close(c)
```

Return Order Information for All Open Orders

First, create a FIX Flyer Engine connection, add a FIX Flyer event listener, and subscribe to FIX sessions as in "Create an Order Using FIX Flyer" on page 1-20. Then, create and send a FIX message for a new order. Display the order information for all open orders.

Create structure **orderStruct** to contain the FIX message for a new order. This order is a market order to sell 1000 IBM shares.

```matlab
orderStruct.BeginString{1,1} = 'FIX.4.4';
orderStruct.CLOrdId{1,1} = '338';
orderStruct.Side{1,1} = '2';
orderStruct.TransactTime{1,1} = datestr(now);
orderStruct.OrdType{1,1} = 'D';
orderStruct.Symbol{1,1} = 'IBM';
orderStruct.HandlInst{1,1} = '1';
orderStruct.MsgType{1,1} = 'D';
```

```
orderStruct.OrderQty{1,1} = '1000';
orderStruct.HeaderFields{1,1} = {'OnBehalfOfCompID','TRADER'};
orderStruct.BodyFields{1,1} = {'NoPartyIDs','3'; ...
                               'PartyID','1'; ...
                               'PartyRole','BBVA'; ...
                               'PartyID','1'; ...
                               'PartyRole','CVGX'; ...
                               'PartyID','1'; ...
                               'PartyRole','GSAM'};
```

Send FIX message using the FIX Flyer connection and the FIX message.

```
status = sendMessage(c,orderStruct);
```

Return and display the order information o for all open orders. The Variables editor
displays the contents of o.

```
o = orderInfo(c,'open');
openvar('o')
```

Close the FIX Flyer Engine connection.

```
close(c)
```

Return Order Information for Specific Symbol

First, create a FIX Flyer Engine connection, add a FIX Flyer event listener, and
subscribe to FIX sessions as in "Create an Order Using FIX Flyer" on page 1-20. Then,
create and send a FIX message for a new order. Display the order information for orders
using a specific symbol.

Create structure orderStruct to contain the FIX message for a new order. This order is
a market order to sell 1000 IBM shares.

```
orderStruct.BeginString{1,1} = 'FIX.4.4';
orderStruct.CLOrdId{1,1} = '338';
orderStruct.Side{1,1} = '2';
orderStruct.TransactTime{1,1} = datestr(now);
orderStruct.OrdType{1,1} = 'D';
orderStruct.Symbol{1,1} = 'IBM';
orderStruct.HandlInst{1,1} = '1';
orderStruct.MsgType{1,1} = 'D';
orderStruct.OrderQty{1,1} = '1000';
orderStruct.HeaderFields{1,1} = {'OnBehalfOfCompID','TRADER'};
orderStruct.BodyFields{1,1} = {'NoPartyIDs','3'; ...
```

```
                                'PartyID','1'; ...
                                'PartyRole','BBVA'; ...
                                'PartyID','1'; ...
                                'PartyRole','CVGX'; ...
                                'PartyID','1'; ...
                                'PartyRole','GSAM'};
```

Send FIX message using the FIX Flyer connection and the FIX message.

```
status = sendMessage(c,orderStruct);
```

Return and display the order information `o` for transactions of IBM shares. The Variables editor displays the contents of `o`.

```
o = orderInfo(c,'symbol','IBM');
openvar('o')
```

Close the FIX Flyer Engine connection.

```
close(c)
```

• "Create an Order Using FIX Flyer" on page 1-20

Input Arguments

c — FIX Flyer Engine connection
`fixflyer` object

FIX Flyer Engine connection, specified as a `fixflyer` object.

status — Order status
`'all'` (default) | `'closed'` | `'open'`

Order status, specified as one of these values. Each value specifies the order information to return.

Order Status Value	Description
`'all'`	All orders
`'closed'`	Closed orders only
`'open'`	Open orders only

Example: `o = orderInfo(c,'all')`

Data Types: `char`

`infoterm` — Order information term
`'clientorderid'` | `'orderstatus'` | `'securityid'` | `'symbol'`

Order information term, specified as one of these values. Each value filters the order information to return.

Value	Description
`'clientorderid'`	Client order identifier
`'orderstatus'`	Order status
`'securityid'`	Security identifier
`'symbol'`	Symbol

To filter order information, specify a corresponding order information term value `infovalue` after `infoterm`. For example, to specify a client order identifier of `10`, use `'clientorderid','10'`.

Example: `o = orderInfo(c,'orderstatus','1')`

Data Types: `char`

`infovalue` — Order information term value
character vector

Order information term value, specified as a character vector.

To filter order information, specify this value after a corresponding order information term `infoterm`. For example, to specify the IBM symbol, use `'symbol','IBM'`.

Example: `o = orderInfo(c,'orderstatus','1')`

Data Types: `char`

Output Arguments

`o` — Order information data
structure

Order information data, returned as a structure. The structure contains many fields where each field is one piece of order information data provided by FIX Flyer.

See Also

See Also
addListener | fixflyer | sendMessage

Topics
"Create an Order Using FIX Flyer" on page 1-20

External Websites
FIX Trading Community

Introduced in R2016b

close

Close FIX Flyer connection

Syntax

```
close(c)
```

Description

`close(c)` closes the FIX Flyer Engine connection `c`.

Examples

Close the FIX Flyer Connection

Create the FIX Flyer Engine connection `c` using these arguments:

- User name `username`
- Password `password`
- IP address `ipaddress`
- Port number `port`

```
username = 'user';
password = 'pwd';
ipaddress = '127.0.0.1';
port = 7002;

c = fixflyer(username,password,ipaddress,port);
```

Close the FIX Flyer Engine connection.

```
close(c)
```

- "Create an Order Using FIX Flyer" on page 1-20

Input Arguments

c — FIX Flyer Engine connection
`fixflyer` object

FIX Flyer Engine connection, specified as a `fixflyer` object.

See Also

See Also
`fixflyer`

Topics
"Create an Order Using FIX Flyer" on page 1-20

Introduced in R2015b

fix2struct

Convert FIX message to structure array

Syntax

```
fixstruct = fix2struct(fixstr)
```

Description

`fixstruct = fix2struct(fixstr)` converts raw FIX messages in the cell array `fixstr` to a structure array `fixstruct`.

Examples

Convert FIX Message to Structure Array

For this example, assume that a counterparty sends you two raw FIX messages in `fixstr`. The FIX protocol version is 4.4.

Convert raw FIX messages in `fixstr` to a structure array `fixstruct`.

```
fixstruct = fix2struct(fixstr)

fixstruct =

     BeginString: {2x1 cell}
         ClOrdID: {2x1 cell}
            Side: {2x1 cell}
     TransactTime: {2x1 cell}
          OrdType: {2x1 cell}
           Symbol: {2x1 cell}
        HandlInst: {2x1 cell}
          MsgType: {2x1 cell}
         OrderQty: {2x1 cell}
```

The structure array `fixstruct` contains a structure for each raw FIX message in `fixstr`. The structure fields correspond to the FIX tags in the raw FIX message.

Display the order type for each FIX message.

```
fixstruct.OrdType

ans =

    'D'
    'D'
```

Both FIX messages specify previously quoted orders.

Input Arguments

`fixstr` — FIX message
cell array

FIX message, specified as a cell array of one or more raw FIX messages.

Example:

Data Types: `cell`

Output Arguments

`fixstruct` — FIX message
structure

FIX message, specified as a structure array containing the converted raw FIX messages in `fixstr`. The structure fields and values correspond to the FIX tag names and values in the raw FIX message.

See Also

See Also
fix2table | fixflyer | struct2fix | table2fix

External Websites
FIX Trading Community

Introduced in R2015b

fix2table

Convert FIX message to table

Syntax

```
fixtable = fix2table(fixstr)
```

Description

`fixtable = fix2table(fixstr)` converts raw FIX messages in the cell array `fixstr` to a table `fixtable`.

Examples

Convert FIX Message to Table

For this example, assume that a counterparty sends you two raw FIX messages in `fixstr`. The FIX protocol version is 4.4.

Convert raw FIX messages in `fixstr` to a table `fixtable`.

```
fixtable = fix2table(fixstr)
```

```
fixtable =
```

BeginString	MsgType	OnBehalfOfCompID	ClOrdID	Side	TransactTime
'FIX.4.4'	'D'	'TRADER'	'338'	'2'	'22-Mar-2016 11:34
'FIX.4.4'	'D'	'TRADER'	'339'	'2'	'22-Mar-2016 11:36

The table `fixtable` contains a row for each raw FIX message in `fixstr`. The variable names in the table correspond to the FIX tags in the raw FIX message.

Display the order type for each FIX message.

```
fixtable.OrdType
```

```
ans =

  2×1 cell array

    'D'
    'D'
```

Both FIX messages specify previously quoted orders.

Input Arguments

`fixstr` — FIX message
cell array

FIX message, specified as a cell array of one or more raw FIX messages.

Example:

Data Types: `cell`

Output Arguments

`fixtable` — FIX message
table

FIX message, specified as a table containing the converted raw FIX messages in `fixstr`. The table variables correspond to the FIX tag names that are specified in the raw FIX message. The table row contains the values that are specified for each tag in the raw FIX message.

See Also

See Also
fix2struct | fixflyer | struct2fix | table2fix

External Websites
FIX Trading Community

Introduced in R2015b

struct2fix

Convert structure array containing FIX tags to cell array of FIX messages

Syntax

```
fixstr = struct2fix(fixstruct)
```

Description

`fixstr = struct2fix(fixstruct)` converts FIX messages in a structure array `fixstruct` to raw FIX messages in the cell array `fixstr`.

Examples

Convert FIX Message from Structure Array to Character Vector

Create two FIX messages using a structure array `fixstruct`. The FIX protocol version is 4.4. Each FIX message represents a sell side transaction for 100 shares of symbol ABC. The order transaction time is the current moment. The order type is a previously quoted order. The order handling instruction is a private automated execution. The message type indicates a new order. The second structure in the structure array has the same order field values except that the order identifier is unique across orders.

```
fixstruct.BeginString{1,1} = 'FIX.4.4';
fixstruct.CLOrdId{1,1}  = '338';
fixstruct.Side{1,1}  = '2';
fixstruct.TransactTime{1,1}  = datestr(now);
fixstruct.OrdType{1,1}  = 'D';
fixstruct.Symbol{1,1}  = 'ABC';
fixstruct.HandlInst{1,1}  = '1';
fixstruct.OrderQty{1,1} = '100';
fixstruct.MsgType{1,1} = 'D';

fixstruct.BeginString{2,1} = 'FIX.4.4';
fixstruct.CLOrdId{2,1}  = '339';
fixstruct.Side{2,1}  = '2';
```

```
fixstruct.TransactTime{2,1}  = datestr(now);
fixstruct.OrdType{2,1}  = 'D';
fixstruct.Symbol{2,1}  = 'ABC';
fixstruct.HandlInst{2,1}  = '1';
fixstruct.OrderQty{2,1} = '100';
fixstruct.MsgType{2,1} = 'D';
```

Convert FIX messages in the structure array `fixstruct` to a cell array of raw FIX messages `fixstr`.

```
fixstr = struct2fix(fixstruct)


fixstr =

  2×1 cell array

    '8=FIX.4.4 35=D 11=338 54=2 60=16-Aug-2016 14:56:48 40=D 55=ABC 21=1 38...'
    '8=FIX.4.4 35=D 11=339 54=2 60=16-Aug-2016 14:56:48 40=D 55=ABC 21=1 38...'
```

Each character vector is a raw FIX message that contains FIX tags and values. The space in between the tag and value pairs is a SOH character. This character is not printable and has a hexadecimal value of `0x01`.

- "Create an Order Using FIX Flyer" on page 1-20

Input Arguments

`fixstruct` — FIX message
structure

FIX message, specified as a structure array. The data in the structure represents one FIX message. The structure fields correspond to FIX tag names. The structure values are the values that you specify in the FIX message.

Example: `fixStruct.BeginString{1,1} = 'FIX.4.4';`
`fixStruct.CLOrdId{1,1} = '338';`
`fixStruct.Side{1,1} = '2';`
`fixStruct.TransactTime{1,1} = datestr(now);`
`fixStruct.OrdType{1,1} = 'D';`
`fixStruct.Symbol{1,1} = 'ABC';`

```
fixStruct.HandlInst{1,1} = '1';
fixStruct.MsgType{1,1} = 'D';
fixStruct.OrderQty{1,1} = '100';
```

Data Types: `struct`

Output Arguments

`fixstr` — FIX message
cell array

FIX message, returned as a cell array of one or more converted raw FIX messages. The number of messages in the output argument depends on the number of messages that you specify in the input argument.

See Also

See Also
`fix2struct` | `fix2table` | `fixflyer` | `table2fix`

Topics
"Create an Order Using FIX Flyer" on page 1-20

External Websites
FIX Trading Community

Introduced in R2015b

table2fix

Convert table containing FIX tags to cell array of FIX messages

Syntax

```
fixstr = table2fix(fixtable)
```

Description

`fixstr = table2fix(fixtable)` converts the FIX messages in the table `fixtable` to raw FIX messages in the cell array `fixstr`.

Examples

Convert FIX Message from Table to Character Vector

Create two FIX messages using a table `fixtable`. The FIX protocol version is 4.4. The first row in the table represents a sell side transaction for 100 shares of symbol `ABC`. The order type is a previously quoted order. The order handling instruction is a private automated execution. The order transaction time is the current moment. The message type indicates a new order. The second row in the table has the same order field variables except that the order identifier is unique across orders.

```
fixtable = table({'FIX.4.4';'FIX.4.4'}, ...
    {'338';'339'},{'2';'2'}, ...
    {datestr(now);datestr(now)}, ...
    {'D';'D'},{'ABC';'ABC'}, ...
    {'1';'1'},{'D';'D'},{'100';'100'}, ...
    'VariableNames',{'BeginString' ...
    'CLOrdId' 'Side' 'TransactTime' ...
    'OrdType' 'Symbol' ...
    'HandlInst' 'MsgType' 'OrderQty'});
```

Convert the FIX messages in the table `fixtable` to a cell array of the raw FIX messages `fixstr`.

```
fixstr = table2fix(fixtable)
```

```
fixstr =

  2×1 cell array

    '8=FIX.4.4 35=D 11=338 54=2 60=16-Aug-2016 14:56:01 40=D 55=ABC 21=1 38...'
    '8=FIX.4.4 35=D 11=339 54=2 60=16-Aug-2016 14:56:01 40=D 55=ABC 21=1 38...'
```

Each character vector is a raw FIX message that contains FIX tags and values. The space in between the tag and value pairs is a SOH character. This character is not printable and has a hexadecimal value of `0x01`.

- "Create an Order Using FIX Flyer" on page 1-20

Input Arguments

`fixtable` — **FIX message**
table

FIX message, specified as table. The table variables correspond to FIX tag names. Each row contains the values you specify for the FIX message. Specify the values for each variable as a cell array of character vectors.

Example: `fixtable = table({'FIX.4.4';'FIX.4.4'},...`
`{'338';'339'},{'2';'2'},...`
`{datestr(now);datestr(now)},...`
`{'D';'D'},{'ABC';'ABC'},...`
`{'1';'1'},{'D';'D'},{'100';'100'},...`
`'VariableNames',{'BeginString' ...`
`'CLOrdId' 'Side' 'TransactTime' ...`
`'OrdType' 'Symbol' ...`
`'HandlInst' 'MsgType' 'OrderQty'});`

Data Types: `table`

Output Arguments

`fixstr` — **FIX message**
cell array

FIX message, returned as a cell array of one or more converted raw FIX messages. The number of messages in the output argument depends on the number of messages that you specify in the input argument.

See Also

See Also
fix2struct | fix2table | fixflyer | struct2fix

Topics
"Create an Order Using FIX Flyer" on page 1-20

External Websites
FIX Trading Community

Introduced in R2015b

krg

Create Kissell Research Group transaction-cost analysis object

To start with transaction cost analysis, use MATLAB to retrieve the encrypted market-impact parameters from the Kissell Research Group (KRG) FTP site. Then, use `krg` to create a transaction-cost analysis object to store the encrypted data. For details about market-impact parameters and data, consult the Kissell Research Group. For a simple example of estimating trading costs, see "Estimate Trading Costs for Collection of Stocks" on page 3-44.

Syntax

```
k = krg(midata)
k = krg(midata,midate)
k = krg(midata,midate,micode)
k = krg(midata,midate,micode,tradedays)
```

Description

`k = krg(midata)` creates a transaction-cost analysis object with market-impact parameter data.

`k = krg(midata,midate)` selects a market-impact date.

`k = krg(midata,midate,micode)` also selects a market-impact code.

`k = krg(midata,midate,micode,tradedays)` adds the number of trading days in a year.

Examples

Create Transaction-Cost Analysis Object

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the

MI_Encrypted_Parameters.csv file. miData contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object k.

```
k = krg(miData)

k =

  krg with properties:

             MiData: [276x12 table]
             MiDate: 09-Sep-2015
             MiCode: 1.00
    TradeDaysInYear: 250.00
```

k has these properties:

- Market-impact data
- Market-impact date
- Market-impact code
- Number of trading days in the year

You can estimate trading costs using the market activity the current day. For details, see "Estimate Trading Costs for Collection of Stocks" on page 3-44.

Create Transaction-Cost Analysis Object with Market-Impact Date

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the ftp function with a user name and password. Navigate to the MI_Parameters folder and retrieve the market-impact data in the MI_Encrypted_Parameters.csv file. miData contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
```

```
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k` with a specific market-impact date `midate`. Set the date to yesterday.

```
midate = 'yesterday';

k = krg(miData,midate)

k =

  krg with properties:

             MiData: [276x12 table]
             MiDate: 09-Sep-2015
             MiCode: 1.00
     TradeDaysInYear: 250.00
```

You can estimate trading costs using the market activity for yesterday. For details, see "Estimate Trading Costs for Collection of Stocks" on page 3-44.

Create Transaction-Cost Analysis Object with Market-Impact Code

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k` with a specific market-impact code `micode`. Set the date to yesterday. Set the code to one.

```
midate = 'yesterday';
micode = 1;
```

```
k = krg(miData,midate,micode)

k =

  krg with properties:

             MiData: [276x12 table]
             MiDate: 09-Sep-2015
             MiCode: 1.00
    TradeDaysInYear: 250.00
```

Using the market activity for yesterday, you can estimate trading costs for a particular market region. For details, see "Estimate Trading Costs for Collection of Stocks" on page 3-44.

Create Transaction-Cost Analysis Object with Number of Trading Days

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the ftp function with a user name and password. Navigate to the MI_Parameters folder and retrieve the market-impact data in the MI_Encrypted_Parameters.csv file. miData contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object k with a specific number of trading days in the year tradedays. Set the number of trading days to 251. By entering [] for the market-impact date and code, krg sets these input arguments to default values.

```
tradedays = 251;

k = krg(miData,[],[],tradedays)

k =

  krg with properties:
```

```
             MiData: [276x12 table]
             MiDate: 09-Sep-2015
             MiCode: 1.00
    TradeDaysInYear: 251.00
```

Using the market activity for yesterday, you can estimate trading costs for a particular market region with 251 trading days in the year. For details, see "Estimate Trading Costs for Collection of Stocks" on page 3-44.

Modify Transaction-Cost Analysis Object Property

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k` using `miData`.

```
k = krg(miData);
```

Modify the `MiDate` property to retrieve market-impact data from a different day.

```
k.MiDate = '05-Dec-2015'

k =

  krg with properties:

             MiData: [276x12 table]
             MiDate: '05-Dec-2015'
             MiCode: 1.00
    TradeDaysInYear: 251.00
```

You can estimate trading costs using the market activity for the specified day. For details, see "Estimate Trading Costs for Collection of Stocks" on page 3-44.

- "Analyze Trading Execution Results" on page 3-2

Input Arguments

`midata` — Market-impact data
table

Market-impact data, specified as a table. This table contains the encrypted market-impact date, code, and parameters. Retrieve this data from the Kissell Research Group FTP site ftp://ftp.kissellresearch.com using your user name and password. For details, consult the Kissell Research Group.

Data Types: `table`

`midate` — Market-impact date
double | character vector | scalar `datetime` array

Market-impact date, specified as a double, character vector, or scalar `datetime` array. By default, the market-impact date is the current date. To decrypt market-impact parameters for a specific date, set this date. For details, consult the Kissell Research Group.

Example: `midate = 'yesterday';`

Data Types: `double` | `char` | `datetime`

`micode` — Market-impact code
numeric scalar

Market-impact code, specified as a numeric scalar. By default, the market-impact code is one. To decrypt market-impact parameters for a specific market region, set this code. For details, consult the Kissell Research Group.

Example: `micode = 1;`

Data Types: `double`

`tradedays` — Number of trading days in the year
numeric scalar

Number of trading days in the year, specified as a numeric scalar. By default, the number of trading days in the year is 250.

Data Types: `double`

Output Arguments

k — Transaction cost analysis
KRG object

Transaction cost analysis, returned as a KRG object. This table describes the object properties.

Property	Description
`MiData`	Kissell Research Group market-impact parameter data
`MiDate`	Kissell Research Group market-impact date
`MiCode`	Kissell Research Group market-impact code
`TradeDaysInYear`	Number of trading days in the year

Tips

- If the market-impact code does not exist in the market-impact data, this error displays.

  ```
  The given region code does not match any records in the market impact data.
  ```

See Also

See Also
`iStar` | `marketImpact` | `priceAppreciation` | `timingRisk`

Topics

External Websites

ftp://ftp.kissellresearch.com

Introduced in R2016a

costCurves

Estimate market-impact cost of order execution

Syntax

```
cc = costCurves(k,trade,tradeQuantity,tqRange,tradeStrategy,tsRange)
```

Description

`cc = costCurves(k,trade,tradeQuantity,tqRange,tradeStrategy,tsRange)` returns the market-impact costs of order execution using:

- Kissell Research Group (KRG) transaction-cost analysis object `k`
- Trade data `trade`
- Trade quantity `tradeQuantity` with a range of values `tqRange`
- Trade strategy `tradeStrategy` with a range of values `tsRange`

Examples

Estimate Market-Impact Cost for an Order

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file KRGExampleData.mat, which is included with the Trading Toolbox.

```
load KRGExampleData
```

The variable TradeData appears in the MATLAB workspace.

TradeData contains these variables:

- Stock symbol
- Stock price
- Average daily volume
- Volatility

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Estimate market-impact costs with the trade quantity 'Size' and strategy 'POV'. Specify the trade quantity range with increments of 0.01 by starting from 0.01 and ending at one. Specify the trade strategy range with increments of 0.05 by starting from 0.05 and ending at 0.5.

```
cc = costCurves(k,TradeData,'Size',(0.01:0.01:1),'POV',(0.05:0.05:0.5));
```

Display the first three rows of market-impact cost data.

```
cc(1:3,:)
```

```
ans =
```

Symbol	Size	Shares	Dollars	POV	TradeTime	Cost_BP	Cost_
'AAL'	0.01	114764.24	6251208.50	0.05	0.19	11.42	0.06
'AAL'	0.01	114764.24	6251208.50	0.10	0.09	17.93	0.10
'AAL'	0.01	114764.24	6251208.50	0.15	0.06	23.42	0.13

The market-impact cost data contains:

- Stock symbol
- Size
- Number of shares in the transaction

5-337

- Dollar amount of the transaction
- Percentage of volume to complete the transaction
- Trade time to complete the transaction in percentage of the day
- Market-impact cost in basis points
- Market-impact cost in dollars per share
- Market-impact cost in dollars

Display cost curves for the first stock for these percentage of volume rates: 5%, 15%, 25%, and 35%.

```
figure
subplot(2,2,1)
plot(cc.Size(1:10:1000)*100,cc.Cost_BP(1:10:1000))
grid on
axis([0 100 0 250])
xlabel({'Size','(%ADV)'})
ylabel({'Cost','(bps)'})
title('POV = 5%')
a = gca;
a.XAxis.TickLabelFormat = '%g%%';

subplot(2,2,2)
plot(cc.Size(3:10:1000)*100,cc.Cost_BP(3:10:1000))
grid on
axis([0 100 0 250])
xlabel({'Size','(%ADV)'})
ylabel({'Cost','(bps)'})
title('POV = 15%')
b = gca;
b.XAxis.TickLabelFormat = '%g%%';

subplot(2,2,3)
plot(cc.Size(5:10:1000)*100,cc.Cost_BP(5:10:1000))
grid on
axis([0 100 0 250])
xlabel({'Size','(%ADV)'})
ylabel({'Cost','(bps)'})
title('POV = 25%')
c = gca;
c.XAxis.TickLabelFormat = '%g%%';

subplot(2,2,4)
```
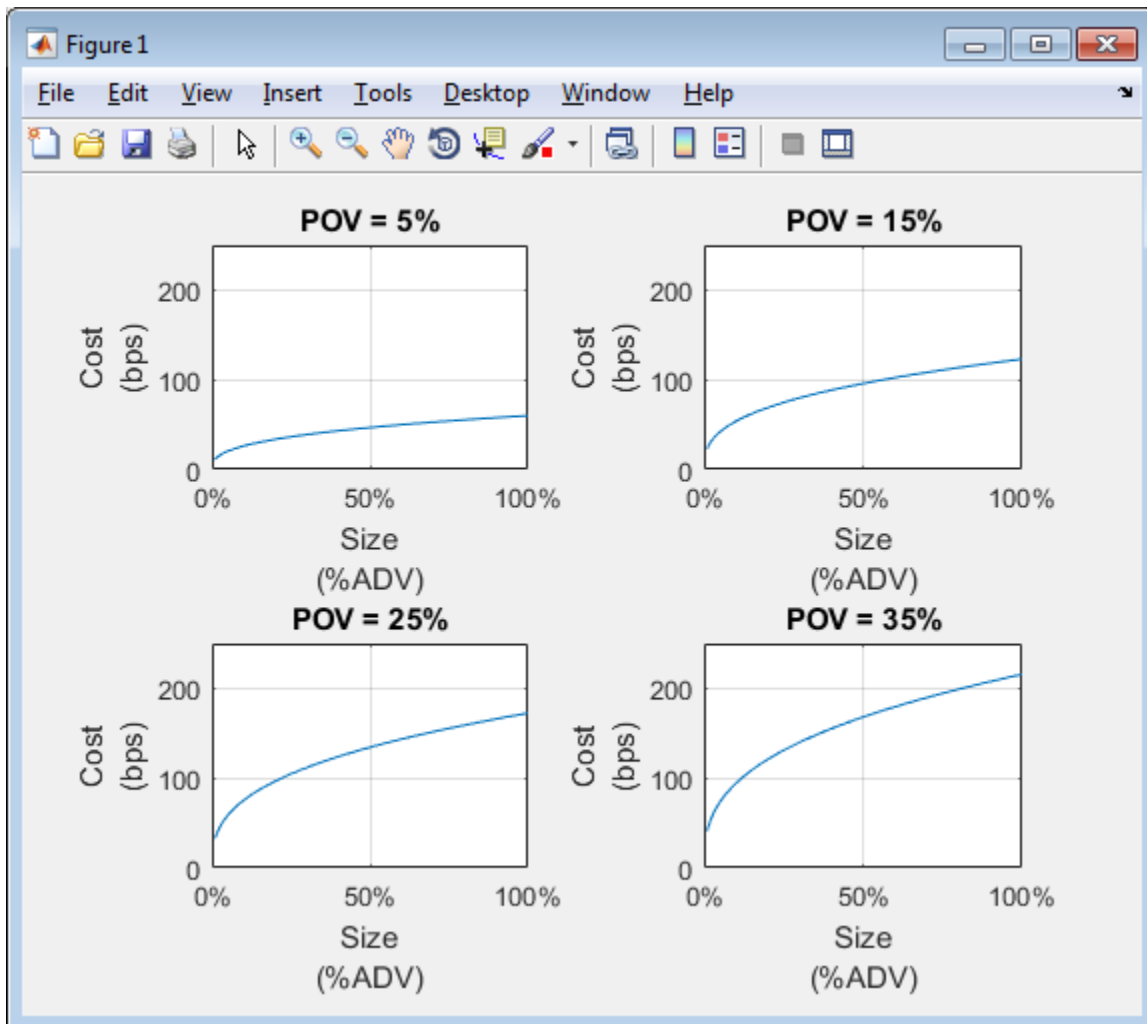
```
plot(cc.Size(7:10:1000)*100,cc.Cost_BP(7:10:1000))
grid on
axis([0 100 0 250])
xlabel({'Size','(%ADV)'})
ylabel({'Cost','(bps)'})
title('POV = 35%')
d = gca;
d.XAxis.TickLabelFormat = '%g%%';
```

This figure demonstrates how fast to trade a specific order size within a price level.

Input Arguments

k — Transaction cost analysis
KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

`trade` — Trade data
table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
Symbol	Stock symbol
Price	Stock price
ADV	Average daily volume
Volatility	Volatility

Example: `trade = table({'XYZ'},100.00,860000,0.27,'VariableNames', {'Symbol' 'Price' 'ADV' 'Volatility'})`

Example: `trade = struct('Symbol','XYZ','Price',100.00,'ADV',860000,'Volatility',0.27)`

These examples do not represent real market data.

Data Types: `struct` | `table`

`tradeQuantity` — Trade quantity
`'Size'` | `'Shares'` | `'Dollars'`

Trade quantity, specified as one of these values.

Value	Trade Quantity Description
`'Size'`	Shares in the transaction, which is a percentage of average daily trading volume

Value	Trade Quantity Description
'Shares'	Number of shares in the transaction
'Dollars'	Total value of the transaction

tqRange — Trade quantity range
vector

Trade quantity range, specified as a vector. `costCurves` uses these values with the trade strategy range values to estimate market-impact costs for different quantities and strategies.

Example: `'Size',(0.01:0.01:1)` specifies a trade quantity range with increments of 0.01 starting from 0.01 and ending at one

Data Types: `double`

tradeStrategy — Trade strategy
`'POV'` | `'TradeTime'`

Trade strategy, specified as one of these values.

Values	Trade Strategy Name
'POV'	Percentage of volume
'TradeTime'	Trade time in percentage of the day

tsRange — Trade strategy range
vector

Trade strategy range, specified as a vector. `costCurves` uses these values with the trade quantity range values to estimate market-impact costs for different quantities and strategies.

Example: `'POV',(0.05:0.05:0.5)` specifies a trade strategy range with increments of 0.05 starting from 0.05 and ending at 0.5

Data Types: `double`

Output Arguments

cc — Cost curves
table | structure

Cost curves, returned as a table or structure with these variable names or fields.

Variable or Field Name	Description
`Symbol`	Stock symbol
`Size`	Shares in a transaction in percentage of average daily trading volume
`Shares`	Number of shares in the transaction
`Dollars`	Dollar amount of the transaction
`POV`	Percentage of volume to complete the transaction
`TradeTime`	Trade time to complete the transaction in percentage of the day
`Cost_BP`	Market-impact cost of the transaction in basis points
`Cost_DollarsPerShare`	Market-impact cost of the transaction in dollars per share
`Cost_Dollars`	Market-impact cost of the transaction in dollars

Tips

- For details about the calculations, contact Kissell Research Group.

References

[1] Kissell, Robert. "A Practical Framework for Transaction Cost Analysis." *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. "Algorithmic Trading Strategies." Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[4] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[5] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

See Also
iStar | krg | marketImpact | portfolioCostCurves | timingRisk

Topics
"Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22

Introduced in R2016a

iStar

Estimate instantaneous trading cost for order

Syntax

```
itc = iStar(k,trade)
```

Description

`itc = iStar(k,trade)` returns the instantaneous trading cost of an order using the Kissell Research Group (KRG) transaction-cost analysis object `k` and trade data `trade`. To estimate the instantaneous trading cost, `iStar` uses the I-Star trading cost model.

Examples

Estimate Instantaneous Trading Cost for Stocks

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Stock symbol
- Side
- Number of shares
- Size
- Stock price
- Average daily volume
- Volatility
- Percentage of volume

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Estimate instantaneous trading cost `itc` for each stock using the Kissell Research Group transaction-cost analysis object `k`. Display the first three instantaneous trading costs.

```
itc = iStar(k,TradeData);

itc(1:3)

ans =

        33.48
       317.58
        62.94
```

Instantaneous trading costs display in basis points.

- "Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22

Input Arguments

k — Transaction cost analysis
KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

`trade` — Trade data
table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
`Symbol`	Stock symbol
`Side`	Buy or sell side
`Shares`	Number of shares in the transaction
`Size`	Shares in the transaction, which is a percentage of average daily trading volume
`Price`	Stock price
`ADV`	Average daily volume
`Volatility`	Volatility
`POV`	Percentage of volume

The trading cost varies with the trade strategy. `iStar` determines the trade strategy using these variables in this order:

1 Percentage of volume
2 Trade time
3 Trade schedule

To change the trade strategy from percentage of volume to trade time, remove the variable `POV` in the table and add the variable `TradeTime` with trade time data. To use the trade schedule strategy, remove the variable `TradeTime` and add the `TradeSchedule` and `VolumeProfile` variables.

If you specify size in the trade data, `iStar` uses the `Size` variable. Otherwise, `iStar` uses the variables `ADV` and `Shares` to determine the size.

For example, to create trade data as a table, enter:

```
trade = table({'XYZ'},{'Buy'},9300,0.06,29.68,860000,0.27,0.17,...
    'VariableNames',{'Symbol' 'Side' 'Shares' 'Size' 'Price' ...
```

```
    'ADV' 'Volatility' 'POV'})
```

To create trade data as a structure, enter:

```
trade.Symbol = {'XYZ'};
trade.Side = {'Buy'};
trade.Shares = 9300;
trade.Size = 0.06;
trade.Price = 29.68;
trade.ADV = 860000;
trade.Volatility = 0.27;
trade.POV = 0.17;
```

These examples do not represent real market data.

Data Types: `struct` | `table`

Output Arguments

`itc` — Instantaneous trading cost
vector

Instantaneous trading cost, returned as a vector. The vector values correspond to the instantaneous trading cost in basis points for each stock in `trade`.

Definitions

I-Star Trading Cost Model

The *I-Star trading cost model (I-Star)* estimates the instantaneous cost of an order. If a market participant immediately releases the entire order to the market for execution, they incur this cost. This cost also refers to the market participant cost accounting for 100% of the market volume over the execution period.

The I-Star model is

$$I^* = a_1 \cdot \left(\frac{Shares}{ADV} \right)^{a_2} \cdot \sigma^{a_3}.$$

Shares are the number of shares to trade. *ADV* is the average daily volume of the stock. σ is the price volatility. a_1, a_2, and a_3 are the model parameters.

Model Parameter	Description
a_1	Price sensitivity to order flow
a_2	Order size shape
a_3	Volatility shape

The general I-Star model that includes stock-specific factors is

$$I^* = a_1 \cdot \left(\frac{Shares}{ADV} \right)^{a_2} \cdot \sigma^{a_3} \cdot Price^{a_5} \cdot X_k^{a_k}.$$

Price is the stock price. a_5 is the price shape model parameter. X_k is the stock-specific factor such as market capitalization, beta, P/E ratio, and Debt/Equity ratio. This formulation can include multiple stock-specific factors. a_k is the corresponding shape parameter for the stock-specific factor X_k.

Tips

•   For details about the formula and calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. "A Practical Framework for Transaction Cost Analysis." *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. "Algorithmic Trading Strategies." Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. "Creating Dynamic Pre-Trade Models: Beyond the Black Box." *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[4] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[5] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[6] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

See Also
krg | liquidityFactor | marketImpact | priceAppreciation | timingRisk

Topics
"Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22

Introduced in R2016a

liquidityFactor

Estimate and compare liquidation costs across stocks

Syntax

```
lf = liquidityFactor(k,trade)
```

Description

`lf = liquidityFactor(k,trade)` returns the ratio of liquidation costs due to liquidity demand by stock for an equal investment value, or liquidity factor. `liquidityFactor` uses the Kissell Research Group (KRG) transaction-cost analysis object `k` and trade data `trade`.

Examples

Determine Liquidity Factor for Stocks

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Stock symbol
- Stock price
- Average daily volume
- Volatility

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Determine liquidity factor `lf` for each stock using the Kissell Research Group transaction-cost analysis object `k`. Display the first three liquidity factor values.

```
lf = liquidityFactor(k,TradeData);

lf(1:3)

ans =

        0.30
        2.37
        0.35
```

`lf` returns the ratios for stock comparison due to liquidity demands.

- "Estimate Portfolio Liquidation Costs" on page 3-26

Input Arguments

k — Transaction cost analysis
KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

trade — Trade data
table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
`Symbol`	Stock symbol
`Price`	Stock price
`ADV`	Average daily volume
`Volatility`	Volatility

Example: `trade = table({'XYZ'},100.00,860000,0.27,'VariableNames', {'Symbol' 'Price' 'ADV' 'Volatility'})`

Example: `trade = struct('Symbol','XYZ','Price',100.00,'ADV',860000,'Volatility',0.27)`

These examples do not represent real market data.

Data Types: `struct` | `table`

Output Arguments

`lf` — Liquidity factor
vector

Liquidity factor, returned as a vector. The vector values are ratios that compare the liquidation costs due to liquidity demands across stocks in `trade` for the dollar value and execution strategy.

Definitions

Liquidity Factor

The *Liquidity Factor (LF)* is a stock-specific measure of price sensitivity to investment dollars.

LF provides investors with a fair and consistent comparison of expected liquidation costs across stocks. LF incorporates stock-specific information to determine its sensitivity to order flow and investment dollars. The LF metric shows the ratio of liquidation costs due to liquidity demand by stock for an equal investment value in each stock. Market impact relies on the order size or shares traded which vary from order to order. LF provides an

apples-to-apples comparison across financial instruments. Consider a stock I that has an LF = 0.10 and a stock II that has an LF = 0.20. Stock II is twice as expensive to transact for an equal dollar value. An investor buys or sells $1 million dollars of stock in stock I and stock II utilizing the same execution strategy. The cost of stock II is twice as large as stock I. The LF metric incorporates stock liquidity, volatility, and price to determine the LF trading cost parameter.

The LF model is

$$\text{LF} = a_1 \cdot \left(\frac{1}{ADV}\right)^{a_2} \cdot \sigma^{a_3} \cdot \left(\frac{1}{Price}\right)^{a_2} \cdot Price^{a_5}.$$

σ is price volatility. ADV is the average daily volume of the stock. $Price$ is the current stock price in local currency. a_1, a_2, a_3, and a_5 are the model parameters.

Model Parameter	Description
a_1	Price sensitivity to order flow
a_2	Order size shape
a_3	Volatility shape
a_5	Price shape

Tips

- For details about the formula and calculations, contact the Kissell Research Group.
- You can expand the LF model to include a stock-specific factor such as market capitalization, beta, P/E ratio, and Debt/Equity ratio. In this case, X_k denotes the stock-specific factor and a_k denotes the corresponding shape parameter. For details about implementing an expanded LF model, contact the Kissell Research Group.

References

[1] Kissell, Robert. "A Practical Framework for Transaction Cost Analysis." *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. "Algorithmic Trading Strategies." Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[4] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[5] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

See Also
iStar | krg | marketImpact | priceAppreciation | timingRisk

Topics
"Estimate Portfolio Liquidation Costs" on page 3-26

Introduced in R2016a

marketImpact

Estimate price movement due to order or trade

Syntax

```
mi = marketImpact(k,trade)
```

Description

`mi = marketImpact(k,trade)` returns the market impact cost for stocks using the Kissell Research Group (KRG) transaction-cost analysis object `k` and trade data `trade`.

Examples

Estimates Market-Impact Costs

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Stock symbol
- Side
- Number of shares
- Size
- Stock price
- Average daily volume
- Volatility
- Percentage of volume

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Estimates market-impact cost `mi` for each stock using the Kissell Research Group transaction-cost analysis object `k`. Display the first three market-impact costs.

```
mi = marketImpact(k,TradeData);

mi(1:3)

ans =

        0.51
       96.86
       10.72
```

Market-impact costs display in basis points.

Input Arguments

k — Transaction cost analysis
KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

trade — Trade data
table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
Symbol	Stock symbol
Side	Buy or sell side
Shares	Number of shares in the transaction
Size	Shares in the transaction, which is a percentage of average daily trading volume
Price	Stock price
ADV	Average daily volume
Volatility	Volatility
POV	Percentage of volume

The trading cost varies with the trade strategy. `marketImpact` determines the trade strategy using these variables in this order:

1 Percentage of volume
2 Trade time
3 Trade schedule

To change the trade strategy from percentage of volume to trade time, remove the variable `POV` in the table and add the variable `TradeTime` with trade time data. To use the trade schedule strategy, remove the variable `TradeTime` and add the `TradeSchedule` and `VolumeProfile` variables.

If you specify size in the trade data, `marketImpact` uses the `Size` variable. Otherwise, `marketImpact` uses the variables `ADV` and `Shares` to determine the size.

For example, to create trade data as a table, enter:

```
trade = table({'XYZ'},{'Buy'},9300,0.06,29.68,860000,0.27,0.17,...
    'VariableNames',{'Symbol' 'Side' 'Shares' 'Size' 'Price' ...
    'ADV' 'Volatility' 'POV'})
```

To create trade data as a structure, enter:

```
trade.Symbol = {'XYZ'};
trade.Side = {'Buy'};
trade.Shares = 9300;
trade.Size = 0.06;
trade.Price = 29.68;
trade.ADV = 860000;
trade.Volatility = 0.27;
trade.POV = 0.17;
```

These examples do not represent real market data.

Data Types: `struct` | `table`

Output Arguments

`mi` — Market-impact cost
vector

Market-impact cost, returned as a vector. The vector values correspond to the market-impact costs in basis points for each stock in `trade`.

Definitions

Market Impact

Market impact (MI) estimates the price movement in a stock caused by a particular trade or order.

Market-impact cost always causes adverse price movement. Buy orders push the stock price higher and sell orders push the stock price lower. Market-impact cost occurs for two reasons: liquidity demands of the traders or investor and the information content of

the order. The liquidity demand of a buy order requires the buyer to provide the market a premium to attract additional sells into the market. The liquidity demand of a sell order causes the seller to offer the stock at a discount to attract additional buys into the market. The information content of the trade typically signals to the market that the stock is under- or overvalued. Buy orders tend to signal to the market that the stock is undervalued thus causing an increase in price to correct for the mispricing. Sell orders tend to signal to the market that the stock is overvalued thus causing a decrease in price to correct for the mispricing. Market-impact cost depends on order size, volatility, company characteristics, and prevailing market conditions over the trading horizon such as liquidity and intraday trading patterns.

MI for an order that executes instantaneously is equal to the I-Star trading cost model (I-Star). For details about I-Star, see `iStar`. When MI equals I-Star, the trading costs are high and prices move adversely. Therefore, investors trade passively to reduce their cost. Thus, they slice the order and trade over time such as minutes, hours, or possibly days. `marketImpact` incorporates the trade strategy of the investors into the cost calculation.

The MI model is

$$\mathrm{MI} = b_1 \cdot I^* \cdot POV^{a_4} + (1 - b_1) \cdot I^*.$$

I^* is I-Star. POV is the percentage of market volume, or participation fraction, of the order. a_4 and b_1 are the model parameters.

Model Parameter	Description
a_4	Percentage of volume rate shape
b_1	Percentage of temporary market impact. Temporary impact is dependent upon the trading strategy. Temporary impact occurs because of the liquidity demands of the investor.
$1 - b_1$	Percentage of permanent market impact. Permanent impact is the unavoidable impact cost. The order does not control the permanent impact. Permanent impact occurs because of the information content of the trade.

Tips

- For details about the formula and calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. "A Practical Framework for Transaction Cost Analysis." *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. "Algorithmic Trading Strategies." Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. "Creating Dynamic Pre-Trade Models: Beyond the Black Box." *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[4] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[5] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[6] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

See Also

iStar | krg | liquidityFactor | priceAppreciation | timingRisk

Topics

Introduced in R2016a

portfolioCostCurves

Estimate market-impact cost of order execution for portfolio

Syntax

```
pcc = portfolioCostCurves(k,portfolio,tradeQuantity,tqRange,
tradeStrategy,tsRange)
```

Description

`pcc = portfolioCostCurves(k,portfolio,tradeQuantity,tqRange, tradeStrategy,tsRange)` returns the market-impact cost of order execution for a portfolio using:

- Kissell Research Group (KRG) transaction-cost analysis object `k`
- Portfolio data `portfolio`
- Trade quantity `tradeQuantity` with a range of values `tqRange`
- Trade strategy `tradeStrategy` with a range of values `tsRange`

Examples

Estimate Market-Impact Cost for a Portfolio Order

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
```

```
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object k.

```
k = krg(miData);
```

Load the example portfolio data from the file KRGExampleData.mat, which is included with the Trading Toolbox.

```
load KRGExampleData
```

The variable PortfolioData appears in the MATLAB workspace.

PortfolioData contains these variables:

- Stock symbol
- Local price
- Price in a different currency if applicable
- Average daily volume
- Volatility
- Number of shares

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Estimate market-impact cost for an order execution on a portfolio of assets. Specify the trade quantity as DollarValue. Specify the trade quantity range tqRange with increments of $10,000,000. Start with a total portfolio value of $100,000,000 and end with $500,000,000. Set the percentage of volume trading strategy POV. Specify the trade strategy range tsRange with increments of 10% by starting with a percentage of volume of 10% and ending with 40%.

```
tqRange = (100000000:10000000:500000000);
tsRange = (0.10:0.10:0.40);

pcc = portfolioCostCurves(k,PortfolioData,'DollarValue',tqRange,...
'POV',tsRange);
```

Display the first three rows of market-impact cost data.

```
pcc(1:3,:)
```

```
ans =

    Size      Shares       TradeValue      AbsTradeValue      POV     TradeTime     Cost_b
    ____    _____    _____    _____     ____    _____     _____

    0.02    5612057.03    100000000.00     328737579.09       0.10      0.18        38.74
    0.02    5612057.03    100000000.00     328737579.09       0.20      0.08        61.18
    0.02    5612057.03    100000000.00     328737579.09       0.30      0.05        80.07
```

The market-impact cost data contains:

- Average trade size across all stocks in the portfolio
- Number of shares in the transaction
- Sum of traded value across all stocks in the portfolio
- Sum of absolute value of the trade value across all stocks in the portfolio
- Average execution percentage of volume to complete the number of shares
- Average trade time in percentage of the day to complete the number of shares
- Market-impact cost in basis points of local price
- Market-impact cost in dollars per share
- Market-impact cost in total dollar value

Display portfolio cost curves for percentage of volume rates: 10%, 20%, 30%, and 40%.

```
figure
size10 = pcc.Size(1:4:end)*100;
size20 = pcc.Size(2:4:end)*100;
size30 = pcc.Size(3:4:end)*100;
size40 = pcc.Size(4:4:end)*100;
cost10 = pcc.Cost_bp(1:4:end);
cost20 = pcc.Cost_bp(2:4:end);
cost30 = pcc.Cost_bp(3:4:end);
cost40 = pcc.Cost_bp(4:4:end);
plot(size10,cost10,size20,cost20,size30,cost30,size40,cost40)
grid on
axis([2 11 25 200])
xlabel({'Size','(%ADV)'})
ylabel({'Cost','(bps)'})
legend('POV = 10%','POV = 20%','POV = 30%','POV = 40%',...
'Location','northwest')
title('Portfolio Costs')
a = gca;
```

```
a.XAxis.TickLabelFormat = '%g%%';
```
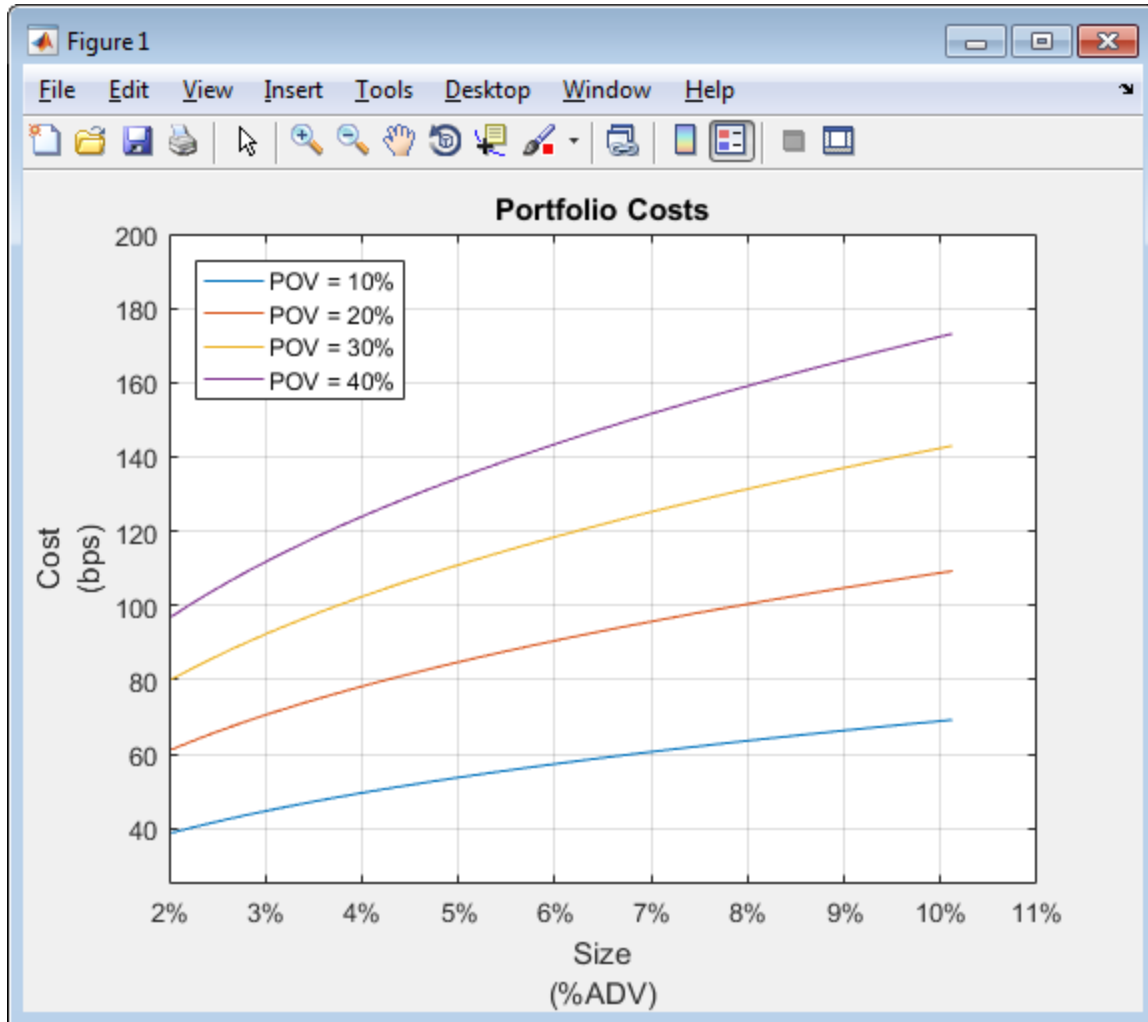

This figure demonstrates using portfolio costs to construct the portfolio and manage portfolio contents. By analyzing portfolio costs, you can determine the optimal portfolio size.

- "Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22

Input Arguments

k — Transaction cost analysis
KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

`portfolio` — Portfolio data
table | structure

Portfolio data that describes the stocks in the portfolio, specified as a table or structure. `portfolio` must contain these variable or field names.

Variable or Field Name	Description
Symbol	Stock symbol.
Price_Local	Local price.
Price_Currency	Price, specified as the stock price with a different currency if the stock trades outside the United States. If the stock trades in the United States, the value equals the local price.
ADV	Average daily volume.
Volatility	Volatility.
Shares	Number of shares.

The number of symbols in the portfolio data must match the number of values for each market-impact parameter in the `miData` property of k. For details about the market-impact parameters, contact the Kissell Research Group.

Example: `portfolio = table({'XYZ'},100.00,100.00,860000,0.27,550,'VariableNames', {'Symbol' 'Price_Local' 'Price_Currency' 'ADV' 'Volatility' 'Shares'})`

Example: `portfolio = struct('Symbol','XYZ','Price_Local',100.00,'Price_Currency',100.00,'ADV',86000`

These examples do not represent real market data.

Data Types: `struct` | `table`

tradeQuantity — Trade quantity
`'DollarValue' | 'PercentValue'`

Trade quantity, specified as one of these values.

Value	Trade Quantity Description
`'DollarValue'`	Total dollar value of the portfolio
`'PercentValue'`	Percentage of the total dollar value of the portfolio

tqRange — Trade quantity range
vector

Trade quantity range, specified as a vector. `portfolioCostCurves` uses these values with the trade strategy range values to estimate market-impact costs for different quantities and strategies.

Example: `'Size',(0.01:0.01:1)` specifies a trade quantity range with increments of 0.01 starting from 0.01 and ending at one

Data Types: `double`

tradeStrategy — Trade strategy
`'POV' | 'TradeTime'`

Trade strategy, specified as one of these values.

Values	Trade Strategy Name
`'POV'`	Percentage of volume
`'TradeTime'`	Trade time in percentage of the day

tsRange — Trade strategy range
vector

Trade strategy range, specified as a vector. `portfolioCostCurves` uses these values with the trade quantity range values to estimate market-impact costs for different quantities and strategies.

Example: `'POV',(0.05:0.05:0.5)` specifies a trade strategy range with increments of 0.05 starting from 0.05 and ending at 0.5

Data Types: `double`

Output Arguments

pcc — Portfolio cost curves
table | structure

Portfolio cost curves, returned as a table or structure with these variable names or fields.

Variable or Field Name	Description
Size	Average trade size across all stocks in the portfolio.
Shares	Number of shares in the transaction.
TradeValue	Trade value, or the total dollar value of the stock position in the portfolio adjusted for side. Long/Buy positions have a positive trade value and Short/Sell positions have a negative trade value.
AbsTradeValue	Sum of absolute value of the trade value across all stocks in the portfolio.
POV	Average execution percentage of volume to complete the number of shares.
TradeTime	Average trade time in percentage of the day to complete the number of shares.
Cost_bp	Market-impact cost in basis points of local price.
Cost_DollarsPerShare	Market-impact cost in dollars per share.
Cost_Dollars	Market-impact cost in total dollar value.

Tips

- To test multiple portfolio transactions, you can use different ranges. You can change the percentage of shares in the transaction or use a different trade strategy. For details, see "Input Arguments" on page 5-365.

• For details about the calculations, contact Kissell Research Group.

References

[1] Kissell, Robert. "A Practical Framework for Transaction Cost Analysis." *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. "Algorithmic Trading Strategies." Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[4] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management.* Cambridge, MA: Elsevier/Academic Press, 2013.

[5] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies.* New York, NY: AMACOM, Inc., 2003.

See Also

See Also
costCurves | iStar | krg | marketImpact | timingRisk

Topics
"Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22

Introduced in R2016a

priceAppreciation

Estimate trading cost due to natural price movement

Syntax

```
alpha = priceAppreciation(k,trade)
```

Description

`alpha = priceAppreciation(k,trade)` returns the trading cost due to the natural price movement of a stock, or price appreciation. `priceAppreciation` uses the Kissell Research Group (KRG) transaction cost object `k` and trade data `trade`.

Examples

Estimate Alpha

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Shares in the transaction, which is a percentage of average daily trading volume
- Number of shares
- Average daily volume
- Percentage of volume
- Trade time in percentage of the day
- Volatility
- Stock price
- Alpha estimate

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Estimate alpha using the Kissell Research Group transaction-cost analysis object `k`. Display the first three alphas.

```
alpha = priceAppreciation(k,TradeData);

alpha(1:3)

ans =

        -9.49
         8.47
         0.93
```

Alphas display in basis points.

- "Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22
- "Optimize Percentage of Volume Trading Strategy" on page 3-31
- "Optimize Trade Time Trading Strategy" on page 3-35
- "Optimize Trade Schedule Trading Strategy" on page 3-39

Input Arguments

k — Transaction cost analysis
KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

`trade` — Trade data
table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
Size	Shares in the transaction, which is a percentage of average daily trading volume
Shares	Number of shares
ADV	Average daily volume
POV	Percentage of volume
TradeTime	Trade time in percentage of the day
Volatility	Volatility
Price	Stock price
Alpha_bp	Alpha estimate in basis points

The trading cost varies with the trade strategy. `priceAppreciation` determines the trade strategy using these variables in this order:

1 Percentage of volume
2 Trade time
3 Trade schedule

To change the trade strategy from percentage of volume to trade time, remove the variable `POV` in the table and add the variable `TradeTime` with trade time data. To use the trade schedule strategy, remove the variable `TradeTime` and add the `TradeSchedule` and `VolumeProfile` variables.

If you specify size in the trade data, `priceAppreciation` uses the `Size` variable. Otherwise, `priceAppreciation` uses the variables `ADV` and `Shares` to determine the size.

Example: `trade = table(0.01,9300,860000,0.17,0.40,0.27,29.68,3,'VariableNames',`

```
{'Size' 'Shares' 'ADV' 'POV' 'TradeTime' 'Volatility' 'Price'
'Alpha_bp'})
```

Example: `trade =`
`struct('Size',0.01,'Shares',9300,'ADV',860000,'POV',0.17,'TradeTime',0.40,'Vol`

These examples do not represent real market data.

Data Types: `struct | table`

Output Arguments

`alpha` — Alpha
vector

Alpha, returned as a vector. The units of alpha, or the natural price movement of the stock, are basis points.

Definitions

Price Appreciation

Price appreciation (PA) estimates the trading cost due to the natural price movement of a stock.

The natural price movement commonly refers to expected return, alpha, price trend, drift, or momentum. This movement represents how the stock moves in a market without any uncertainty. PA represents the trading cost due to the underlying trading strategy. For example, buying passively in a rising market or selling passively in a falling market causes the fund to incur higher costs due to market movement. Conversely, buying in a falling market or selling in a rising market causes the fund to incur lower costs due to transacting at the better prices. PA is based on the alpha estimate you specify in the trade data. Funds and managers heavily guard their alpha estimates and expected returns. These expectations are highly proprietary and valued. This function lets you input alpha estimates directly into the model running on your desktop that prevents information leakage.

The PA model is represented as a linear trend. The PA model is

$$PA = 0.5 \cdot Alpha\_bp \cdot \left(\frac{Shares}{ADV} \right) \cdot \left(\frac{1 - POV}{POV} \right).$$

Shares are the number of shares to trade. *ADV* is the average daily volume of a stock. *POV* is the percent of market volume, or participation fraction, for the order. *Alpha_bp* is the alpha estimate for the day in basis points. A positive value for the alpha estimate indicates adverse price movement for the order. A negative value for the alpha estimate indicates favorable price movement.

Tips

- For details about the formula and calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. "A Practical Framework for Transaction Cost Analysis." *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. "Algorithmic Trading Strategies." Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[4] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[5] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

See Also
iStar | krg | liquidityFactor | marketImpact | timingRisk

Topics
"Conduct Sensitivity Analysis to Estimate Trading Costs" on page 3-22

Introduced in R2016a

timingRisk

Estimate uncertainty of market-impact cost

Syntax

```
tr = timingRisk(k,trade)
```

Description

`tr = timingRisk(k,trade)` returns the uncertainty of the market-impact cost estimate, or timing risk. `timingRisk` uses the Kissell Research Group (KRG) transaction-cost analysis object `k` and trade data `trade`.

Examples

Estimate Timing Risk for Stocks

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
cd(f,'MI_Parameters');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction-cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Trading Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Stock symbol
- Side
- Number of shares
- Size
- Stock price
- Average daily volume
- Volatility
- Percentage of volume

For a description of the example data, see "Kissell Research Group Data Sets" on page 3-9.

Estimate timing risk `tr` for each stock using the Kissell Research Group transaction-cost analysis object `k`. Display the first three timing risk values.

```
tr = timingRisk(k,TradeData);

tr(1:3)

ans =

        159.05
        242.37
         62.88
```

Timing risk trading costs display in basis points.

Input Arguments

k — Transaction cost analysis
KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

`trade` — Trade data
table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
Symbol	Stock symbol
Side	Buy or sell side
Shares	Number of shares in the transaction
Size	Shares in the transaction, which is a percentage of average daily trading volume
Price	Stock price
ADV	Average daily volume
Volatility	Volatility
POV	Percentage of volume

The trading cost varies with the trade strategy. `timingRisk` determines the trade strategy using these variables in this order:

1. Percentage of volume
2. Trade time
3. Trade schedule

To change the trade strategy from percentage of volume to trade time, remove the variable `POV` in the table and add the variable `TradeTime` with trade time data. To use the trade schedule strategy, remove the variable `TradeTime` and add the `TradeSchedule` and `VolumeProfile` variables.

If you specify size in the trade data, `timingRisk` uses the `Size` variable. Otherwise, `timingRisk` uses the variables `ADV` and `Shares` to determine the size.

For example, to create trade data as a table, enter:

```
trade = table({'XYZ'},{'Buy'},9300,0.06,29.68,860000,0.27,0.17,...
    'VariableNames',{'Symbol' 'Side' 'Shares' 'Size' 'Price' ...
    'ADV' 'Volatility' 'POV'})
```

To create trade data as a structure, enter:

```
trade.Symbol = {'XYZ'};
trade.Side = {'Buy'};
trade.Shares = 9300;
trade.Size = 0.06;
trade.Price = 29.68;
trade.ADV = 860000;
trade.Volatility = 0.27;
trade.POV = 0.17;
```

These examples do not represent real market data.

Data Types: `struct` | `table`

Output Arguments

`tr` — Timing risk
vector

Timing risk, returned as a vector. The vector values correspond to the timing risk in basis points for each stock in `trade`.

Definitions

Timing Risk

Timing risk (TR) estimates the uncertainty surrounding the estimated transaction cost.

Price volatility and liquidity risk creates uncertainty. Price volatility causes the price to be either higher or lower than expected due to factors independent of the order. Liquidity risk causes the market-impact cost to be either higher or lower than estimated due to market volumes. TR is dependent upon volumes, intraday trading patterns, and market impact resulting from other market participants. The TR model is

$$\text{TR} = \sigma \cdot \sqrt{\frac{1}{3} \cdot \frac{1}{250} \cdot \frac{Shares}{ADV} \cdot \left(\frac{1-POV}{POV}\right)} \cdot 10^4.$$

σ is price volatility. 250 is the number of trading days in the year. *Shares* are the number of shares to trade. *ADV* is the average daily volume of the stock. *POV* is the percentage of market volume, or participation fraction, of the order.

Tips

- For details about the formula and calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. "A Practical Framework for Transaction Cost Analysis." *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. "Algorithmic Trading Strategies." Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. "TCA in the Investment Process: An Overview." *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[4] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[5] Glantz, Morton, and Robert Kissell. *Multi-Asset Risk Modeling*. Cambridge, MA: Elsevier/Academic Press, 2013.

[6] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

See Also
iStar | krg | liquidityFactor | marketImpact | priceAppreciation

Topics

Introduced in R2016a