IB-Matlab User Guide

Version 1.08

March 22, 2012

http://UndocumentedMatlab.com/IB-Matlab

Undocumented Matlab

unbelievable features; unbelievable quality; unbelievable cost effectiveness; unbelievable service

Table of Contents

DISCLAIMER

THIS SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

MUCH EFFORT WAS INVESTED TO ENSURE THE CORRECTNESS, ACCURACY AND USEFULLNESS OF THE INFORMATION PRESENTED IN THIS DOCUMENT. HOWEVER, THERE IS NO GUARANTEE THAT THE INFORMATION IS COMPLETE OR ERROR-FREE. THE AUTHOR AND COPYRIGHT HOLDERS CANNOT TAKE ANY RESPONSIBILITY FOR POSSIBLE CONSEQUENCES DUE TO THIS DOCUMENT OR THE USE OF THE IB-MATLAB SOFTWARE.

WHEN USING THIS DOCUMENT AND THE IB-MATLAB SOFTWARE, USERS MUST VERIFY THE BEHAVIOUR CAREFULLY ON THEIR SYSTEM BEFORE USING THE SAME FUNCTIONALITY FOR LIVE TRADES. USERS SHOULD EITHER USE THIS DOCUMENT AND IB-MATLAB AT THEIR OWN RISK, OR NOT AT ALL.

1 Introduction

InteractiveBrokers (*IB*, www.interactivebrokers.com) provides brokerage and financial data-feed services. IB customers can use its services using specialized applications (so-called "*clients*") that can be installed on the user's computer. These client applications provide a user-interface that enables the user to view portfolio and market information, as well as to issue orders to the IB server. The most widely-used IB client application is TWS (*Trader Work Station*).[1]

In addition to TWS, IB provides other means of communicating with its server. A lightweight client application called the *IB Gateway* is installed together with TWS. IB Gateway does not have a fancy GUI like TWS, but provides exactly the same API functionality as TWS while using fewer system resources and running more efficiently.[2]

IB also publishes an interface (*Application Programming Interface*, or *API*) that enables user applications to connect to the IB server using one of its client applications (either IB Gateway or TWS). This API enables user trading models to interact with IB: send trade orders, receive market and portfolio information, process execution and tick events etc.

IB provides its API for three target platforms: Windows, Mac and Linux (Mac and Linux actually use the same API installation).[3] The API has several variants, including C++, Java, DDE, and ActiveX (DDE and ActiveX only on Windows).

Matlab is a programming development platform that is widely-used in the financial sector. Matlab enables users to quickly analyze data, display results in graphs or interactive user interfaces, and to develop automated, semi-automated and decision-support trading models.

Unfortunately, IB does not provide an official Matlab API connector. While IB's Java connector can be used directly in Matlab, setting up the event callbacks and data conversions between Matlab and the connector is definitely not easy. You need to be familiar with both Matlab *and* Java, at least to some degree.

This is the role of **IB-Matlab** (http://UndocumentedMatlab.com/IB-Matlab). **IB-Matlab** uses IB's Java API to connect Matlab to IB, providing a seamless interface within Matlab to the entire set of Java API functionality. Users can activate IB's API by simple Matlab commands, without any need to know Java.

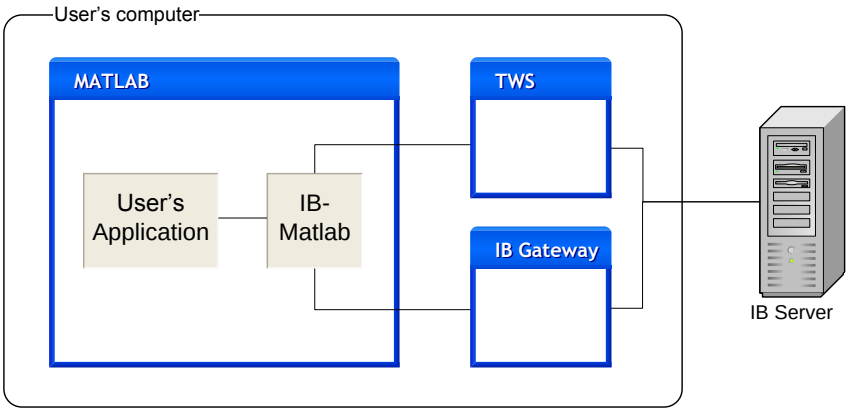
In fact, Java's cross-platform compatibility means that exactly the same **IB-Matlab** code runs on all platforms supported by both IB and Matlab, namely Windows (both 32 and 64 bit), Macs and Linux/Unix.

---

[1] http://www.interactivebrokers.com/en/software/tws/twsguide.htm#usersguidebook/getstarted/intro_to_get_started.htm

[2] http://www.interactivebrokers.com/en/p.php?f=programInterface#connectivity-clear; http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/api/run_the_api_through_the_ib_gateway.htm

[3] http://www.interactivebrokers.com/en/p.php?f=programInterface



IB-Matlab consists of two parts that provide different ways of interacting with IB:

1. A Java package (*IBMatlab.jar*) that connects to the TWS/Gateway and provides full access to IB's Java API functionality. Matlab users can use a special connector object in Matlab to invoke the Java API functions directly from within Matlab.
2. A Matlab wrapper (*IBMatlab.p*[4]) that does not provide 100% of the functionality, only the 20% that is used 80% of the time. This wrapper is a pure Matlab implementation that enables Matlab users to access the most important IB functionalities without needing to know anything about Java.

Active trading actions (buy, sell, short and cancel) and query actions (market, streaming quotes, open orders, historical, account and portfolio data) can be initiated with simple one-line Matlab code that uses the Matlab wrapper (*IBMatlab.p*). Additional trading features (the full range of IB's Java API) can be accessed using the connector object exposed by **IB-Matlab**.

Users can easily attach Matlab code (callbacks) to IB events. This enables special operations (e.g., adding an entry in an Excel file, sending an email or SMS) whenever an order is fully or partially executed, or a specified price is reached, for example.

Reviews of **IB-Matlab**'s Matlab wrapper were published in 2011[5] and 2012[6] in *Automated Trader* magazine and can be downloaded from **IB-Matlab**'s web page.[7]

This document explains how to install and use **IB-Matlab**. Depending on the date that you have installed **IB-Matlab**, some features discussed in this document may not work as described. However, as part of your annual license renewal you will receive the latest version of **IB-Matlab** that includes all the functionality detailed herein.

---

[4] In **IB-Matlab** releases prior to February 15, 2012, this wrapper was named *IB_trade*, not *IBMatlab*. *IB_trade* has exactly the same interface and functionality as *IBMatlab*, except for features that were added since 2/2012. In the vast majority of cases, any use of *IBMatlab* in this document can be substituted with *IB_trade* without any further change.

[5] http://www.automatedtrader.net/articles/software-review/84091/the-virtue-of-simplicity

[6] http://www.automatedtrader.net/articles/software-review/107768/mashup

[7] http://undocumentedmatlab.com/files/IB-Matlab_Review.pdf, http://undocumentedmatlab.com/files/IB-Matlab_Review2.pdf

2 Installation

IB-Matlab requires the following in order to run:

1. An active account at IB

   Use of IB's Demo account is not recommended because it is very limited in comparison with the functionalities of a live account. If you wish to test IB-Matlab, we recommend using the Paper-Trade account that IB assigns you when you register. A Paper-Trade account resembles the live account much more closely than a Demo account.

2. An installation of TWS and/or the IB Gateway – (normally installed together)

3. An installation of Matlab 7.1 (R14 SP3) or a newer release

   If you have an earlier release of Matlab, some API functionality may still be available on your system. Contact Yair (altmany@gmail.com) for details.

The installation procedure for **IB-Matlab** is as follows:

1. Ensure that you have read and accepted the license agreement. This is required even for the trial version of **IB-Matlab**. If you do not accept the license agreement then you cannot use **IB-Matlab**.

2. Place the **IB-Matlab** files (esp. *IBMatlab.jar*, *IBMatlab.p, and IBMatlab.m*) in a dedicated folder (for example: *C:\IB-Matlab*). Do not place the files in one of Matlab's installation folders.

3. Add the new **IB-Matlab** folder to your Matlab path using the path tool (in Matlab's Desktop menu, run File / Set path… and save). The **IB-Matlab** folder needs to be in your Matlab path whenever you run *IBMatlab*.

4. Type the following in your Matlab command prompt:

   ```
   >> edit('classpath.txt');
   ```

   This will open the *classpath.txt* file for editing in the Matlab editor. This file includes the Java static classpath that is used in Matlab and is typically located under the %matlabroot%\toolbox\local\ folder (e.g., *C:\Program Files\Matlab\R2011b\toolbox\local*).

5. Add the full path to the *IBMatlab.jar* file into the *classpath.txt* file (you may need to repeat this step whenever you install a new Matlab release on your computer). For example, on a Windows computer, if you placed the **IB-Matlab** files in *C:\IB-Matlab*, then the new line in the *classpath.txt* file should be as follows:

   C:\IB-Matlab\IBMatlab.jar

When trying to save the *classpath.txt* file, you might get an error message saying that the file is read-only. To solve this, enable write-access to this file:

In Windows, open Windows Explorer, right-click the *classpath.txt* file, select "Properties", and unselect the "Read-only" attribute; In Linux/Unix, run *chmod a+w classpath.txt*.

If you cannot get administrator access to modify the file's read-only attribute, then place a copy of the *classpath.txt* file in your Matlab startup folder. This is the folder used when you start Matlab - you can check this by issuing the following command at the Matlab command prompt:

```
>> pwd
```

Note that placing the modified *classpath.txt* file in your Matlab startup folder enabled you to run **IB-Matlab** whenever you use this startup folder – so if you ever use a different Matlab startup folder then you will need to copy the *classpath.txt* file to the new startup folder. Also note that the *classpath.txt* file depends on the Matlab release – it will only work on your current release of Matlab – if you try to use a different Matlab release with this same startup folder, then Matlab itself (not just **IB-Matlab**) may fail to load.

For these reasons, a much safer approach is to update the *classpath.txt* file in Matlab's default location, namely the %matlabroot%\toolbox\local\ folder.

6. Restart Matlab (there is no need to restart the computer)

7. If you are running the Production version of **IB-Matlab**, then you will need to activate your license at this point. When you purchase your license you will be given specific instructions how to do this.

8. Ensure that either TWS or IB Gateway is working and logged-in to IB.

9. In TWS, go to the main menu's Edit / Global Configuration… / API / Settings and make the following changes:[8]

    a. enable the "Enable ActiveX and Socket Clients" checkbox

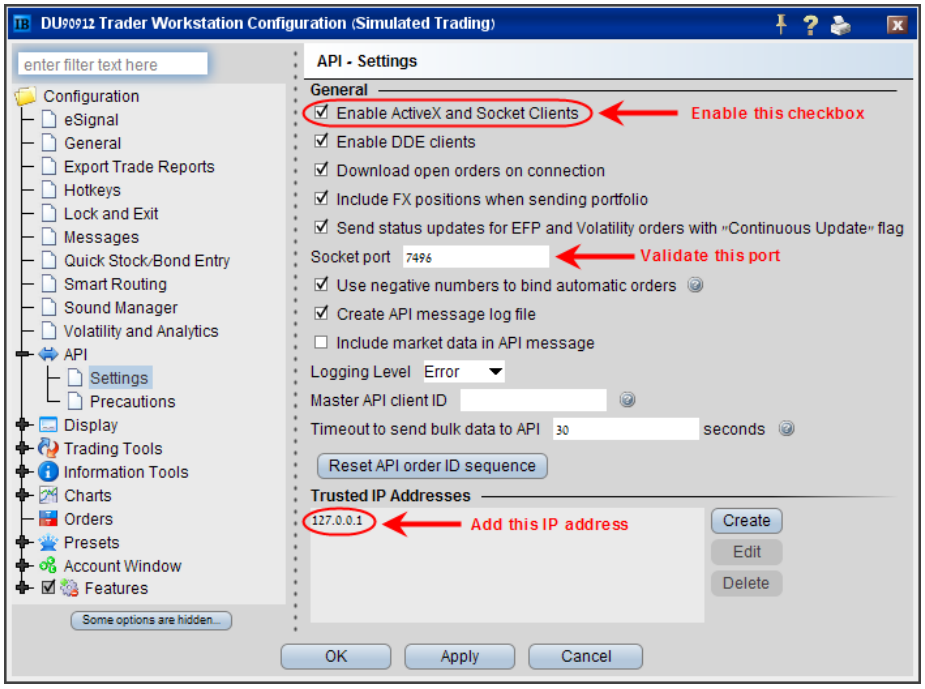
    b. add 127.0.0.1 (=localhost, i.e. the current computer) to the list of trusted IP addresses. If you wish to use **IB-Matlab** on a different computer than the one that runs TWS, add **IB-Matlab** machine's IP address to the list of trusted IP addresses.

    c. validate the Socket port used by TWS for API communication. This should normally be 7496. It can be changed to some other value, but

[8] If you do not make these changes, then **IB-Matlab** will either not be able to connect to IB, or will require popup confirmation upon each request by **IB-Matlab**.

then you will need to specify the Port parameter when you use *IBMatlab* for the first time after each Matlab restart.[9]

d. elect whether to specify a Master Client ID (positive integer number), that will provide access to all orders created by any API client. You can then use this in *IBMatlab* by specifying the ClientId parameter.[10]



10. If you plan to use the IB Gateway, then the configuration is very similar: Go to the Gateway's main menu Configure / Settings / API / Settings and modify the configuration as for TWS above. The only difference is that the "Enable ActiveX and Socket Clients" checkbox is not available in the Gateway's configuration, because it is always enabled for trusted IPs (which is good).[11]

11. You can now run *IBMatlab* within Matlab. To verify that **IB-Matlab** is properly installed, retrieve your current IB account information, using the following commands (which are explained in section 4 below):

```
>> data = IBMatlab('action','account_data')
>> data = IBMatlab('action','portfolio')
```

12. If you get an error "*IBMatlab.jar not found in static Java classpath. Please add IBMatlab.jar to classpath.txt*", then repeat step #5 above carefully, restart Matlab and retry step #11. If you still get the error, please contact Yair.

---

[9] See section 13 below for more details

[10] See section 13 below for more details

[11] If you forget to add the localhost IP to the list of trusted IP addresses, *IBMatlab* will complain that it cannot connect to IB

3 Using IBMatlab

IB-Matlab's Matlab wrapper function is called *IBMatlab*. This function is contained within the *IBMatlab.p* file. Its accompanying *IBMatlab.m* file provides online documentation using standard Matlab syntax, e.g.:

```
>> help IBMatlab
>> doc IBMatlab
```

The *IBMatlab* function accepts a variable number of parameters, and returns two objects: a `data` object (that contains the returned query data, or the order ID of a sent trade order), and a connector object that provides full access to the Java API.[12]

IBMatlab can accept input parameters in either of two formats:

- As name-value pairs – for example:

```
>> data = IBMatlab('action','account, 'AccountName','DU12345');
```

- As a pre-prepared struct of parameters – for example:

```
>> params = [];  % initialize
>> params.Action      = 'account';
>> params.AccountName = 'DU12345';
>> data = IBMatlab(params);
```

These input formats are equivalent. You can use whichever format you feel more comfortable with.

IBMatlab is quite tolerant of user input: parameter names are case-insensitive (whereas most IB values are case-sensitive), parameter order does not matter, and in some cases the inputs can be shortened. For example, the following are all equivalent:

```
>> data = IBMatlab('action','account', 'AccountName','DU12345');
>> data = IBMatlab('action','account_data', 'accountname','DU12345');
>> data = IBMatlab('action','account_data', 'AccountName','DU12345');
>> data = IBMatlab('Action','Account_Data', 'AccountName','DU12345');
>> data = IBMatlab('ACTION','ACCOUNT_DATA', 'AccountName','DU12345');
>> data = IBMatlab('AccountName','DU12345', 'action','account_data');
```

The full list of acceptable input parameters is listed in the sections below, grouped by usage classification.

When using *IBMatlab*, there is no need to worry about connecting or disconnecting from TWS/Gateway – *IBMatlab* handles these activities automatically, without requiring user intervention. The user only needs to ensure that TWS/Gateway is active and logged-in when the *IBMatlab* command is invoked in Matlab.

[12] See section 15 below for more details

Much of *IBMatlab*'s functionality relates to a specific security that you choose to query or trade. IB is not very forgiving if you do not provide the exact security specifications (a.k.a. *Contract*) that it expects: in such a situation, data is not returned, and an often-cryptic error message is displayed in Matlab's Command Window:[13]

```
>> data = IBMatlab('action','query', 'symbol','EUR')
 [API.msg2] No security definition has been found for the request
{153745243, 200}
data =
            reqId: 153745243
          reqTime: '13-Feb-2012 21:25:42'
         dataTime: ''
    dataTimestamp: -1
           ticker: 'EUR'
         bidPrice: -1
         askPrice: -1
             open: -1
            close: -1
              low: -1
             high: -1
        lastPrice: -1
           volume: -1
             tick: 0.01
```

Unfortunately, IB is not very informative about what exactly was wrong with our request, we need to discover this ourselves. It turns out that in this specific case, we need to specify a few additional parameters, since the defaults (localSymbol=symbol, secType='STK', exchange='SMART') are invalid for this security:

```
>> data = IBMatlab('action','query', 'symbol','EUR', ...
                   'localSymbol','EUR.USD', 'secType','cash', ...
                   'exchange','idealpro')
data =
            reqId: 153745244
          reqTime: '13-Feb-2012 21:28:51'
         dataTime: '13-Feb-2012 21:28:52'
    dataTimestamp: 734912.895051898
           ticker: 'EUR'
         bidPrice: 1.32565
         askPrice: 1.32575
             open: -1
            close: 1.3197
              low: 1.32075
             high: 1.32835
        lastPrice: -1
           volume: -1
             tick: 5e-005
          bidSize: 26000000
          askSize: 20521000
```

[13] The error messages can be suppressed using the MsgDisplayLevel parameter, and can also be trapped and processed using the CallbackMessage parameter – see section 14 below for more details

In other cases, we may also need to explicitly specify the Currency (default='USD') and for options/future we also need to specify the Expiry, Strike and Right parameters. In one particular case that we encountered, it was not even sufficient to specify the Expiry in YYYYMM format because a particular underlying security had two separate futures expiring in the same month:

```
>> data = IBMatlab('action','query','symbol','TNA','secType','opt',...
                   'expiry','201202','strike',47,'right','CALL')

 [API.msg2] The contract description specified for TNA is ambiguous;
you must specify the multiplier. {149386474, 200}
```

The solution in this particular case was simply to specify the exact Expiry in YYYYMMDD format (i.e., specifying the full date rather than just the month).

Here is the full list of Contract properties supported by *IBMatlab*:[14]

Parameter	Data type	Default	Description
Symbol	string	(none)	The symbol of the underlying asset
LocalSymbol	string	''	The local exchange symbol of the underlying asset. When left empty, IB sometimes tries to infer it from Symbol and the other properties.
SecType	string	'STK'	One of: 'STK', 'OPT', 'FUT', 'IND', 'FOP', 'CASH', 'BAG'
Exchange	string	'SMART'	The exchange that should process the request. SMART uses IB's SmartRouting to optimize order execution time and cost.[15]
Currency	string	'USD'	The currency to be used. Possible ambiguities may mean that this field must be specified. For example, when Exchange='SMART' and Symbol='IBM', then Currency must be explicitly specified since IBM can trade in either GBP or USD. In light of these potential ambiguities, it is a good idea to always specify Currency.
Expiry	string	''	'YYYYMM' or 'YYYYMMDD' format
Strike	number	0.0	The strike price (for options)
Right	string	''	One of: 'P', 'PUT', 'C', 'CALL'

Please note that numbers should not be enclosed with quote marks when specifying parameter values. Therefore, we should specify `IBMatlab(..., 'Strike',5.20)` and not `IBMatlab(..., 'Strike','5.20')`. Otherwise, Matlab will get confused when trying to interpret the string '5.20' as a number, and very odd results might happen.

[14] This list closely mirrors IB's Java API list of Contract details: http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/java/contract.htm. Some of the Java API properties mentioned in the online list are not supported by *IBMatlab*, but they can still be accessed via **IB-Matlab**'s Java connector object, as described in section 15 below.

[15] http://www.interactivebrokers.com/en/p.php?f=smartRouting

4 Querying account and portfolio data

IB user accounts have numerous internal properties/parameters, ranging from the account currency to cash balance, margin information etc. You can retrieve this information in Matlab using the following simple command:

```
>> data = IBMatlab('action','account_data')

data =
           AccountCode: 'DU12345'
              currency: []
           accountName: 'DU12345'
          AccountReady: 'true'
           AccountType: 'INDIVIDUAL'
           AccruedCash: -456.4
       AccruedDividend: 0
        AvailableFunds: 261700.68
              Billable: 0
           BuyingPower: 779656.96
           CashBalance: -825400.37
    CorporateBondValue: 0
              Currency: 'USD'

   (and so on ... - dozens of different account parameters)
```

As can be seen, the returned `data` object is a simple Matlab struct, whose fields match the IB account properties. To access any specific field, use the standard Matlab dot notation:

```
>> myBuyingPower = data.BuyingPower; %=779656.96 in this specific case
```

If your TWS account is linked to more than one IB account (as is commonly the case for Financial Advisors, for example), then you may need to specify the optional AccountName input parameter, so that IB would know which IB account you wish to access:

```
>> data = IBMatlab('action','account_data', 'AccountName','DU12345');
```

To retrieve an IB account's portfolio (list of held securities), run a similar command in Matlab, with a 'portfolio' (or 'portfolio_data') action:

```
>> data = IBMatlab('action','portfolio')
data =
1x12 struct array with fields:
    symbol
    localSymbol
    exchange
    secType
    currency
    right
    expiry
    strike
    position
    marketValue
    marketPrice
    averageCost
```

This returns a Matlab array of structs, where each struct element in the array represents a different security held in the IB account. For example:

```
>> data(1)
ans =
          symbol: 'AMZN'
    localSymbol: 'AMZN'
       exchange: 'NASDAQ'
        secType: 'STK'
       currency: 'USD'
          right: '0'
         expiry: ''
         strike: 0
       position: 9200
    marketValue: 1715800
    marketPrice: 186.5
    averageCost: 169.03183335
```

In some cases, IB might return empty data in response to account/portfolio requests. Two workarounds have been found for this, although they might not cover all cases.[16] The workarounds are to simply re-request the information, and to force a programmatic reconnection to IB (more on the Java connector in section 15 below):

```
data = IBMatlab('action','portfolio');
if isempty(data)   % empty data - try to re-request the same data
   [data, ibConnectionObject] = IBMatlab('action','portfolio');
end
if isempty(data)   % still empty data - try to disconnect/reconnect
   ibConnectionObject.disconnectFromTWS;   % disconnect from IB
   sleep(1);                               % let IB cool down a bit
   data = IBMatlab('action','portfolio');  % will automatically reconnect
end
```

[16] For example, the IB API has a known limitation that it does not return the portfolio position if the clearing house is not IB

5 Querying current market data

Let us start with a simple example where we retrieve the current market information for Google Inc., which trades using the GOOG symbol on IB's SMART exchange (the default exchange), with USD currency (the default currency):

```
>> data = IBMatlab('action','query', 'symbol','GOOG')
 data =
        reqId: 22209874
      reqTime: '02-Dec-2010 00:47:23'
     dataTime: '02-Dec-2010 00:47:23'
dataTimestamp: 734474.032914491
       ticker: 'GOOG'
     bidPrice: 563.68
     askPrice: 564.47
         open: 562.82
        close: 555.71
          low: 562.4
         high: 571.57
    lastPrice: -1
       volume: 36891
         tick: 0.01
      bidSize: 3
      askSize: 3
     lastSize: 0
```

As can be seen, the returned `data` object is a Matlab struct whose fields are self-explanatory. To access any specific field, use the standard Matlab notation:

```
>> price = data.bidPrice;  %=563.68 in this specific case
```

To retrieve the current market data, three pre-conditions must be met:

1. The IB account must subscribe to the service for retrieving information for the specified security
2. The specified security can be found on the specified exchange using the specified classification properties (a.k.a. *Contract*)
3. The security is currently traded (i.e., its market is currently open)

If any of these conditions are not met, the information returned by IB will be invalid and an error message will be displayed in the Matlab command window,[17] as the following examples illustrate:

[17] The error messages can be suppressed using the MsgDisplayLevel parameter, and can also be trapped and processed using the CallbackMessage parameter – see section 14 below for more details

```
>> data = IBMatlab('action','query', 'symbol','GOOG');

 [API.msg2] Requested market data is not subscribed.Error&BEST/STK/Top
{153745220, 354}

data =
    dateTime: {1x0 cell}
        open: []
        high: []
         low: []
       close: []
      volume: []
       count: []
         WAP: []
     hasGaps: []
```

This illustrates a situation where we are not subscribed to data for this specific security type and/or exchange. A similar yet different message is returned when we try to get historical data for an unsubscribed security type/exchange:

```
>> data = IBMatlab('action','history', 'symbol','GOOG');

 [API.msg2] Historical Market Data Service error message:No market data
permissions for NASDAQ STK {153745241, 162}
```

If we specify an incorrect security name or classification properties, then the data is similarly not returned, and an error message is displayed (see the discussion in section 3 above).

Additional market data about a security can be retrieved by using IB's Generic Tick List mechanism, which is accessed via the GenericTickList parameter. This parameter is a **string** (default=" =empty) that accepts comma-separated integers such as '236' or '100,101'.[18] Note that the GenericTickList value should be a string ('236'), not a number (236).

```
>> data = IBMatlab('action','query', 'symbol','GOOG', ...
                   'GenericTickList','100');  % Note: '100', not 100!
```

One of the useful tick-types is 236, which returns information about whether or not the specified security is shortable.[19] Only some securities and exchanges support this feature (mainly US stocks), and only for streaming quotes[20] (not for regular market queries). When the feature is supported, an additional `shortable` field is returned with basic information about the security's shortability.[21]

---

[18] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/tables/generic_tick_types.htm

[19] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#using_the_shortable_tick.htm

[20] See section 7 for details on streaming quotes

[21] See section 11.3 for details about the shortable mechanism, with a full working example that uses callbacks

6 Historical data

Historical data can be retrieved from IB, subject to your account's subscription rights, and IB's lengthy list of pacing violation limitations.[22] At the moment (February 2012), these limitations include the following:

1. Historical data can only be requested for the past year – if more than this is requested, the entire request is dropped
2. Historical data is limited to 2000 results (data bars) – if more than that is requested, the entire request is dropped
3. Identical historical data requests within 15 seconds is prohibited (this is automatically prevented by **IB-Matlab**, which returns the previous results in such a case)
4. Requesting 6+ historical data requests for the same Contract, Exchange and Tick Type within 2 seconds is prohibited.
5. Requesting 60+ historical data requests of any type within 10-minutes is prohibited.
6. Only certain combinations of bar Duration and Size are supported:

Duration	Bar Size
1 Y	1 day
6 M	1 day
3 M	1 day
1 M	1 day, 1 hour
1 W	1 day, 1 hour, 30 mins, 15 mins
2 D	1 hour, 30 mins, 15 mins, 3 mins, 2 mins, 1 min
1 D	1 hour, 30 mins, 15 mins, 5 mins, 3 mins, 2 mins, 1 min, 30 secs
14400 S (4 hours)	1 hour, 30 mins, 15 mins, 5 mins, 3 mins, 2 mins, 1 min, 30 secs, 15 secs
7200 S (2 hours)	1 hour, 30 mins, 15 mins, 5 mins, 3 mins, 2 mins, 1 min, 30 secs, 15 secs, 5 secs
3600 S (1 hour)	15 mins, 5 mins, 3 mins, 2 mins, 1 min, 30 secs, 15 secs, 5 secs
1800 S (30 mins)	15 mins, 5 mins, 3 mins, 2 mins, 1 min, 30 secs, 15 secs, 5 secs, 1 sec
960 S (15 mins)	5 mins, 3 mins, 2 mins, 1 min, 30 secs, 15 secs, 5 secs, 1 sec
300 S (5 mins)	3 mins, 2 mins, 1 min, 30 secs, 15 secs, 5 secs, 1 sec
60 S (1 min)	30 secs, 15 secs, 5 secs, 1 sec

[22] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/api/historical_data_limitations.htm

Other Duration values (that are not specified in the table) are sometimes, but not always, accepted by IB. For example, 60D is accepted, but 61D is not. In such cases, you can always find a valid alternative (3M instead of 61D, for example).

Note that IB-Matlab does not prevent you from entering invalid Durations and Bar Sizes – it is up to you to verify that your specified parameters are accepted by IB. If they are not, then IB will report an error message that will be displayed in the Matlab command window.

Subject to these limitations, retrieval of information in **IB-Matlab** is quite simple. For example, to return the 1-hour bars from the past day:

```
>> data = IBMatlab('action','history', 'symbol','IBM', ...
                   'barSize','1 hour', 'useRTH',1)
data =
       dateTime: {1x7 cell}
           open: [161.08 160.95 161.66 161.17 161.57 161.75 162.07]
           high: [161.35 161.65 161.70 161.60 161.98 162.09 162.34]
            low: [160.86 160.89 161.00 161.13 161.53 161.61 161.89]
          close: [160.93 161.65 161.18 161.60 161.74 162.07 162.29]
         volume: [5384 6332 4580 2963 4728 4465 10173]
          count: [2776 4387 2990 1921 2949 2981 6187]
            WAP: [161.07 161.25 161.35 161.31 161.79 161.92 162.14]
        hasGaps: [0 0 0 0 0 0 0]
```

As can be seen, the returned `data` object is a Matlab struct whose fields are:[23]

- dateTime – a cell-array of date strings, or a numeric array of date values (see the FormatDate parameter, explained below)
- open – the bar's opening price
- high – the high price during the time covered by the bar
- low – the low price during the time covered by the bar
- close – the bar's closing price
- volume – the trading volume during the time covered by the bar
- count – number of trades that occurred during the the time covered by the bar Note: only valid when WhatToShow='Trades' (see below)
- WAP – the weighted average price during the time covered by the bar
- hasGaps – whether or not there are gaps (unreported bars) in the data

[23] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/java/historicaldata.htm#HT_historicalDataw

The fields are Matlab data arrays (numeric arrays for the data and cell-arrays for the timestamps). To access any specific field, use the standard Matlab notation:

```
>> data.dateTime
ans =
    '20110225  16:30:00'   '20110225  17:00:00'   '20110225  18:00:00'
    '20110225  19:00:00'   '20110225  20:00:00'   '20110225  21:00:00'
    '20110225  22:00:00'

>> lastOpen = data.open(end);  % =162.07 in this specific case
```

As noted above, if any of the limitations for historical data requests is not met, then an error message is displayed in the Matlab command prompt and no data is returned. Unfortunately, IB's error messages are not always very descriptive in their explanation of what was actually wrong with the request.

The parameters that affect historical data retrieval closely mirror those expected by the Java API:[24]

Parameter	Data type	Default	Description
EndDateTime	string	'' (empty string), meaning now	Use the format 'YYYYMMDD hh:mm:ss TMZ' (the TMZ time zone is optional[25])
DurationValue	integer	1	Together with DurationUnits this parameter specifies the historical data duration, subject to the limitations on possible Duration/BarSize
DurationUnits	string	'D'	One of: • S (seconds) • D (days) • W (weeks) • M (months) • Y (years)
BarSize	string	'1 min'	Specifies the size of the bars that will be returned (within IB/TWS limits). Valid values include: • 1 sec, 5/15/30 secs • 1 min, 2/3/5/15/30 mins • 1 hour • 1 day

[24] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/java/reqhistoricaldata.htm

[25] The list of time zones accepted by IB is listed in section 9.1 below

Parameter	Data type	Default	Description
WhatToShow	string (case insensitive)	'Trades'	Determines the nature of data being extracted. Valid values include: • Trades (invalid for FOREX) • Midpoint • Bid • Ask • Bid_Ask • Historical_Volatility • Option_Implied_Volatility
UseRTH	integer or logical flag	0 = false	Determines whether to return all data available during the requested time span, or only data that falls within Regular Trading Hours. Valid values include: • 0 or false: all data is returned even where the market in question was outside of its regular trading hours • 1 or true: only data within the regular trading hours is returned, even if the requested time span falls partially or completely outside of the RTH.
FormatDate	integer	1	Determines the date format applied to returned bars. Valid values include: 1) dates applying to bars returned in the format: 'yyyymmdd hh:mm:dd' 2) dates are returned as a long integer (# of seconds since 1/1/1970 GMT)

Note that some securities and exchanges do not support certain historical parameter combinations. For example, FOREX (currency) historical data requests on the IDEALPRO exchange do not support WhatToShow='Trades' but only 'MIDPOINT'. IB displays a very cryptic error message in such cases, and we are only left with the option of guessing what parameter value to modify, or ask IB's customer support.

Also note that some exchanges, while returning the requested historical data, may not provide all of the historical data fields listed above. For example, in the case of FOREX on IDEALPRO again, the Volume, Count and WAP fields are not returned, and appear as arrays of -1 when returned to the user in the `data` struct:

```
>> data = IBMatlab('action','history', 'symbol','EUR', ...
                   'localSymbol','EUR.USD', 'secType','cash', ...
                   'exchange','idealpro', 'barSize','1 day', ...
                   'DurationValue',3, 'WhatToShow','midpoint')
data =
    dateTime: {'20120210'  '20120211'  '20120214'}
        open: [1.32605  1.3286   1.32095]
        high: [1.3321   1.329075 1.328425]
         low: [1.321625 1.315575 1.320275]
       close: [1.328575 1.319825 1.325975]
      volume: [-1 -1 -1]
       count: [-1 -1 -1]
         WAP: [-1 -1 -1]
     hasGaps: [0 0 0]
```

Note that in this specific example, historical daily (BarSize = '1 day') data from the past 3 days have been requested on 2012-02-13 (Monday). However, the received data was for 2012-02-09 (Thursday), 2012-02-10 (Friday) and 2012-02-13 (Monday). Data was not received for 2012-02-11 and 2012-02-12 because the specified security was not traded during the weekend.

Another oddity is that the dates were reported with an offset of 1 (2012-02-10 instead of 2012-02-09 etc.). The reason is that the information is collected on a daily basis and reported as of the first second after midnight, i.e., on the following date. This is indeed confusing, so if you rely on the reported historical data dates in your analysis, then you should take this into consideration.

In some cases, users may be tempted to use the historical data mechanism in order to retrieve real-time data. This is actually relatively easy to set-up. For example, implement an endless loop in Matlab that sleeps for 60 seconds, requests the latest historical data for the past minute and goes to sleep again (more advanced Matlab users would probably implement a recurring timer object that wakes up every minute). In such cases, the user should consider using the streaming quotes mechanism rather than historical quotes. Streaming quotes are the subject of the following section.

7 Streaming quotes

Streaming quotes are a near-real-time mechanism, where IB sends information to **IB-Matlab** about ongoing quote ticks (bids and asks). Note that IB apparently does not send "flat" ticks, i.e., quotes where the price does not change.

IB limits the streaming to about 50 tick messages per second. This rate is ok for up to several dozen highly-traded securities, but not for the entire S&P 500.

The streaming quotes mechanism has two distinct parts:

1. Request IB to start sending the stream of quotes for a specified security. This is done by using Action='query' and QuotesNumber with a positive >1 value.
2. Later, whenever you wish to read the latest quote(s), simply use Action='query' and QuotesNumber= -1 (minus one). This will return the information without stopping the background streaming.

For example, let's request 100 streaming quotes for EUR.USD:

```
>> reqId = IBMatlab('action','query', 'symbol','EUR', ...
                    'localSymbol','EUR.USD', 'currency','USD', ...
                    'secType','cash', 'exchange','idealpro', ...
                    'QuotesNumber',100)
reqId =
   147898050
```

This causes IB to start sending quotes to **IB-Matlab** in the background, up to the specified QuotesNumber, without affecting normal Matlab processing. This means that you can continue to work with Matlab, process/display information etc.

QuotesNumber can be any number higher than 1 for streaming to work (a value of 1 is the standard market-query request that was described in section 5 above). To collect the streaming quotes endlessly, simply set QuotesNumber to the value `inf`. Note that in Matlab, `inf` is a number not a string, so do not enclose it in quotes (`'inf'`) when submitting the request.

Also note that the request to start streaming quotes returns the request ID, not data.

The quotes are collected into an internal data buffer in **IB-Matlab**. A different buffer is maintained for each localSymbol. The buffer size can be controlled using the QuotesBufferSize parameter, which has a default value of 1. This means that by default only the latest streaming quote of each type (bid/ask) is stored, along with high/low/close data.

If you set a higher value for QuotesBufferSize,[26] then up to the specified number of latest bid quotes will be stored (note: only bid quotes are counted here):

```
>> reqId = IBMatlab('action','query', 'symbol','GOOG', ...
                    'QuotesNumber',100, 'QuotesBufferSize',500)
```

Note that using a large QuotesBufferSize increases memory usage, which could have an adverse effect if you use a very large buffer size (many thousands) and/or streaming for a large number of different securities.[27]

Subsequent requests to retrieve the latest accumulated quotes buffer data, without stopping the background streaming, should use QuotesNumber = -1 (minus one). These requests return a data struct that looks like this:

```
>> dataStruct = IBMatlab('action','query', 'localSymbol','EUR.USD', ...
                         'QuotesNumber',-1)
dataStruct =
               reqId: 147898050
              symbol: 'EUR'
         localSymbol: 'EUR.USD'
            isActive: 1
      quotesReceived: 6
     quotesToReceive: 10
    quotesBufferSize: 1
     genericTickList: ''
                data: [1x1 struct]
            contract: [1x1 com.ib.client.Contract]
```

Note that you only need to specify the Symbol/LocalSymbol and the QuotesNumber in the subsequent requests[28] – all other parameters are unnecessary since the system already knows about this symbol's parameters from the initial streaming request. This is useful and easy to use, but also means that you cannot have two simultaneous streams for the same LocalSymbol, even using different parameters.

In the returned `dataStruct`, we can see the following fields:

- `reqId` – this is the request ID for the original streaming request, the same ID that was returned by *IBMatlab* when we sent our initial request
- `symbol`, `localSymbol` – determine the underlying security being streamed
- `isActive` – a logical flag indicating whether quotes are currently being streamed for the security. When QuotesNumber bid quotes have been received, this flag is set to false (0).

[26] QuotesBufferSize is a numeric parameter like QuotesNumber, so do not enclose the parameter value within quotes

[27] Quotes use about 1.5KB of Matlab memory. So, if QuotesBufferSize=1500, then for 20 symbols **IB-Matlab** would need 20*1500*1.5KB = 45MB of Matlab memory when all 20 buffers become full (which could take a while).

[28] **IB-Matlab** versions since 2012-01-15 only need to use LocalSymbol; earlier versions of **IB-Matlab** used Symbol to store the streaming data. This means that the earlier versions cannot stream EUR.USD and EUR.JPY simultaneously, since they both have the same symbol (EUR). In practice, most stocks have Symbol = LocalSymbol so this distinction does not really matter.

- `quotesReceived` – total number of streaming bid quotes received for this security
- `quotesToReceive` – total number of streaming bid quotes requested for the security (=QuotesNumber parameter). When `quotesReceived` >= `quotesToReceive`, streaming quotes are turned off and `isActive` is set to false (0). Note that it is possible that `quotesReceived` > `quotesToReceive`, since it takes a short time for the streaming quotes cancellation request to reach IB, and during this time it is possible that new real-time quotes have arrived.
- `quotesBufferSize` – the size of the data buffer (=QuotesBufferSize parameter)
- `genericTickList` – any GenericTickList requested in the initial request will be kept here for possible reuse upon resubscribing to the streaming quotes (see the ReconnectEvery parameter described below).
- `contract` – a Java object that holds the definition of the security, for possible reuse upon resubscribing to the streaming quotes.
- `data` – this is a sub-struct that holds the actual buffered quotes data

To get the actual quotes data, simply read the `data` field of this `dataStruct`:

```
>> dataStruct.data
ans =
        dataTimestamp: 734892.764653854
                 high: 1.3061
        highTimestamp: 734892.762143183
                  low: 1.29545
         lowTimestamp: 734892.762143183
                close: 1.30155
       closeTimestamp: 734892.762143183
             bidPrice: 1.2986
    bidPriceTimestamp: 734892.764653854
              bidSize: 1000000
     bidSizeTimestamp: 734892.764653854
             askPrice: 1.29865
    askPriceTimestamp: 734892.764499421
              askSize: 18533000
     askSizeTimestamp: 734892.764653854
```

Note that each data item has an associated timestamp, because different data items are sent separately from the IB server. You can convert the timestamps into human-readable string by using Matlab's ***datestr*** function, as follows:

```
>> datestr(dataStruct.data.dataTimestamp)
ans =
24-Jan-2012 23:56:32
```

The `dataTimestamp` field currently holds the same information as `bidPriceTimestamp`. Future versions of **IB-Matlab** may modify this to indicate the latest timestamp of any received quote, not necessarily a bid quote.

If instead of using QuotesBufferSize=1 (which is the default value), we had used QuotesBufferSize=3, then we would see not the latest quote but the latest 3 quotes:

```
>> reqId = IBMatlab('action','query', 'symbol','EUR', ...
                    'localSymbol','EUR.USD', 'currency','USD', ...
                    'secType','cash', 'exchange','idealpro', ...
                    'QuotesNumber',10, 'QuotesBufferSize',3);

% now at any following point in time run the following command to get
the latest 3 quotes:

>> dataStruct = IBMatlab('action','query', 'localSymbol','EUR.USD', ...
                         'QuotesNumber',-1);
>> dataStruct.data
ans =
    dataTimestamp: [734892.99760235 734892.99760235 734892.997756076]
             high: 1.3061
    highTimestamp: 734892.99740162
              low: 1.29545
     lowTimestamp: 734892.99740162
         bidPrice: [1.30355 1.3035 1.30345]
bidPriceTimestamp: [734892.99760235 734892.99760235 734892.997756076]
          bidSize: [2000000 4000000 4000000]
 bidSizeTimestamp: [734892.997756076 734892.997756076 734892.997756076]
         askPrice: [1.30355 1.3036 1.30355]
askPriceTimestamp: [734892.997667824 734892.997667824 734892.997756076]
          askSize: [3153000 2153000 4153000]
 askSizeTimestamp: [734892.997756076 734892.997756076 734892.997756076]
            close: 1.30155
   closeTimestamp: 734892.997407037
```

Note that the high, low and close fields are only sent once by the IB server, as we would expect. Only the bid and ask information is sent as a continuous stream of data from IB. Also note how each of the quote values has an associated timestamp.

To stop collecting streaming quotes for a security, simply send the request again, this time with QuotesNumber=0. The request will return the `dataStruct` with the latest data that was accumulated up to that time.

Another parameter that can be used for streaming quotes in **IB-Matlab** attempts to bypass a problem that sometimes occurs with high-frequency streams. In such cases, it has been reported that after several thousand quotes, IB stops sending streaming quotes data, without any reported error message. The ReconnectEvery numeric parameter (default=2000) controls the number of quotes (total of all streaming

securities) before IB-trade automatically reconnects to IB and re-subscribes to the streaming quotes. You can specify any positive numeric value, or `inf` to accept streaming quotes without any automated reconnection.

In summary, here are the parameters that directly affect streaming quotes in *IBMatlab*:

Parameter	Data type	Default	Description
QuotesNumber	integer	1	One of: • inf – continuous endless streaming quotes for the specified security • N>1 – stream only N quotes • 1 – get only a single quote (i.e., non-streaming snapshot) • 0 – stop streaming quotes • -1 – return the latest accumulated quotes data while continuing to stream new quotes data
QuotesBufferSize	integer	1	Controls the number of streaming quotes stored for user retrieval in a cyclic buffer. When this number of quotes have been received, the oldest quote will be discarded whenever a new quote arrives.
ReconnectEvery	integer	2000	Number of quotes (total of all securities) before automated reconnection to IB and re-subscription to the streaming quotes. • inf – accept streaming quotes without automated reconnection • N>0 – automatically reconnect and re-subscribe to streaming quotes after N quotes are received.

8 Sending trade orders

Three classes of trade orders are supported in **IB-Matlab**: Buy, Sell, and Short. These are implemented in *IBMatlab* using the corresponding values for the Action parameter: 'Buy', 'Sell', and 'SShort'.

Several additional *IBMatlab* parameters affect trade orders. The most widely-used properties are Type (default='LMT'), Quantity, and LimitPrice. Additional properties are explained below. Here is a simple example for buying and selling a security:

```
orderId = IBMatlab('action','BUY', 'symbol','GOOG', 'quantity',100,...
                   'type','LMT', 'limitPrice',600);

orderId = IBMatlab('action','SELL', 'symbol','GOOG', 'quantity',100,...
                   'type','LMT', 'limitPrice',600);
```

In this example, we have sent an order to Buy/Sell 100 shares of GOOG on the SMART exchange, using an order type Limit and limit price of USD 600. *IBMatlab* returns the corresponding orderId assigned by IB – we can use this orderID later to modify open orders, cancel open orders, or follow-up in TWS or in the trade logs.

IB's API supports a long list of order types. Unfortunately, many of these may not be available on your TWS and/or for the requested exchange and security type.[29] You need to carefully ensure that the order type is accepted by IB before using it in *IBMatlab*. Here is the full list of order types supported by *IBMatlab*, which is a subset of the list in IB's documentation:[30]

Class	Order type full name	Order type abbreviation	Description
Limit risk	Limit	LMT	Buy or sell a security at a specified price or better.
	Market-to-Limit	MTL	A Market-To-Limit order is sent as a Market order to execute at the current best price. If the entire order does not immediately execute at the market price, the remainder of the order is re-submitted as a Limit order with the limit price set to the price at which the market order portion of the order executed.
	Market with Protection	MKT PRT	A Market With Protection order is sent as a Market order to execute at the current best price. If the entire order does not immediately execute at the market price, the remainder of the order is re-submitted as a Limit order with the limit price set by Globex to a price slightly higher/lower than the current best price

[29] http://www.interactivebrokers.com/en/p.php?f=orderTypes

[30] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/tables/supported_order_types.htm

Class	Order type full name	Order type abbreviation	Description
Limit risk	Stop	STP	A Stop order becomes a Market order to buy (sell) once market price rises (drops) to the specified trigger price
	Stop Limit	STP LMT	A Stop Limit order becomes a Limit order to buy (sell) once the market price rises (drops) to the specified trigger price
	Trailing Limit if Touched	TRAIL LIT	A Trailing Limit-If-Touched sell order sets the trigger price at a fixed amount (or %) above the market price. If the market price falls, the trigger price falls by the same amount; if the market price rises, the trigger price remains unchanged. If the price rises all the way to the trigger price, the order is submitted as a Limit order. (*and vice versa for buy orders*)
	Trailing Market If Touched	TRAIL MIT	A Trailing Market-If-Touched sell order sets a trigger price at a fixed amount (or %) above the market price. If the market price falls, the trigger price falls by the same amount; if the market price rises, the trigger price remains unchanged. If the price rises all the way to the trigger price, the order is submitted as a Market order. (*and vice versa for buy orders*)
	Trailing Stop	TRAIL	A Trailing Stop sell order sets the stop trigger price at a fixed amount (or %) below the market price. If the market price rises, the stop trigger price rises by the same amount; if the market price falls, the trigger price remains unchanged. If price falls all the way to trigger price, the order is submitted as a Market order (*and vice versa for buy orders*)
	Trailing Stop Limit	TRAIL LIMIT	A Trailing Stop Limit sell order sets the stop trigger price at a fixed amount (or %) below the market price. If the market price rises, the stop trigger price rises by the same amount; if the market price falls, the trigger price remains un-changed. If the price falls all the way to the trigger price, the order is submitted as a Limit order. (*and vice versa for buy orders*)
Execute speed	Market	MKT	An order to buy (sell) a security at the offer (bid) price currently available in the marketplace. There is no guarantee that the order will fully or even partially execute at any specific price.

Class	Order type full name	Order type abbreviation	Description
Execution speed	Market-if-Touched	MIT	A Market-if-Touched order becomes a Market order to buy (sell) once the market price drops (rises) to the specified trigger price.
	Market-on-Close	MOC	Market-on-Close will execute as a Market order during the market close time, as close to the closing price as possible.
	Pegged-to-Market	PEG MKT	A Limit order whose price adjusts automatically relative to market price, using specified offset amount
	Relative	REL	An order whose price is dynamically derived from the current best bid (offer) in the marketplace. For a buy order, the price is calculated by adding the specified offset (or %) to the best bid. A limit price may optionally be entered to specify a cap for the amount you are willing to pay. (*and vice versa for sell orders*)
Price Improvement	Box Top	BOX TOP	A Market order that is automatically changed to Limit order if it does not execute immediately at market price
	Limit-on-Close	LOC	Limit-on-close will execute at the market close time, at the closing price, if the closing price is at or better than the limit price, according to the exchange rules; Otherwise the order will be cancelled.
	Limit if Touched	LIT	A Limit-if-Touched order becomes a Limit order to buy (sell) once the market price drops (rises) to the specified trigger price.
	Pegged-to-Midpoint	PEG MID	A Limit order whose price adjusts automatically relative to midpoint price using specified offset amount
	VWAP - best efforts	VWAP	Achieves the Volume-Weighted Average Price on a best-effort basis (see details in section 9.1 below).
	VWAP - guaranteed	Guarrante-edVWAP	The VWAP for a stock is calculated by adding the dollars traded for every transaction in that stock ("price" x "number of shares traded") and dividing the total shares traded. By default, a VWAP order is computed from the open of the market to the market close, and is calculated by volume weighting all transactions during this time period. TWS allows you to modify the cut-off and expiration times using the Time in Force and Expiration Date fields respectively

In addition to specifying the order Type, Quantity and LimitPrice, several other parameters may need to be specified to fully describe the order.

Here is a summary of the parameters that directly affect trade orders in *IBMatlab*:

Parameter	Data type	Default	Description
Action	string	(none)	One of: 'Buy', 'Sell', or 'SShort' (note the double-S in SShort)
Quantity	integer	0	Number of requested shares
Type	string	'LMT'	Refer to the order-types table above
LimitPrice	number	0	Limit price used in Limit order types
AuxPrice	number	0	Extra numeric data used by some order types (e.g., STP uses AuxPrice, not LimitPrice)
TIF	string	'GTC'	Time-in-Force. One of: • GTC – Good Till Cancelled • GTD – Good Till Date • IOC – Immediate-fill or Cancelled • Day – Good until end of trading day • DTC – Day Till Cancelled
OCAGroup	string	''	One-Cancels-All group name. This can be specified for several trade orders so that whenever one of them gets cancelled or filled, the others get cancelled automatically.
ParentId	integer	0	Useful for setting child orders of a parent order: these orders are only active when their parent order is active or gets triggered
TrailStopPrice	number	0	The stop price used when Type='TrailLimit'
GoodAfterTime	string	''	Format: 'YYYYMMDD hh:mm:ss TMZ' (TMZ is optional[31])
GoodTillDate	string	''	Format: 'YYYYMMDD hh:mm:ss TMZ' (TMZ is optional[32])
TriggerMethod	integer	0	One of:[33] • 0=Default • 1=Double-Bid-Ask • 2=Last • 3=Double-Last • 4=Bid-Ask • 7=Last-or-Bid-Ask • 8=Mid-point

[31] The list of time zones accepted by IB is listed in section 9.1 below

[32] The list of time zones accepted by IB is listed in section 9.1 below

Parameter	Data type	Default	Description
BracketDelta	number	[]=empty	Price offset for stop-loss and take-profit bracket child orders (see section 9.3 below). Note: BracketDelta may be a single value or a [lowerDelta,upperDelta] pair of values Note: value(s) must be positive: - low bracket will use limitPrice – lowerDelta - high bracket will use limitPrice + upperDelta
BracketTypes	cell array of 2 strings	Buy: {'STP', 'LMT'} Sell: {'LMT', 'STP'}	Types of child bracket orders. The first string in the cell array defines the order type for the lower bracket; the second string defines the order type for the upper bracket. See section 9.3 below for additional details.
OutsideRTH	integer or logical flag	0=false	• 0 or false: order should not execute outside regular trading hours • 1 or true: order can execute outside regular trading hours if required
OrderId	integer	(*auto-assigned*)	If specified, and if the specified OrderId is still open, then the specified order data will be updated, rather than creating a new order. This enables users to modify the order Type, LimitPrice and other important parameters of existing open orders (see section 10 below).
AccountName	string	"	The specific IB account used for this trade. Useful when you handle multiple IB accounts, otherwise leave empty.
FAProfile	string	"	The Financial Advisor allocation profile to which trades will be allocated. Only relevant for Financial Advisor accounts, otherwise leave empty.
Hold	integer or logical flag	0=false	• 0 or false: order will be sent to IB immediately • 1 or true: order will be prepared but not sent to IB. The user can them modify the order properties before sending to IB. See section 9.5 below for additional details.

[33] http://www.interactivebrokers.com/en/software/tws/usersguidebook/configuretws/modify_the_stop_trigger_method.htm

9 Specialized trade orders

Several specialized order types are supported by *IBMatlab*, each of which has dedicated parameters for configuration. These order types include VWAP (best effort), TWAP, bracket orders, and automated orders.

9.1 VWAP (best-effort) orders

When the order Type is 'VWAP' (the best-effort type, since the guaranteed type has Type='GuaranteedVWAP'), IB treats the order as a Market order with a VWAP algo strategy.[34] *IBMatlab* enables specifying the algo strategy's properties, as follows:

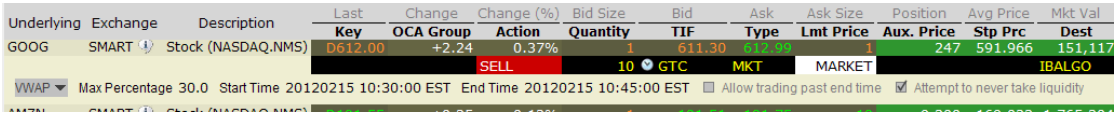
Parameter	Data type	Default	Description
MaxPctVol	number	0.1=10%	Percent of participation of average daily volume up to 0.5 (=50%).
StartTime	string	'9:00:00 EST'	Format: 'YYYYMMDD hh:mm:ss TMZ' (TMZ is optional)
EndTime	string	'16:00:00 EST'	(*same as StartTime above*)
AllowPastEndTime	integer or logical flag	1=true	If true, allow the algo to continue to work past the specified end time if the full quantity has not been filled.
NoTakeLiq	integer or logical flag	0=false	If true, discourage the VWAP algo from hitting the bid or lifting the offer if possible. This may help to avoid liquidity-taker fees, and could result in liquidity-adding rebates. But it may also result in greater deviations from the benchmark and partial fills, since the posted bid/offer may not always get hit as the price moves up/down. IB will use best efforts not to take liquidity, however, there will be times that it cannot be avoided.

Here is an example for specifying a best-effort VWAP trade order:

```
orderId = IBMatlab('action','SELL', 'symbol','GOOG', 'quantity',10, ...
                   'type','VWAP', 'limitPrice',600, 'MaxPctVol',0.3, ...
                   'StartTime','20120215 10:30:00 EST', ...
                   'EndTime',  '10:45:00 EST', ...
                   'AllowPastEndTime',false, ...
                   'NoTakeLiq',true);
```

[34] http://www.interactivebrokers.com/en/trading/orders/vwapAlgo.php; http://www.interactivebrokers.com/en/software/tws/twsguide_Left.htm#CSHID=usersguidebook%2Falgos%2Ftwap.htm|StartTopic=usersguidebook%2Falgos%2Fvwap.htm|SkinName=ibskin

When we run the command above in Matlab, we see the following in IB's TWS:



Note that IB automatically routes the trade to its internal servers (IBALGO) rather than directly to the relevant exchange as it would do in most other cases. Also note that the VWAP order is NOT guaranteed to execute. Best-effort VWAP algo orders result in lower commissions than the Guaranteed VWAP, but the order may not fully execute and is not guaranteed, so if you need to ensure this, use Guaranteed VWAP.

StartTime and EndTime dictate when the VWAP algo will begin/end working, regardless of whether or not the entire quantity has been filled. The End Time supersedes the TIF (time in force) parameter. Note that the order will automatically be cancelled at the designated EndTime regardless of whether the entire quantity has filled unless AllowTradingPastEndTime=1. If an EndTime is specified, then set AllowTradingPastEndTime=1 (or true) to allow the VWAP algo to continue to work past the specified EndTime if the full quantity has not been filled.

Note: If you specify and StartTime and EndTime, TWS confirms that acceptability of the time period using yesterday's trading volume. If the time period you define is too short, you will receive a message with recommended time adjustments.

In the example above, note the optional date (`20120215`) in StartTime. In the EndTime parameter, no date was specified, so today's date will be used, at 10:45:00 EST.

The time-zone part is also optional, but we strongly recommend specifying it, to prevent ambiguities. Not all of the world's time zones are accepted, but some of the major ones are, and you can always convert a time to one of these time zones. The full list of time-zones supported by IB is given below: [35]

Time zone supported by IB	Description
GMT	Greenwich Mean Time
EST	Eastern Standard Time
MST	Mountain Standard Time
PST	Pacific Standard Time
AST	Atlantic Standard Time
JST	Japan Standard Time
AET	Australian Standard Time

IB only enables VWAP algo orders for US equities.

[35] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#supported_time_zones.htm

9.2 TWAP (best-effort) orders

When the order Type is 'TWAP', IB treats the order as a Limit order with a TWAP algo strategy.[36] *IBMatlab* enables specifying the algo strategy's properties, as follows:

Parameter	Data type	Default	Description
StrategyType	string	'Marketable'	One of: • 'Marketable' • 'Matching Midpoint' • 'Matching Same Side' • 'Matching Last'
StartTime	string	'9:00:00 EST'	Format: 'YYYYMMDD hh:mm:ss TMZ' (TMZ is optional)
EndTime	string	'16:00:00 EST'	(*same as StartTime above*)
AllowPastEndTime	integer or logical flag	1=true	If true, allow the algo to continue to work past the specified end time if the full quantity has not been filled.

Note: StartTime, EndTime and AllowPastEndTime were described in section 9.1.

Here is an example for specifying a TWAP trade order:

```
orderId = IBMatlab('action','SELL', 'symbol','GOOG', 'quantity',10, ...
                   'type','TWAP', 'limitPrice',600, ...
                   'StrategyType','Matching Last', ...
                   'StartTime','20120215 10:30:00 EST', ...
                   'EndTime',  '10:45:00 EST', ...
                   'AllowPastEndTime',false);
```

Note that, as with VWAP, IB automatically routes the trade to its internal servers (IBALGO) rather than directly to the relevant exchange as it would do in most other cases. Also note that the TWAP order is NOT guaranteed to execute. The order will trade if and when the StrategyType criterion is met.

IB only enables TWAP algo orders for US equities.

[36] http://www.interactivebrokers.com/en/trading/orders/twapAlgo.php; http://www.interactivebrokers.com/en/software/tws/twsguide_Left.htm#CSHID=usersguidebook%2Falgos%2Ftwap.htm|StartTopic=usersguidebook%2Falgos%2Ftwap.htm|SkinName=ibskin

9.3 Bracket orders

Bracket orders are trades which aim to limit losses while locking-in profits, by sending two opposite-side child orders to offset a parent order.[37] This mechanism ensures that the child orders are made active only when the parent order executes.

Both of the bracket child orders have the same amount as the parent order, and belong to the same OCA (One-Cancels-All) group, so that if one of the child orders is triggered and gets executed, the opposing order is automatically cancelled. Similarly, canceling the parent order will automatically cancel all its child orders.

Buy orders are bracketed by a high-side sell Limit (Type=LMT) order and a low-side sell Stop (Type=STP) order; Sell orders are bracketed by a high-side buy Stop order and a low side buy Limit order.

In IB-Matlab, brackets can only be assigned to parent Buy or Sell orders having Type=LMT or LMTSTP. Specifying bracket orders is very simple, using the BracketDelta parameter. This parameter (default=[] = empty) accepts a single number value or an array of two numeric values, which specify the offset from the parent order's LimitPrice. If BracketDelta is 2-value array [lowerDelta,upperDelta], then lowerDelta is used as the offset for the lower child, and upperDelta is used for the upper child; if BracketDelta is a single number, then this offset is used for both child orders. The corresponding child order limits will be set to LimitPrice-lowerDelta and LimitPrice+upperDelta, respectively.

IBMatlab returns the orderId of the parent order; the child orders have order IDs that are orderId+1 and orderId+2, respectively.

For example, the following trade order:

```
parentOrderId = IBMatlab('action','BUY', 'symbol','GOOG', ...
                         'quantity',100, 'type','LMT', ...
                         'limitPrice',600, 'BracketDelta',[20,50]);
```

Will result in the following situation in IB:

Underlying	Exchange	Description	Last	Change	Change (%)	Bid Size	Bid	Ask	Ask Size	Position
			Key	OCA Group	Action	Quantity	Time in Force	Type	Lmt Price	Aux. Price
GOOG	SMART	Stock (NASDAQ.NMS)	D612.20	+2.44	0.40%	1	611.30	614.51	1	247
			4.2	989560904	SELL	100	GTC	LMT	650.00	
			4.1	989560904	SELL	100	GTC	STP		580.00
			4		BUY	100	GTC	LMT	600.00	

In this screenshot, notice that the parent order is shown as active (blue; IB status: "*Order is being held and monitored*") at the bottom. This order has a Last-Key value of "4" and is a simple Buy at Limit 600 order.

[37] http://www.interactivebrokers.com/en/trading/orders/bracket.php, http://ibkb.interactivebrokers.com/node/1043

The child orders are shown above their parent as inactive (red; IB status: "*Waiting for parent order to fill*"). These orders are of Type=LMT (for the 650 take-profit order) and STP (for the 580 stop-loss order). Note that the child orders have a Last-Key value that derives from their parent (4.1, 4.2 respectively) and the same OCA group name, which is automatically generated based on the order timestamp.

It is possible to specify child bracket orders of different types than the default LMT and STP. This can be done using the BracketTypes parameter. For example, to set an upper bracket of type MIT (Market-If-Touched) rather than LMT for the preceding example, we could do as follows:

```
parentOrderId = IBMatlab('action','BUY', 'symbol','GOOG', ...
                         'quantity',100, 'type','LMT', ...
                         'limitPrice',600, 'BracketDelta',[20,50], ...
                         'BracketTypes',{'STP','MIT'});
```

Underlying	Exchange	Description	Last	Change	Chg %	Bid Size	Bid	Ask	Ask Size	Position
			Key	OCA Group	Action	Quantity	Time in Force	Type	Lmt Price	Trigger Price
GOOG	SMART	Stock (NASDAQ.NMS)	D613.10	+3.34	0.55%	5	612.78	613.96	1	247
			10.2	989560918	SELL	100	GTC	MIT		650.00
			10.1	989560918	SELL	100	GTC	STP		580.00
			10		BUY	100	GTC	LMT	600.00	

Another method to achieve this modification would be to use the relevant child order ID (which is parentOrderId+2 for the upper child) and modify its type from LMT to MIT (see section 10.2 below for details).

9.4 Automated orders

Automated orders are similar to orders of types REL and TRAIL. The idea is to modify a Limit order's LimitPrice based on instantaneous market bid and ask quotes plus (or minus) a certain number of security tick value. At a certain point in time, the order, if not fulfilled or cancelled by then, can automatically be transformed from LMT to some other type (e.g., MKT).

IBMatlab implements automated orders using a timer that periodically checks the latest bid/ask quotes for the specified security and modifies the order's LimitPrice (and possibly the order Type) accordingly.

Unlike IB's REL and TRAIL order types (and their variants, e.g., TRAIL MIT etc.), which update the LimitPrice continuously, *IBMatlab*'s automated orders are only updated periodically. This could be problematic in highly-volatile securities – in such cases users should use IB's standard REL and TRAIL. However, for low-volatility securities, the flexibility offered by *IBMatlab*'s automated orders could be beneficial.

The parameters that affect automated orders in *IBMatlab* are the following:

Parameter	Data type	Default	Description
LimitBasis	string	(*none*)	One of: 'BID', 'ASK'
LimitDelta	integer	0	Units of the security's minimal tick value
LimitRepeatEvery	number	0	Update timer period in seconds
LimitChangeTime	string	(*now+10 hrs*)	Time at which to change the order type automatically, if it was not fulfilled or cancelled by then. Format: 'YYYYMMDD hh:mm:ss' local time
LimitChangeType	string	'MKT'	The new order type to be used at LimitChangeTime
Tick	number	0	Override the security's reported tick value, used by LimitDelta. This is useful for securities/exchanges that do not report a valid tick in market queries (see section 5 above).

Note: using *IBMatlab*'s automated orders, implied by setting a non-empty LimitBasis parameter value, automatically sets the order type to LMT, regardless of the original order Type requested by the user.

For example, the tick value for GOOG is 0.01. To send a Limit BUY order, that is updated to BID - 2 ticks (i.e., BID - 0.02) every 15 seconds, run the following:

```
orderId=IBMatlab('action','BUY', 'symbol','GOOG', 'quantity',100, ...
                 'type','LMT', 'limitPrice',600, 'LimitBasis','BID',...
                 'LimitDelta',-2, 'LimitRepeatEvery',15);
```

9.5 Setting special order attributes

Most of the important order parameters that are supported by IB are also supported as *IBMatlab* parameters. However, IB also supports additional properties that in some cases may be important.

For example, we may wish to specify the security identifier (via the contract object's m_secIDType and m_secId properties[38]), or to specify the All-or-None flag (via the order object's m_allOrNone property[39]).

These properties are not available as *IBMatlab* parameters, but they can still be specified in **IB-Matlab** using either of two methods:

- We can use the `ibConnectionObject` Java object returned by *IBMatlab* as a second output value, as explained in Section 15 below. In this method, we would use `ibConnectionObject` to create the initial contract and order objects, modify them with the requested properties, and then use `ibConnectionObject` again to send the order to IB. Section 15.3 shows a usage example of this.
- We can use *IBMatlab*'s Hold parameter to prepare the contract and order object, then modify them with the extra properties, and finally use `ibConnectionObject` to send the order to IB. The difference vs. the previous method is that we don't need to create the contract and order objects – *IBMatlab* takes care of this for us.

Here is a typical usage example of using *IBMatlab*'s Hold mechanism:

```
% Prepare the initial contract and order objects
[orderId, ibConnectionObject, contract, order] = ...
    IBMatlab('action','BUY', 'Hold',true, ...);

% Modify some contract properties
contract.m_secIdType = 'ISIN';
contract.m_secId = 'US0378331005';  % =Apple Inc.

% Modify some order properties
order.m_clearingIntent = 'Away';
order.m_settlingFirm = 'CSBLO';
order.m_allOrNone = true;

% Send the modified order to IB
ibConnectionObject.placeOrder(orderId, contract, order);
```

Note that the contract and order objects are only returned from *IBMatlab* for trading orders (i.e., Action = 'Buy', 'Sell' or 'SShort'). The same applies for the Hold parameter (default=false; set to 1 or true if you wish to hold the trading order for modification before sending to IB).

---

[38] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/java/contract.htm

[39] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/java/order.htm

10 Accessing and cancelling open trade orders

10.1 Retrieving the list of open orders

To retrieve the list of open IB orders use Action='query' and Type='open' as follows:

```
>> data = IBMatlab('action','query', 'type','open')

data =
1x3 struct array with fields:
    orderId
    contract
    order
    orderState
    status
    filled
    remaining
    avgFillPrice
    permId
    parentId
    lastFillPrice
    clientId
    whyHeld
    message
```

This returns a Matlab struct array, where each array element represents a different open order. In this particular case, we see the three open bracket orders from section 9.3 above.

You can access any of the orders using the standard Matlab dot notation:

```
>> data(1)
ans =
          orderId: 154410310
         contract: [1x1 struct]
            order: [1x1 struct]
       orderState: [1x1 struct]
           status: 'Submitted'
           filled: 0
        remaining: 100
     avgFillPrice: 0
           permId: 989560927
         parentId: 0
    lastFillPrice: 0
         clientId: 8981
          whyHeld: []
          message: [1x162 char]
```

```
>> data(2)
ans =
          orderId: 154410311
         contract: [1x1 struct]
            order: [1x1 struct]
       orderState: [1x1 struct]
           status: 'PreSubmitted'
           filled: 0
        remaining: 100
     avgFillPrice: 0
           permId: 989560928
         parentId: 154410310
    lastFillPrice: 0
         clientId: 8981
          whyHeld: 'child,trigger'
          message: [1x182 char]
```

Each of the order structs contains the following data fields:[40]

- `orderId` – this is the ID returned by *IBMatlab* when you successfully submit a trade order. It is the ID that is used by IB to uniquely identify the trade.
- `contract` – this is a struct object that contains the Contract information, including all the relevant information about the affected security
- `order` – this is another struct object that contains information about the specific trade order's parameters
- `orderState` – this is another struct object that contains information about the current status of the open order. An order can be open with several possible states, and this is reported in this struct's fields.
- `status` – indicates the order status e.g., 'Submitted', 'PreSubmitted', etc.
- `filled` – indicates the number of shares that have been executed in the order
- `remaining` – number of shares remaining to be executed in the order
- `avgFillPrice` – average price of the filled (executed) shares; 0 if no fills
- `permId` – the permanent ID used to store the order in the IB server
- `parentId` – the order ID of the order's parent order; 0 if no parent
- `lastFillPrice` – last price at which shares in the order were executed
- `clientId` – ID of the client used for sending the order (see section 13 below)
- `whyHeld` – specific reasons for holding the order in an open state
- `message` – a detailed message string stating the order's state. This is basically just a string that contains all the fields above and their values.

[40] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/java/orderstatus.htm

For example:

```
>> data(2).contract
ans =
                m_conId: 30351181
               m_symbol: 'GOOG'
              m_secType: 'STK'
               m_expiry: []
               m_strike: 0
                m_right: '?'
           m_multiplier: []
             m_exchange: 'SMART'
             m_currency: 'USD'
          m_localSymbol: 'GOOG'
          m_primaryExch: []
       m_includeExpired: 0
            m_secIdType: []
                m_secId: []
     m_comboLegsDescrip: []
            m_comboLegs: '[]'
            m_underComp: []

>> data(1).order
ans =
                CUSTOMER: 0
                    FIRM: 1
             OPT_UNKNOWN: '?'
      OPT_BROKER_DEALER: 'b'
           OPT_CUSTOMER: 'c'
               OPT_FIRM: 'f'
              OPT_ISEMM: 'm'
              OPT_FARMM: 'n'
         OPT_SPECIALIST: 'y'
          AUCTION_MATCH: 1
    AUCTION_IMPROVEMENT: 2
    AUCTION_TRANSPARENT: 3
              EMPTY_STR: ''
               m_orderId: 154410311
              m_clientId: 8981
                m_permId: 989560928
                m_action: 'SELL'
        m_totalQuantity: 100
            m_orderType: 'STP'
             m_lmtPrice: 580
             m_auxPrice: 0
                  m_tif: 'GTC'
             m_ocaGroup: '989560927'
              m_ocaType: 3
             m_transmit: 1
             m_parentId: 154410310

   (plus many more internal order properties...)
```

```
>> data(1).orderState
ans =
                m_status: 'Submitted'
            m_initMargin: '1.7976931348623157E308'
           m_maintMargin: '1.7976931348623157E308'
         m_equityWithLoan: '1.7976931348623157E308'
             m_commission: 1.79769313486232e+308
          m_minCommission: 1.79769313486232e+308
          m_maxCommission: 1.79769313486232e+308
    m_commissionCurrency: 'USD'
           m_warningText: []
```

Don't let the number `1.79769313486232e+308` alarm you – this is simply IB's way of specifying uninitialized data.

Note: IB warns[41] that "*It is possible that orderStatus() may return duplicate messages. It is essential that you filter the message accordingly.*"

We can filter the results based on a specific Symbol and/or OrderId. For example:

```
>> data = IBMatlab('action','query','type','open','OrderId',154410310)
data =
       orderId: 154410310
      contract: [1x1 struct]
         order: [1x1 struct]
    orderState: [1x1 struct]
         (etc.)
```

Or alternatively (note that symbol filtering is case insensitive):

```
>> data = IBMatlab('action','query', 'type','open', 'symbol','goog')
```

Of course, it is possible that there are no open orders that match the filtering criteria:

```
>> data = IBMatlab('action','query', 'type','open', 'symbol','xyz')
data =
     []
```

Note that you can only retrieve (and modify) open orders that were originally sent by your **IB-Matlab** ClientID. Trades that were placed directly in TWS, or via another API client that connects to TWS, or by another **IB-Matlab** connection session with a different ClientID, will not be accessible to you. If this limitation affects your work, you could try to use a static ClientID of 0, thereby enabling access to all open orders placed by any **IB-Matlab** session (since they all have the same ClientID 0) as well as directly on TWS (which uses the same ClientID 0). See section 13 below for additional details on ClientID.

[41] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/java/orderstatus.htm

10.2 Modifying open orders

To modify parameters of open orders, we need to first ensure they are really open (duh!). This sounds trivial, but one would be surprised at how common a mistake is to try to update an order that has already been filled or cancelled.

When we are certain that the order is open, we can resend the order with modified parameters, along with the OrderId parameter. The OrderId parameter tells *IBMatlab* (and IB) to modify that specific order, rather than to create a new order:

```
orderId = IBMatlab('action','BUY', 'symbol','GOOG', 'quantity',100,...
                   'type','LMT', 'limitPrice',600);

% Let some time pass...

% If the requested order is still open
if ~isempty(IBMatlab('action','query','type','open','OrderId',orderId))

      % Send the trade with modified parameters
      IBMatlab('action','BUY', 'symbol','GOOG', 'quantity',50, ...
               'type','MKT', 'orderID',orderId);

end
```

10.3 Cancelling open orders

To cancel open orders, we need (as above) to first ensure that they are really open (again, duh!), although in this case it does not really matter so much if we are trying to cancel a non-existing order. The only side-effect will be a harmless message sent to the Matlab command window, no real harm done.

To cancel the trade, simply use Action='cancel' with the specific order ID:

```
% If the requested order is still open
if ~isempty(IBMatlab('action','query','type','open','OrderId',orderId))

      % Cancel the requested order
      data = IBMatlab('action','CANCEL', 'orderID',orderId);

end
```

To cancel ALL open orders simply discard the OrderId parameter from the command:

```
data = IBMatlab('action','CANCEL');  % cancel ALL open orders
```

In both cases, the returned `data` is an array of structs corresponding to the cancelled order(s), as described in section 10.1 above.

Alternately, we can use the Java connector object for this (see section 15 for details):

```
% Place an order, return the orderId and the Java connector object
[orderId, ibConnectionObject] = IBMatlab('action','BUY', ...);

% Cancel the order using the underlying Java connector object
ibConnectionObject.cancelOrder(orderId);
```

11 Processing IB events

11.1 Processing events in IB-Matlab

IB uses an asynchronous event-based mechanism for sending information to clients. This means that we do not simply send a request to IB and wait for the answer. Instead, we send a request, and when IB is ready it will send us one or more (or zero) events in response. These events carry data, and by analyzing the stored event data we (hopefully) receive the answer that we were waiting for.

These callbacks are constantly being "fired" (i.e., invoked) by asynchronous messages from IB, ranging from temporary market connection losses/reconnections, to error messages and responses to market queries. Some of the events are triggered by user actions (market or portfolio queries, for example), while others are triggered by IB (e.g., disconnection notifications). The full list of IB events (and their data) is documented in the online API documentation.[42]

Matlab has built-in support for asynchronous events, called *Callbacks* in Matlab jargon.[43] Whereas Matlab callbacks are normally used in conjunction with Graphical User Interfaces (GUI), they can also be used with **IB-Matlab**, which automatically converts all the Java events received from IB into Matlab callbacks.

There are two types of callbacks that you can use in **IB-Matlab**:

- Generic callback – this is a catch-all callback function that is triggered upon any IB event. Within this callback, you would need to write some code to distinguish between the different event types in order to process the events' data. A skeleton for this is given below. The parameter controlling this callback in *IBMatlab* is called CallbackFunction.
- Specific callback – this is a callback function that is only triggered when the specific event type is received from IB. Since the event type is known, you can process its event data more easily than in the generic callback case. However, you would need to specify a different specific callback for each of the event types that you wish to process.

The parameters controlling the specific callbacks in *IBMatlab* are called CallbackXXX, where *XXX* is the name of the IB event (the only exception to this rule is CallbackMessage, which handles the IB *error* event – the reason is that this event sends informational messages in addition to errors,[44] so IB's event name is misleading in this specific case).

---

[42] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/java/java_eclientsocket_methods.htm

[43] http://www.mathworks.com/help/techdoc/creating_guis/f16-999606.html

[44] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/java/error.htm

When you specify any callback function to *IBMatlab*, either the generic kind (CallbackFunction) or a specific kind (CallbackXXX), the command action does not even need to be related to the callback (for example, you can set CallbackExecDetails together with Action='query').

```
data = IBMatlab('action','query', ..., ...
                'CallbackExecDetails',@IBMatlab_CallbackExecDetails);
```

where `IBMatlab_CallbackExecDetails()` is a Matlab function created by you that accepts two input parameters:

- `hObject` – the Java connector object that is described in section 15 below
- `eventData` – a Matlab struct that contains the event's data in separate fields

An example for specifying a Matlab callback function is:

```
function IBMatlab_CallbackExecDetails(ibConnector, eventData)
    % do the callback processing here
end
```

You can pass external data to your callback functions using the callback cell-array format. For example, to pass two extra data values:[45]

```
callbackDetails = {@IBMatlab_CallbackExecDetails, 123, 'abc'};
IBMatlab('action','query',..., 'CallbackExecDetails',callbackDetails);

function IBMatlab_CallbackExecDetails(ibConn,eventData,extra1,extra2)
    % do the callback processing here
end
```

When you specify any callback function to *IBMatlab*, you only need to set it once, in any *IBMatlab* command. Unlike most *IBMatlab* parameters, which are not remembered across *IBMatlab* commands and need to be re-specified, callbacks do not need to be re-specified. They are remembered from the moment they are first set, until such time as Matlab exits or the callback parameter is changed.[46]

To reset a callback (i.e., remove the callback invocation), simply set the callback parameter value to [] (empty square brackets) or '' (empty string):

```
data = IBMatlab('action','query', ..., 'CallbackExecDetails','');
```

[45] http://www.mathworks.com/help/techdoc/creating_guis/f16-999606.html#brqow8p

[46] It is not an error to re-specify the callbacks in each *IBMatlab* command, it is simply useless and makes the code less readable

Matlab callbacks are invoked even if you use the Java connector object (see section 15) for requesting data from IB. This is actually very useful: you can use the connector object to easily send a request to IB, and then process the results in a Matlab callback function.

Here is the list of currently-supported callback events in *IBMatlab*:

***IBMatlab* parameter**	**IB Event**	**Triggered by *IBMatlab*?**	**Called when?**
CallbackAccountDownloadEnd	accountDownloadEnd	Yes	in response to account queries, after all UpdateAccount events were sent, to indicate end of data
CallbackBondContractDetails	bondContractDetails	Yes	in response to market queries; not really used in *IBMatlab*
CallbackConnectionClosed	connectionClosed	Yes	when IB-Matlab loses its connection (or reconnects) to TWS/Gateway
CallbackContractDetails	contractDetails	Yes	in response to market queries; used in *IBMatlab* only to get the tick value
CallbackContractDetailsEnd	contractDetailsEnd	Yes	when all ContractDetails events have been sent, to indicate end of data
CallbackCurrentTime	currentTime	Yes	numerous times during regular work; returns the current server system time
CallbackDeltaNeutralValidation	deltaNeutralValidation	No	in response to a Delta-Neutral DN RFQ
CallbackExecDetails	execDetails	Yes	whenever an order is partially or fully filled, or in response to *reqExecutions()* on the Java connector
CallbackExecDetailsEnd	execDetailsEnd	Yes	when all the ExecDetails events have been sent, to indicate end of data
CallbackFundamentalData	fundamentalData	No	in response to calling *reqFundamentalData()* on the Java connector
CallbackHistoricalData	historicalData	Yes	in response to a historical data request, for each of the result bars separately
CallbackManagedAccounts	managedAccounts	No	when a successful connection is made to a Financial Advisor account, or when *reqManagedAccts ()* is called on the Java connector

IBMatlab parameter	IB Event	Triggered by *IBMatlab*?	Called when?
CallbackMessage	error	Yes	whenever IB wishes to send the user an error or informational message. See section 14 below.
CallbackNextValidId	nextValidId	No	after connecting to IB
CallbackOpenOrder	openOrder	Yes	in response to a user query for open orders, for each open order
CallbackOpenOrderEnd	openOrderEnd	Yes	after all OpenOrder events have been sent for a request, to indicate end of data
CallbackOrderStatus	orderStatus	Yes	in response to a user query for open orders (for each open order), or when an order's status changes
CallbackTickPrice	tickPrice	Yes	in response to a market query, for price fields (e.g., bid)
CallbackTickSize	tickSize	Yes	in response to a market query, for size fields (e.g., bidSize)
CallbackTickString	tickString	Yes	in response to a market query, for string fields (e.g., lastTimestamp)
CallbackTickGeneric	tickGeneric	Yes	in response to a query with a GenericTickList param
CallbackTickEFP	tickEFP	No	when the market data changes. Values are updated immediately with no delay.
CallbackTickOptionComputation	tickOptionComputation	No	when the market in an option or its underlier moves. TWS's option model volatilities, prices, and deltas, along with the present value of dividends expected on that options underlier are received
CallbackTickSnapshotEnd	tickSnapshotEnd	Yes	when all events in response to a snapshot query request have been sent, to indicate end of data
CallbackRealtimeBar	realtimeBar	No	in response to calling *reqRealtimeBars()* on the Java connector
CallbackReceiveFA	receiveFA	No	in response to calling *requestFA()* on the Java connector
CallbackScannerData	scannerData	No	in response to calling *reqScannerSubscription()* on the Java connector

IBMatlab parameter	IB Event	Triggered by *IBMatlab*?	Called when?
CallbackScannerDataEnd	scannerDataEnd	No	when the last scannerData event has been sent, to indicate end of data
CallbackScannerParameters	scannerParameters	No	in response to calling *reqScannerParameters()* on the Java connector
CallbackUpdateAccountTime	updateAccountTime	Yes	together with the Update-AccountValue callbacks, to report on the event time
CallbackUpdateAccountValue	updateAccountValue	Yes	for every single property in the list of account properties, when the account data is requested (see section 4) or updated
CallbackUpdateMktDepth	updateMktDepth	No	when the market depth has changed
CallbackUpdateMktDepthL2	updateMktDepthL2	No	when the Level II market depth has changed
CallbackUpdateNewsBulletin	updateNewsBulletin	No	for each new bulletin if the client has subscribed by calling *reqNewsBulletins()* on the Java connector
CallbackUpdatePortfolio	updatePortfolio	Yes	when account updates are requested or occur

11.2 Example – using CallbackExecDetails

The *execDetails* event is triggered whenever an order is fully or partially executed. Let us trap this event and send the execution information into a CSV file for later use in Excel (also see section 12 below):

```
orderId = IBMatlab('action','BUY', 'symbol','GOOG', 'quantity',1, ...
                   'limitPrice',600, ...
                   'CallbackExecDetails',@IBMatlab_CallbackExecDetails);
```

Where the function `IBMatlab_CallbackExecDetails` is defined as follows (for example, in a file called *IBMatlab_CallbackExecDetails.m*):[47]

```
function IBMatlab_CallbackExecDetails(ibConnector, eventData, varargin)

   % Extract the basic event data components
   contractData  = eventData.contract;
   executionData = eventData.execution;

   % Example of extracting data from the contract object:
   % http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/java/contract.htm
   symbol  = char(eventData.contract.m_symbol);
   secType = char(eventData.contract.m_secType);
   % ... several other contract data field available - see above webpage

   % Example of extracting data from the execution object:
   % http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/java/execution.htm
   orderId     = eventData.execution.m_orderId;
   execId      = char(eventData.execution.m_execId);
   time        = char(eventData.execution.m_time);
   exchange    = char(eventData.execution.m_exchange);
   side        = char(eventData.execution.m_side);
   shares      = eventData.execution.m_shares;
   price       = eventData.execution.m_price;
   permId      = eventData.execution.m_permId;
   liquidation = eventData.execution.m_liquidation;
   cumQty      = eventData.execution.m_cumQty;
   avgPrice    = eventData.execution.m_avgPrice;
   % ... several other execution data field available - see above webpage

   % Convert the data elements into a comma-separated string
   csvline = sprintf('%s,%d,%s,%d,%d,%f\n', time, orderId, symbol, ...
                                            shares, cumQty, price);

   % Now append this comma-separated string to the CSV file
   fid = fopen('executions.csv', 'at');  % 'at' = append text
   fprintf(fid, csvline);
   fclose(fid);

end  % IBMatlab_CallbackExecDetails
```

[47] This file can be downloaded from: http://UndocumentedMatlab.com/files/IBMatlab_CallbackExecDetails.m

11.3 Example – using CallbackTickGeneric

In this example, we attach a user callback function to *tickGeneric* events in order to check whether a security is shortable[48] (also see section 5 above).

Note: according to IB,[49] "*Generic Tick Tags cannot be specified if you elect to use the Snapshot market data subscription*", and therefore we need to use the streaming-quotes mechanism, so QuotesNumber>1:

```
orderId = IBMatlab('action','Query', 'symbol','GOOG', ...
                   'GenericTicklist','236', 'QuotesNumber',2, ...
                   'CallbackTickGeneric',@IBMatlab_CallbackTickGeneric);
```

where the function `IBMatlab_CallbackTickGeneric` is defined as follows:[50]

```
function IBMatlab_CallbackTickGeneric(ibConnector, eventData, varargin)

    % Only check the shortable tick type =46, according to
    % http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm
#apiguide/api/tick_values.htm%23XREF_tick_values_generic_tick

    if eventData.field == 46  % 46=Shortable

        % Get this event's tickerId (=orderId as returned from the
        % original IBMatlab command)
        tickerId = eventData.tickerId;

        % Get the corresponding shortable value
        shortableValue = eventData.generic;

        % Now check whether the security is shortable or not
        title = sprintf('Shortable info for request %d', tickerId);
        if (shortableValue > 2.5)      % 3.0
            msgbox('>1000 shares available for a short', title, 'help');
        elseif (shortableValue > 1.5)  % 2.0
            msgbox('This contract will be available for short sale if
shares can be located', title, 'warn');
        elseif (shortableValue > 0.5)  % 1.0
            msgbox('Not available for short sale', title, 'warn');
        else
            msg = sprintf('Unknown shortable value: %g',shortableValue);
            msgbox(msg, title, 'error');
        end
    end  % if shortable tickType

end  % IBMatlab_CallbackTickGeneric
```

Note that in this particular example we could also have simply used the streaming quotes data, instead of using the callback:

```
>> dataS = IBMatlab('action','query','symbol','GOOG','quotesNumber',-1);
>> shortableValue = dataS.data.shortable;  % =3 for GOOG
```

---

[48] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/tables/using_the_shortable_tick.htm

[49] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/tables/generic_tick_types.htm

[50] This code can be downloaded from: http://UndocumentedMatlab.com/files/IBMatlab_CallbackTickGeneric.m

12 Tracking trade executions

IB-Matlab provides several distinct ways to programmatically track trade executions:

12.1 User requests

To retrieve the list of trade executions done in the IB account today, use Action='query' and Type='executions' as follows (note the similarities to the request for open order, section 10.1 above):

```
>> data = IBMatlab('action','query', 'type','executions')

data =
1x3 struct array with fields:
    orderId
    execId
    time
    exchange
    side
    shares
    symbol
    price
    permId
    liquidation
    cumQty
    avgPrice
    contract
    execution
```

This returns a Matlab struct array, where each array element represents a different execution event.

You can access any of the orders using the standard Matlab dot notation:

```
>> data(1)
ans =
        orderId: 154735358
         execId: '00018037.4ff27b0e.01.01'
           time: '20120216  18:50:14'
       exchange: 'ISLAND'
           side: 'BOT'
         shares: 1
         symbol: 'GOOG'
          price: 602.82
         permId: 300757703
    liquidation: 0
         cumQty: 1
       avgPrice: 602.82
       contract: [1x1 struct]
      execution: [1x1 struct]
```

```
>> data(2)
ans =
        orderId: 154737092
         execId: '00018037.4ff2a3b8.01.01'
           time: '20120216  18:58:57'
       exchange: 'BEX'
           side: 'SLD'
         shares: 3
         symbol: 'GOOG'
          price: 605.19
         permId: 300757711
    liquidation: 0
        cumQty: 3
       avgPrice: 605.19
       contract: [1x1 struct]
      execution: [1x1 struct]
```

Each of the order structs contains the following data fields:[51]

- `orderId` – this is the ID returned by *IBMatlab* when you successfully submit a trade order. It is the ID that is used by IB to uniquely identify the trade. TWS orders have a fixed order ID of zero (0).
- `execId` – the unique ID assigned to this execution
- `time` – indicates the time of execution (local user time, not IB server time)
- `exchange` – the exchange which executed the trade
- `side` – BOT (=buy) or SLD (=sell)
- `shares` – the number of executed shares
- `symbol` – the security's symbol (use the contract field to get the LocalSymbol)
- `price` – the execution price
- `permId` – the permanent ID used to store the order in the IB server
- `liquidation` – Identifies the position as one to be liquidated last should the need arise
- `cumQty` – the cumulative quantity of shares filled in this trade (used for partial executions)
- `avgPrice` – the weighted average price of partial executions for this trade
- `contract` – this is a struct object that contains the Contract information, including all the relevant information about the affected security
- `execution` – this is another struct object that contains information about the specific execution's parameters

[51] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/java/orderstatus.htm

For example:

```
>> data(2).contract
ans =
           m_conId: 30351181
          m_symbol: 'GOOG'
         m_secType: 'STK'
          m_expiry: []
          m_strike: 0
           m_right: []
      m_multiplier: []
        m_exchange: 'BEX'
        m_currency: 'USD'
     m_localSymbol: 'GOOG'
     m_primaryExch: []
  m_includeExpired: 0
       m_secIdType: []
           m_secId: []
m_comboLegsDescrip: []
       m_comboLegs: '[]'
       m_underComp: []

>> data(2).execution
ans =
         m_orderId: 154737092
        m_clientId: 8101
          m_execId: '00018037.4ff2a3b8.01.01'
            m_time: '20120216  18:58:57'
      m_acctNumber: 'DU90912'
        m_exchange: 'BEX'
            m_side: 'SLD'
          m_shares: 3
           m_price: 605.19
          m_permId: 300757711
     m_liquidation: 0
          m_cumQty: 3
        m_avgPrice: 605.19
```

We can filter the results based on a specific Symbol and/or OrderId. For example:

```
>> data = IBMatlab('action','query','type','open','OrderId', 154737092)
data =
        orderId: 154737092
         execId: '00018037.4ff2a3b8.01.01'
         (etc.)
```

Or alternately (note that symbol filtering is case insensitive):

```
>> data = IBMatlab('action','query', 'type','open', 'symbol','goog')
```

Of course, it is possible that there are no open orders that match the filtering criteria:

```
>> data = IBMatlab('action','query', 'type','open', 'symbol','xyz')
data =
     []
```

12.2 Automated log files

IB-Matlab automatically stores two log files of trade executions. Both files have the same name, and different extensions.

The default file name (<LogFileName>) for these files is *IB_tradeslog_yyyymmdd*, where yyyymmdd is the current date. For example, on 2012-02-15 the log files will be called *IB_tradeslog_20120215.csv* and *IB_tradeslog_20120215*.mat.

<LogFileName> can be modified by setting the LogFileName parameter in *IBMatlab* (default = './IB_tradeslog_YYYYMMDD.csv'). Note the leading './' in the default value of LogFileName – you can use any other folder path if you want to store the log files in a different folder than the current Matlab folder.

- A CSV (comma separated values) text file named <LogFileName>.csv. A separate line is stored for each execution event. This file can be opened in Excel as well as by any text editor.

- A MAT (Matlab compressed format) binary file named <LogFileName>.mat that stores the struct array explained in section 12.1 above, excluding the sub-structs `contract` and `execution`.

12.3 Using CallbackExecDetails

You can set the CallbackExecDetails parameter to a user-defined Matlab function that will process each execution event at the moment that it is reported. Section 11.2 above contains a working example of such a function.

As noted in section 11.1, you only need to set CallbackExecDetails once (this is normally done in the same *IBMatlab* command that sends the trade order). You do not need to re-specify this callback in subsequent *IBMatlab* commands, unless you wish to override the parameter with a different function, or to cancel it (in which case you would set it to [] or '').

13 Forcing connection parameters

IB-Matlab does not require any special configuration when connecting to IB. It uses a random client ID when first connecting to TWS or the IB Gateway, and this is perfectly fine for the vast majority of uses.

However, in some specific cases, users may wish to control the connection properties. This is supported in **IB-Matlab** using the following input parameters:

Parameter	Data type	Default	Description
ClientId	integer	(*random*)	A number that identifies **IB-Matlab** to TWS/Gateway. 0 acts as another TWS.
Host	string	'localhost' = '127.0.0.1'	IP address of the computer that runs TWS/Gateway
Port	integer	7496	Port number used by TWS/Gateway for API communication
AccountName	string	''	The specific IB account used for queries or trades. Useful when you handle multiple IB accounts, otherwise leave empty.
FAProfile	string	''	The Financial Advisor allocation profile to which trades will be allocated. Only relevant for Financial Advisor accounts, otherwise leave empty.

The ClientID, Host and Port properties should match the API configuration of the TWS/Gateway applications, as described in section 2 above (installation steps #9-10).

You can set a static Client ID in order to be able to modify open orders placed in a different **IB-Matlab** session (i.e., after the original **IB-Matlab** client has disconnected from IB and a new **IB-Matlab** has connected). IB normally prevents clients from modifying orders placed by other clients, but if all your clients use the same ID then this limitation will not affect you.

Matlab-to-IB reconnections occur automatically when **IB-Matlab** needs to issue any request to IB and the connection is not live for whatever reason. This happens upon the initial *IBMatlab* request (when the initial connection needs to be established); after TWS or the IB Gateway were closed; after a call was made to `ibConnectionObject.disconnectFromTWS` (see below); after a Matlab restart; after a specified number of streaming quotes, and in a few other special cases.[52]

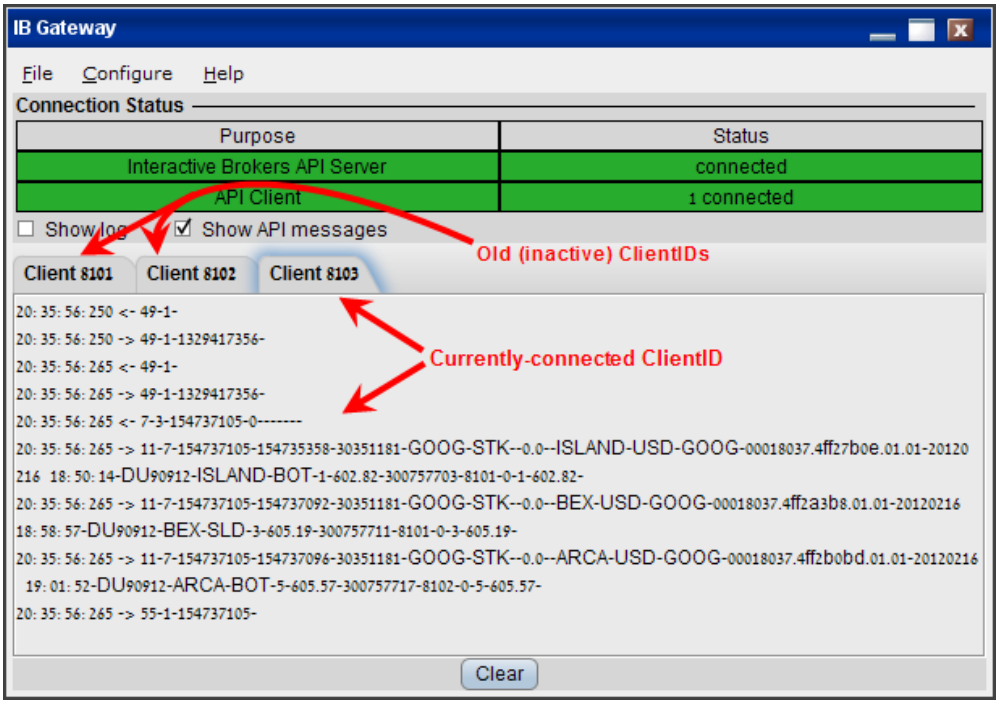
In reconnections of any kind, **IB-Matlab** automatically tries to reuse the same ClientID as in the previous connection.

When a new ClientID is specified for any *IBMatlab* command, *IBMatlab* automatically disconnects the previous client ID and reconnects as the new ClientID.

[52] See the ReconnectEvery parameter (section 7 above).

In the IB Gateway, this will be seen as a dark-gray tab contents for the old ClientID and a light-gray tab contents for the connected ClientID:

```
data = IBMatlab('action','query', 'type','executions', 'ClientID',8103)
```



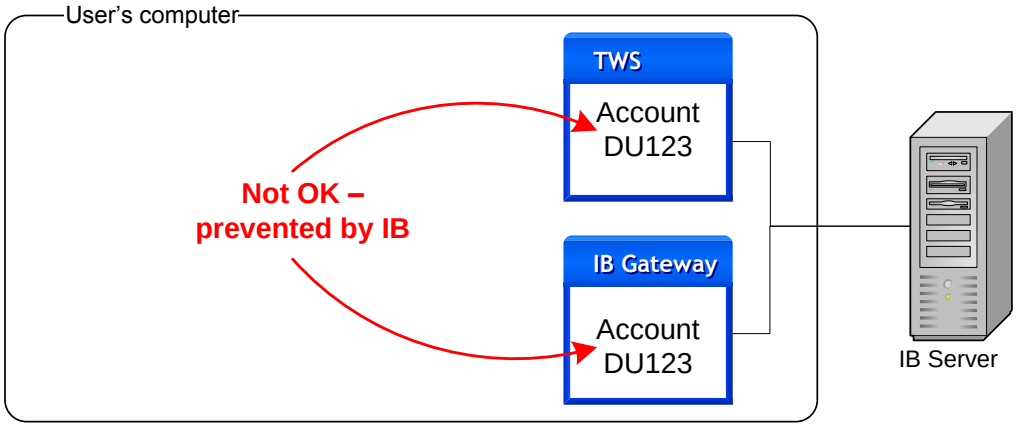
While specifying a new ClientID automatically disconnects and reconnects to IB, you can also force a disconnection/reconnection for the same ClientID, by using the Java connector object (discussed in section 15 below). Following a disconnection from IB, **IB-Matlab** will automatically reconnect to IB upon the very next use of *IBMatlab*:

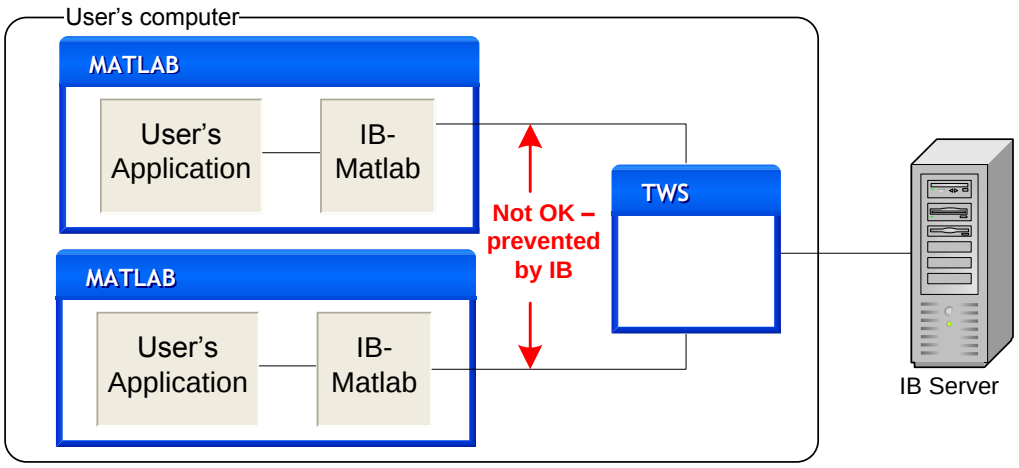
```
[data, ibConnectionObject] = IBMatlab('action',...);  % do whatever
ibConnectionObject.disconnectFromTWS;   % disconnect from IB
data = IBMatlab('action','portfolio');  % will automatically reconnect
```

ClientID 0 is special: it simulates the TWS and enables **IB-Matlab** to receive all the open orders that were interactively entered in TWS. Instead of ClientID 0, you can use any other value that you pre-configured as the Master API Client ID in the TWS/Gateway's API configuration screen (see section 2 installation step #9). Using a Master Client ID enables the **IB-Matlab** client to receive all open orders that were placed in the IB account using any Client ID, not just TWS. If you only connect **IB-Matlab** and no other API client to TWS, and if you only use the static ClientID 0, then you do not need to worry about the Master API Client ID setup.

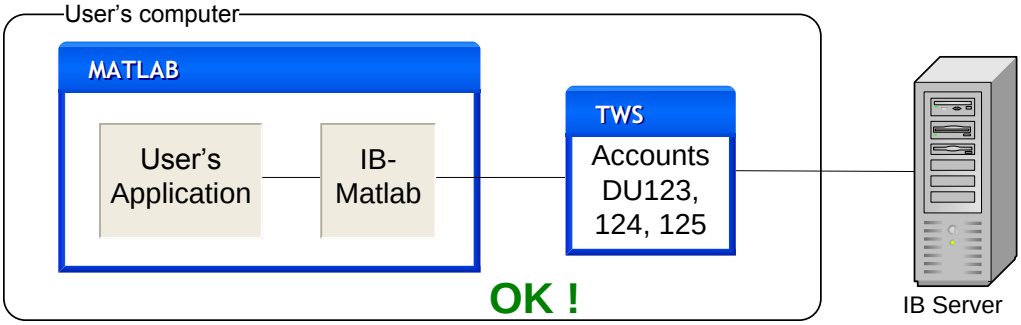
Another use is to connect **IB-Matlab** on one computer (which has Matlab installed) to TWS/Gateway on another computer, which may not necessarily have Matlab. In this case, simply set the Host and possibly also the Port parameters.

Note that TWS and the IB Gateway have a limitation that they can only be connected to a single **IB-Matlab** client at any time. Also, TWS and the IB Gateway cannot be logged-in at the same time to the same IB account. These IB limitations mean you cannot simultaneously connect multiple **IB-Matlab** instances to the same IB account.

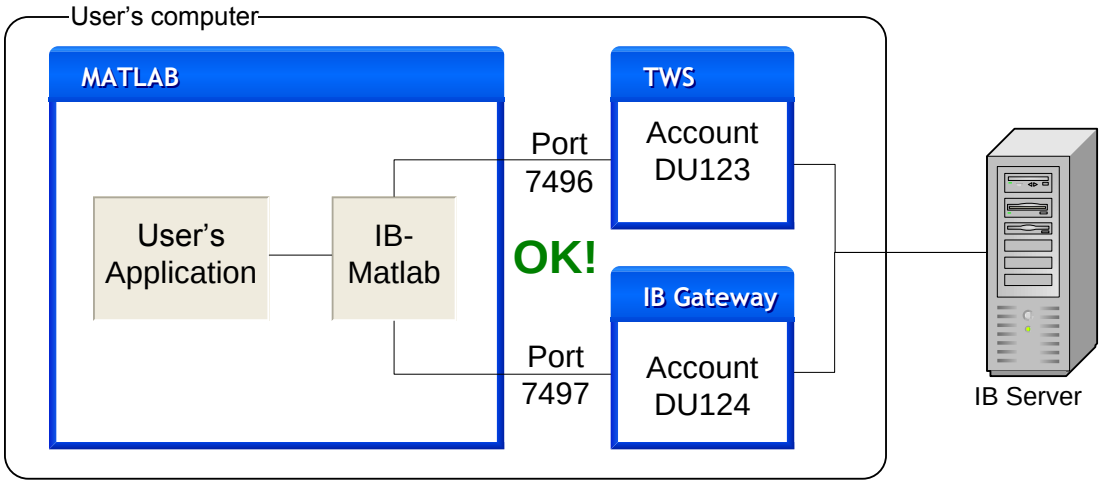


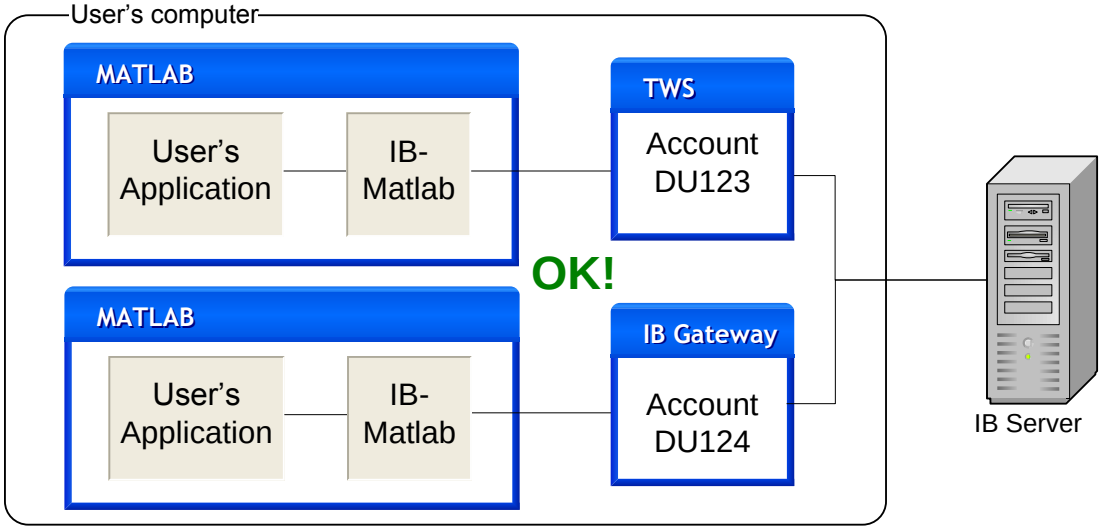


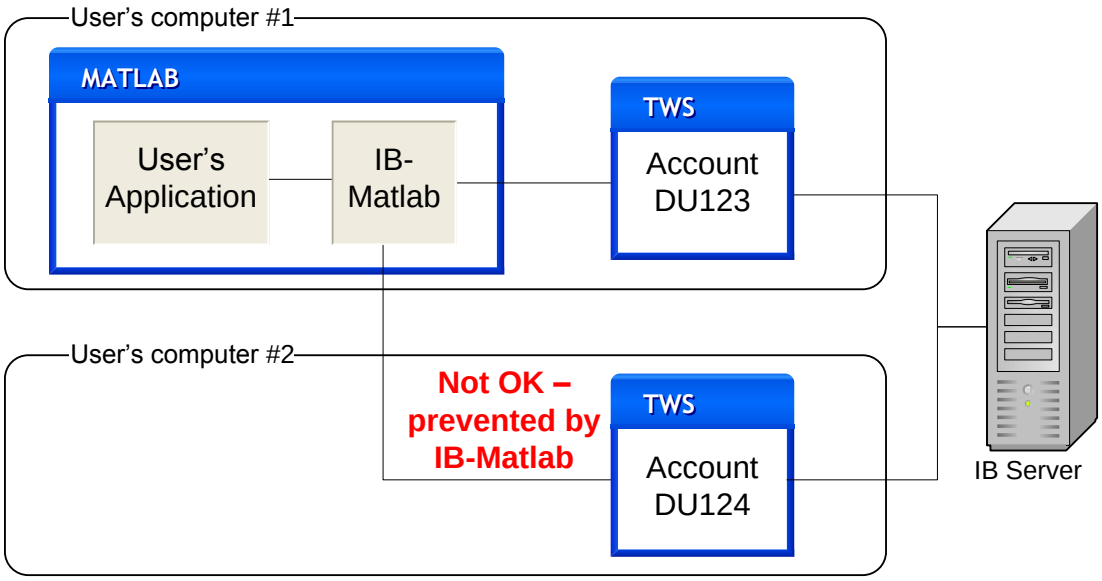
On the other hand, it is possible to control multiple IB accounts from the same TWS application, and in such a case **IB-Matlab** can access all of these accounts when it connects to TWS, using the AccountName parameter. Please refer to your TWS documentation (or IB's customer service) to set up your TWS accordingly.

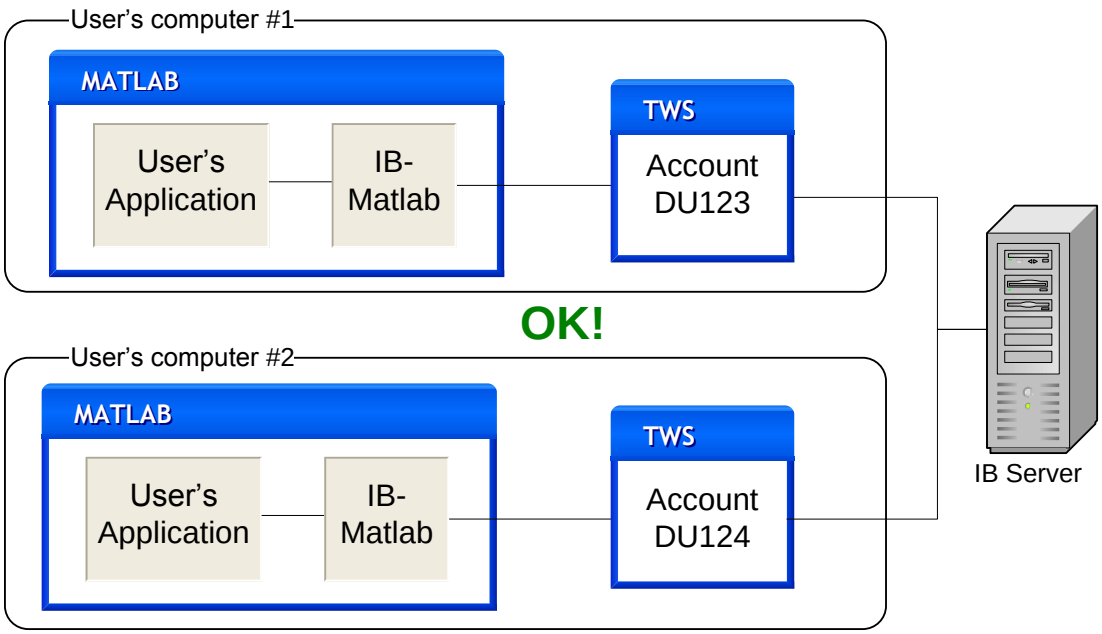


It is also possible to run TWS with one IB account, and IB Gateway with another account, either on the same computer or on different machines. You can then connect one or more **IB-Matlab** instances to these IB applications at the same time. Simply ensure that your Host, Port and AccountName parameters are OK for any *IBMatlab* command. **IB-Matlab** can maintain simultaneous connections to both TWS and IB Gateway, on different Ports, as long as they are all on the same Host; to control two or more TWS/Gateways on different machines, it is better to use distinct **IB-Matlab** instances, i.e., distinct Matlab sessions, each running its own **IB-Matlab** instance.






User's computer #1
MATLAB
User's Application
IB-Matlab
TWS
Account DU123
User's computer #2
Not OK – prevented by IB-Matlab
TWS
Account DU124
IB Server


User's computer #1
MATLAB
User's Application
IB-Matlab
TWS
Account DU123
OK!
User's computer #2
MATLAB
User's Application
IB-Matlab
TWS
Account DU124
IB Server

14 Messages and general error handling

IB constantly sends messages of various severity levels to **IB-Matlab**. These range from the mundane (e.g., "*Market data farm connection is OK:cashfarm {-1, 2104}*") to the problematic (e.g., "*No security definition has been found for the request {153745227, 200}*"). All these messages arrive as regular events of type *error*, just like all the other information sent from IB (see section 11 above for details).

IB-Matlab automatically displays messages in the Matlab command window. The user can control the display of these messages using the MsgDisplayLevel parameter, which accepts the following possible values:

- -2 – most verbose output, including all the information contained in all incoming IB events (not just messages)
- -1 – display all messages as well as basic events information
- 0 (default) – display all messages, but not other events
- 1 – only display error messages, not informational messages
- 2 – do not display any automated output onscreen (not even errors)

We can trap and process the message events just like any other IB events, using Matlab callbacks. Note that the parameter for message callback is CallbackMessage, although for some reason the IB event is called *error*:

```
data = IBMatlab('action','query', ..., 'MsgDisplayLevel',-1, ...
                'CallbackMessage',@IBMatlab_CallbackMessage);
```

The information contained in the message events varies depending on message type.[53] The three main variants appear to be events with one of the following data sets:

Contents	Description	Displayed as	Displayed onscreen if
message	general error messages	[API.msg1]	MsgDisplayLevel <= 1
message, id (data1), code (data2)	errors and informational messages	[API.msg2]	MsgDisplayLevel <= 0
message, exception object	severe IB errors (exceptions)	[API.msg3]	MsgDisplayLevel <= 1

The full list of message codes (data2) for API.msg2 (which is the most common message type) is listed online.[54] It is sub-divided into three groups:

- Error messages (codes 100-999)
- System messages (codes 1000-1999)
- Warning messages (codes 2000-2999)

[53] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide/java/error.htm

[54] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#api_message_codes.htm

As noted above, most of the messages are of type API.msg2, and contain two numeric fields: data2 contains the message code, and data1 contains message-specific data. For most message codes (e.g., "*Market data farm connection is OK:cashfarm*" =code 2104), there is no associated message-specific data, and in such cases data1 = -1.

In some cases, however, data1 does contain relevant information. For example, "*No security definition has been found for the request {153745227, 200}*" tells us that this error (code=2000 happened for the specific request ID 153745227. We can therefore correlate between the error and the trade request that triggered this error.

As noted above, the API.msg2 messages reported by IB are often cryptic, and we sometimes need to use detective skills in order to track down the root cause of a problem.[55] Several mechanisms can help us with this detective work:

- We could set *IBMatlab*'s MsgDisplayLevel input parameter to -1 or -2 (see above).

- We could set *IBMatlab*'s Debug input parameter (default=0) to 1. This will display in the Matlab Command Window a long list of parameters used by *IBMatlab* to prepare the request for IB. Check this list for any default values that should actually be set to some non-default values.

- We could set the API logging level to "Detailed" in the TWS/Gateway API configuration window.[56] By default it is set to "Error", and this can be changed at any time. This affects the amount of information (verbosity) logged in IB's log files, which are located in IB's installation folder (e.g., C:\Program Files\Jts).[57] The log files are separated by the day of week, and have names such as: *ibgateway.Thu.log, log.Wed.txt, api.8981.Tue.log*. These refer, respectively, to the main Gateway log, the main TWS log,[58] and a log of requests/responses for a specific ClientID. The *api.*.log* file reflects the contents of the corresponding tab in the Gateway application.[59] Note that setting the logging level to "Detail" has a performance overhead and should be avoided except when debugging a specific issue. In other cases, you can set the level to "Information", "Warning" or back to the default "Error".

[55] See examples in section 3 above

[56] See section 2, installation step 9d

[57] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/api/api_logging.htm

[58] The list of IDs used in the *ibgateway.*.log, log.*.txt* log files when the logging level is "Detailed", is described here: http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/api/api_request_server_response_message_identifiers.htm; information about the format of the extra log entries can be found in http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/api/api_logging.htm

[59] See the screenshot in section 13 above

In addition to messages reported by IB, the user's program must check for and handle cases of exceptions caused by **IB-Matlab**. In the vast majority of cases, these are due to invalid input parameters being passed to *IBMatlab* (for example, an invalid Action parameter value). However, an exception could also happen due to network problems, or even an occasional internal bug due to an unhandled edge-case situation.

To trap and handle such programmatic exceptions, wrap your calls to *IBMatlab* within a try-catch block, as follows:

```
try
    data = IBMatlab('action','query', ... );
catch
    % process the exception here
end
```

Try-catch blocks do not have any performance or memory overhead and are a very effective way to handle programmatic errors. We highly recommend that you use them very liberally within your user program, not just to wrap *IBMatlab* calls but also for any other processing tasks. I/O sections in particular (reading/writing to files) are prone to errors and are prime candidates for such exception handling. The same applies for processing blocks that handle user inputs (we can never really be too sure what invalid junk a user might enter in there, can we?).

Very common causes of errors when using **IB-Matlab** are relying on default parameter values, and specifying numeric parameter values within string quotes (e.g., '1' rather than 1).[60] Users of **IB-Matlab** should take extra precaution in their programs to ensure that these common mistakes do not occur.

[60] Both of these were discussed in section 3 above

15 Using the Java connector object

15.1 Using the connector object

Each call to *IBMatlab* returns two output values:

- `data` - generally contains the request ID or the requested query data
- `ibConnectionObject` – a Java object reference

In most cases, users do not need to use `ibConnectionObject` and so we can generally ignore the second output value and simply call *IBMatlab* with a single output:

```
data = IBMatlab('action','query', ... );
```

However, flexible and feature-rich as *IBMatlab* is, it does not contain the entire set of functionalities exposed by IB's Java API. In order to access these additional functionalities, we need to use `ibConnectionObject`:

```
[data, ibConnectionObject] = IBMatlab('action','query', ... );
```

`ibConnectionObject` is a java object of type `javahandle_withcallbacks.IBMatlab.-IBConnection`. You can call its publicly-accessible methods (functions) just like any other Matlab function. For example:

```
[data, ibConnectionObject] = IBMatlab('action','query', ... );

flag = ibConnectionObject.isConnected;   % no input params, so no ()
ibConnectionObject.disconnectFromTWS();  % no real need for () here
ibConnectionObject.cancelOrder(153745227);
```

There is an almost exact correlation between the methods in `ibConnectionObject` and the methods documented in IB's Java API (for both requests[61] and responses[62]). This is not coincidental: `ibConnectionObject` is in many respects an interface object to IB's Java API. Therefore, the full documentation of `ibConnectionObject` is really the official IB Java API documentation.

When you call any of the request methods, you cannot really call the corresponding event methods to receive and process the data. For example, if you call `ibConnectionObject.reqCurrentTime()`, you cannot call the corresponding `currentTime()` method. Instead, it is automatically being called by the underlying Java engine as a new event. However, as noted in section 11.1, all these events can be trapped and processed within Matlab callbacks. In this particular case, a *currentTime* event is raised and this can be trapped and processed in a user Matlab function specified by the CallbackCurrentTime parameter.

---

[61] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/java/java_eclientsocket_methods.htm

[62] http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/java/java_ewrapper_methods.htm

15.2 Programming interface

The following is the publicly-accessible interface of `ibConnectionObject`:

```
// Contract, ContractDetails, EClientSocket, EWrapper, EWrapperMsgGenerator,
// Execution, ExecutionFilter, Order, OrderState, ScannerSubscription, UnderComp
import com.ib.client.*;

public class IBConnection

{

    public final static String DEFAULT_TWS_HOST = "localhost";
    public final static int    DEFAULT_TWS_PORT = 7496;
    public final static int    DEFAULT_TWS_CLIENT_ID = 1;

    // Getter functions for the connection parameters

    public String getHost()
    public int    getPort()
    public int    getClientId()

    /*********************************
     * Active requests to IB via TWS
     *********************************/

    // Check if connected to TWS
    public boolean isConnected()

    // Disconnect from TWS
    public void disconnectFromTWS()

    // Request the version of TWS instance to which the API application is connected
    public int getServerVersion()

    // Request the time the API application made a connection to TWS
    public String getTwsConnectionTime()

    // Request the current server time
    public void reqCurrentTime ()
    public void Systime()

    // Request market data
    public void reqMktData(int tickerId, String m_symbol, String m_secType,
                           String m_expiry, double m_strike, String m_right,
                           String m_exchange, String m_currency,
                           String m_localSymbol, String genericTickList,
                           boolean snapshotFlag)

    public void reqMktData(int tickerId, String m_symbol, String m_secType,
                           String m_expiry, double m_strike, String m_right,
                           String m_exchange, String m_currency,
                           String m_localSymbol, boolean snapshotFlag)

    public void reqMktData(int tickerId, Contract contract, String genericTickList,
                           boolean snapshotFlag)

    public void reqMktData(int tickerId, Contract contract, boolean snapshotFlag)

    // Cancel a market data request
    public void cancelMktData(int tickerId)

    // Request market depth data
    public void reqMktDepth(int tickerId, String symbol, String secType,
                            String expiry, double strike, String right,
                            String exchange, String currency, String localSymbol,
                            int numRows)

    public void reqMktDepth(int tickerId, Contract contract, int numRows)
```

```
// Cancel a market depth request
public void cancelMktDepth(int tickerId)

// Request historic market data
public void reqHistoricalData (int tickerId, String symbol, String secType,
                               String expiry, double strike, String right,
                               String exchange, String currency,
                               String localSymbol, String endDateTime,
                               String durationStr, String barSizeSetting,
                               String whatToShow, int useRTH, int formatDate)

public void reqHistoricalData (int tickerId, Contract contract,
                               String endDateTime, String durationStr,
                               String barSizeSetting, String whatToShow,
                               int useRTH, int formatDate)

// Cancel historic data request
public void cancelHistoricalData (int tickerId)

// Request contract details
public void reqContractDetails (int tickerId, Contract contract)

// Place an order
public void placeOrder(int id, String symbol, String secType, String expiry,
                       double strike, String right, String exchange,
                       String currency, String localSymbol,String action,
                       int Quantity, String Type, double lmtPrice,
                       double auxPrice, String tif, String ocaGroup,
                       int parentId, String goodAfterTime,
                       String goodTillDate,double trailStopPrice,
                       int triggerMethod, boolean outsideRTH)

public void placeOrder(int id, Contract contract, Order order)

// Create a contract
public Contract createContract(String symbol, String secType, String expiry,
                               double strike, String right, String exchange,
                               String currency, String localSymbol)

// Create an order
public Order createOrder(String action, int quantity, String type,
                         double lmtPrice, double auxPrice, String tif,
                         String ocaGroup, int parentId, String goodAfterTime,
                         String goodTillDate, double trailStopPrice,
                         int triggerMethod, boolean outsideRTH)

// Cancel a placed order
public void cancelOrder(int tickerId)

// Requests account values, portfolio, and account update time information
public void reqAccountUpdates (Boolean subscribeFlag, String acctCode)

// Requests a list of the day's execution reports
public void reqExecutions (int reqId, ExecutionFilter executionFilter)

// Requests a list of current open orders for the requesting client and
// associates TWS open orders with the client.
// The association only occurs if the requesting client has a Client ID of 0.
public void reqOpenOrders ()

// Requests a list of all open orders
public void reqAllOpenOrders ()

// Automatically associates a new TWS with the client.
// The association only occurs if the requesting client has a Client ID of 0
public void reqAutoOpenOrders (Boolean autoBindFlag)

// Requests IB news bulletins
public void reqNewsBulletins (Boolean allMsgsFlag)
```

```
// Cancels IB news bulletins
public void cancelNewsBulletins ()

// Requests a list of Financial Advisor (FA) managed account codes
public void reqManagedAccts ()

// Requests FA configuration information from TWS
public void requestFA (int faDataType)

// Modifies FA configuration information from the API
public void replaceFA (int faDataType, String xmlStr)

// Requests an XML doc that describes valid parameters of a scanner subscription
public void reqScannerParameters ()

// Requests market scanner results
public void reqScannerSubscription (int tickerId,
                                    ScannerSubscription scannerSubscription)

// Cancels a scanner subscription
public void cancelScannerSubscription (int tickerId)

// Requests real-time bars
public void reqRealTimeBars(int tickerId, Contract contract, int intVal,
                            String str, Boolean flag)

// Cancels real-time bars
public void cancelRealTimeBars(int tickerId)

// Exercises options
public void exerciseOptions(int tickerId, Contract contract, int exerciseAction,
                            int exerciseQuantity, String account, int override)

// Requests Reuters global fundamental data. There must be a subscription to
// Reuters Fundamental setup in Account Management before you can receive data
public void reqFundamentalData(int id, Contract contract, String str)

// Cancels Reuters global fundamental data
public void cancelFundamentalData(int id)

// Sets the level of API request and processing logging
public void setServerLogLevel (int logLevel)

// Set the message display level
// (0=display all messages; 1=display errors only; 2=display no messages)
public void setMsgDisplayLevel(int displayLevel)

// Get the message display level
public int getMsgDisplayLevel()

// Set the Done flag
public void setDone(boolean flag)

// Get the Done flag
public boolean isDone()

/***************************
 * IB Callbacks
 ***************************/

// Error and informational messages
public void error(String str)
public void error(int data1, int data2, String str)
public void error(Exception e)

// TWS connection has closed
public void connectionClosed()

// Get market data
public void tickPrice(int tickerId, int field, double price, int canAutoExecute)

public void tickSize(int tickerId, int field, int size)
```

```
public void tickString(int tickerId, int field, String value)

public void tickGeneric(int tickerId, int field, double generic)

public void tickEFP(int tickerId, int field, double basisPoints,
                    String formattedBasisPoints, double totalDividends,
                    int holdDays, String futureExpiry, double dividendImpact,
                    double dividendsToExpiry)

public void tickOptionComputation(int tickerId, int field, double impliedVol,
                                  double delta, double modelPrice,
                                  double pvDividend)

public void tickOptionComputation(int tickerId, int field, double impliedVol,
                                  double delta, double optPrice,
                                  double pvDividend, double gamma, double vega,
                                  double theta, double undPrice)

public void tickSnapshotEnd(int reqId)

// Receives execution report information
public void execDetails(int orderId, Contract contract, Execution execution)

// Receives historical data results
public void historicalData(int reqId, String date, double open, double high,
                           double low, double close, int volume, int count,
                           double WAP, boolean hasGaps)

// Receives the next valid order ID upon connection
public void nextValidId(int orderId)

// Receive data about open orders
public void openOrder(int orderId, Contract contract, Order order)
public void openOrder(int orderId, Contract contract, Order order,
                      OrderState orderState)

// Receive data about orders status
public void orderStatus(int orderId, String status, int filled, int remaining,
                        double avgFillPrice, int permId, int parentId,
                        double lastFillPrice, int clientId)

public void orderStatus(int orderId, String status, int filled, int remaining,
                        double avgFillPrice, int permId, int parentId,
                        double lastFillPrice, int clientId, String whyHeld)

// Receives a list of Financial Advisor (FA) managed accounts
public void managedAccounts(String accountsList)

// Receives Financial Advisor (FA) configuration information
public void receiveFA(int faDataType, String xml)

// Receives an XML doc that describes valid parameters of a scanner subscription
public void scannerParameters(String xml)

// Receives market scanner results
public void scannerData(int reqId, int rank, ContractDetails contractDetails,
                        String distance, String benchmark, String projection,
                        String legsStr)

public void scannerDataEnd(int reqId)

// Receives the last time account information was updated
public void updateAccountTime(String timeStamp)

// Receives current account values
public void updateAccountValue(String key, String value, String currency)
public void updateAccountValue(String key, String value, String currency,
                               String accountName)

// Receives IB news bulletins
public void updateNewsBulletin(int msgId, int msgType, String message,
                               String origExchange)
```

```
    // Receives market depth information
    public void updateMktDepth(int tickerId, int position, int operation, int side,
                               double price, int size)

    // Receives Level 2 market depth information
    public void updateMktDepthL2(int tickerId, int position, String marketMaker,
                                 int operation, int side, double price, int size)

    // Receives current portfolio information
    public void updatePortfolio(Contract contract, int position, double marketPrice,
                                double marketValue, double averageCost,
                                double unrealizedPNL, double realizedPNL)

    public void updatePortfolio(Contract contract, int position, double marketPrice,
                                double marketValue, double averageCost,
                                double unrealizedPNL, double realizedPNL,
                                String accountName)

    // Receives contract information
    public void contractDetails(int reqId, ContractDetails contractDetails)

    // Receives bond contract information
    public void bondContractDetails(int reqId, ContractDetails contractDetails)

    // Identifies the end of a given contract details request
    public void contractDetailsEnd(int reqId)

    // Receives real-time bars
    public void realtimeBar(int reqId, long time, double open, double high,
                            double low, double close, long volume, double wap,
                            int count)

    // Receives the current system time on the server
    public void currentTime(long time)

    // Receives Reuters global fundamental market data
    public void fundamentalData(int reqId, String data)

    public void accountDownloadEnd(String accountName)

    public void deltaNeutralValidation(int reqId, UnderComp underComp)

    public void execDetailsEnd(int reqId)

    public void openOrderEnd()
}
```

15.3 Usage example

Let us use the Java connector object to implement the Arrival Price algo example that is provided in the official IB Java API.[63]

This Arrival Price example shows how easy it is to convert Java code available in the official API or support forums (or even code supplied by IB's API customer support team) to Matlab using **IB-Matlab**:

First, here is the original Java code:

```
import com.ib.client.TagValue;

Contract m_contract = new Contract();

Order m_order = new Order();

Vector<TagValue> m_algoParams = new Vector<TagValue>();

/** Stocks */
m_contract.m_symbol = "MSFT";
m_contract.m_secType = "STK";
m_contract.m_exchange = "SMART";
m_contract.m_currency = "USD";

/** Arrival Price */
m_algoParams.add( new TagValue("maxPctVol","0.01") );
m_algoParams.add( new TagValue("riskAversion","Passive") );
m_algoParams.add( new TagValue("startTime","9:00:00 EST") );
m_algoParams.add( new TagValue("endTime","15:00:00 EST") );
m_algoParams.add( new TagValue("forceCompletion","0") );
m_algoParams.add( new TagValue("allowPastEndTime","1") );

m_order.m_action = "BUY";
m_order.m_totalQuantity = 1;
m_order.m_orderType = "LMT";
m_order.m_lmtPrice = 0.14
m_order.m_algoStrategy = "ArrivalPx";
m_order.m_algoParams = m_algoParams;
m_order.m_transmit = false;

m_client.placeOrder(40, m_contract, m_order);
```

[63] http://www.interactivebrokers.com/en/trading/orders/arrivalprice.php, http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/tables/ibalgo_parameters.htm

And now for the corresponding Matlab code (notice how closely it resembles the original Java code):[64]

```
import com.ib.client.TagValue;

% First create the contract for the requested security
m_contract = ibConnectionObject.createContract(...
                 'MSFT','STK','',0,'','SMART','USD','MSFT');

% Alternately, we could have done as follows:
m_contract = ibConnectionObject.createContract(...
                 '','','',0,'','','','');
m_contract.m_symbol   = 'MSFT';
m_contract.m_secType  = 'STK';
m_contract.m_exchange = 'SMART';
m_contract.m_currency = 'USD';

% Now set the Arrival Price algoParams
m_algoParams = java.util.Vector;
m_algoParams.add( TagValue('maxPctVol','0.01') );
m_algoParams.add( TagValue('riskAversion','Passive') );
m_algoParams.add( TagValue('startTime','9:00:00 EST') );
m_algoParams.add( TagValue('endTime','15:00:00 EST') );
m_algoParams.add( TagValue('forceCompletion','0') );
m_algoParams.add( TagValue('allowPastEndTime','1') );

% Now create the order, using algoParams
m_order = ibConnectionObject.createOrder(...
                 'BUY', 1, 'LMT', 0.14, 0, '', ...
                 '', 0, '', '', realmax, 0, false);

m_order.m_algoStrategy = 'ArrivalPx';
m_order.m_algoParams = m_algoParams;
m_order.m_transmit = false;

% Finally, send the order to the IB server
ibConnectionObject.placeOrder(40, m_contract, m_order);
```

Note: A related mechanism is explained in section 9.5 above.

[64] This code can be downloaded from: http://UndocumentedMatlab.com/files/IBMatlab_ArrivalPriceAlgo.m

16 Sample model using IB-Matlab – Pairs Trading

16.1 Once a day - decide whether two securities are co-integrated

1. Download http://www.spatial-econometrics.com/ (jplv7.zip) and unzip into a new folder.

2. Add the new toolbox function to you Matlab path (naturally, use the actual folder name in which you've unzipped the toolbox, rather than my C:\... example):

```
pathAdditions = genpath('C:\SpatialEconometricsToolbox');
addpath(pathAdditions);
```

3. Use IB-Matlab to get daily historical data (closing price) on both securities for the past year:[65]

```
IBM_history  = IBMatlab('action','history', 'Symbol','IBM', ...
                        'DurationValue',1,'DurationUnits','Y', ...
                        'BarSize','1 day');
IBM_close_prices = IBM_history.close';  % array of 252 values

GOOG_history = IBMatlab('action','history', 'Symbol','GOOG', ...
                        'DurationValue',1,'DurationUnits','Y', ...
                        'BarSize','1 day');
GOOG_close_prices = GOOG_history.close';  % array of 252 values
```

4. Use *adf()* on each of the securities to ensure that they are NOT mean-trending:

5. Use *cadf()* on both securities to ensure that they ARE cointegrated => returns flag, β:

```
modelData.beta = ...
```

6. If any of the securities failed any of the tests, bail out

```
if ...
   return;
end
```

7. Store STD = *std(*$Close_{sec1} - \beta * Close_{sec2}$) for later use in the runtime part below

```
modelData.std = std(IBM_close_prices - ...
                    modelData.beta * GOOG_close_prices);
```

8. Use IB-Matlab to stream quote data for the two securities:[66]

---

[65] See section 6 for details

[66] See section 7 for details

```
modelData.ids(1) = IBMatlab('action','query', 'Symbol','IBM', ...
                            'QuotesNumber',inf);
modelData.ids(2) = IBMatlab('action','query', 'Symbol','GOOG', ...
                            'QuotesNumber',inf);
```

9. Use IB-Matlab to attach our user-defined callback processing function:[67]

```
IBMatlab('action','query', 'Symbol','GOOG', ...
         'CallbackTickPrice',{@IBMatlab_CallbackTickPrice,modelData});
```

16.2 Runtime – process TickPrice streaming-quote events

```
function IBMatlab_CallbackTickPrice(ibConnector, eventData, modelData)

   persistent lastPrice1 lastPrice2

   if (eventData.field == 1)  % ==com.ib.client.TickType.BID
      if (eventData.price > 0)  % disregard invalid values
         if (eventData.tickerId == modelData.ids(1))
            lastPrice1 = eventData.price;
            ignoreFlag = false;
         elseif (eventData.tickerId == modelData.ids(2))
            lastPrice2 = eventData.price;
            ignoreFlag = false;
         else
            % ignore - not one of the requested symbols pair
            ignoreFlag = true;
         end

         % Check whether the monitored symbols have diverged from their
         % steady-state prices in either direction
         if ~ignoreFlag && ~isempty(lastPrice1) && ~isempty(lastPrice2)
            deltaPrice = lastPrice1 - modelData.beta*lastPrice2;
            if deltePrice < -2*modelData.std
               % GOOG overbought vs. IBM, so buy IBM, sell GOOG
               IBMatlab('action','BUY',  'Quantity',1, 'Type','MKT',...
                        'Symbol','IBM');
               IBMatlab('action','SELL', 'Quantity',1, 'Type','MKT',...
                        'Symbol','GOOG');
            elseif deltePrice > 2*modelData.std
               % GOOG oversold vs. IBM, so sell IBM, buy GOOG
               IBMatlab('action','SELL', 'Quantity',1, 'Type','MKT',...
                        'Symbol','IBM');
               IBMatlab('action','BUY',  'Quantity',1, 'Type','MKT',...
                        'Symbol','GOOG');
            end
         end
      end
   end  % IBMatlab_CallbackTickPrice
```

[67] See section 11 for details

Appendix – Official IB resources

- API download page - http://www.interactivebrokers.com/en/p.php?f=programInterface&p=a&ib_entity=llc
- API Reference Guide - http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm
- Online Reference Guide – Java - http://www.interactivebrokers.com/php/apiUsersGuide/apiguide.htm#apiguide/java/java_eclientsocket_methods.htm
- Java API Quick Reference (PDF) - http://www.interactivebrokers.com/download/inst/JavaAPIQuickReference.pdf
- Getting Started with the TWS Java API (PDF) - http://www.interactivebrokers.com/download/JavaAPIGettingStarted.pdf
- Getting Started with the TWS Java API for Advisors (PDF) - http://www.interactivebrokers.com/download/GettingStartedJavaAPIAdvisors.pdf
- Java API Samples for the Getting Started Guide (ZIP) - http://www.interactivebrokers.com/download/JavaAPIExamples.zip
- Full Reference Guide (PDF) - http://www.interactivebrokers.com/download/newMark/PDFs/APIprintable.pdf
- API Release Notes - http://www.interactivebrokers.com/en/p.php?f=programInterface#notes-clear
- API Webinars - http://www.interactivebrokers.com/en/general/education/webinars.php?p=a&ib_entity=llc
- Discussion Forum - http://www.interactivebrokers.com/en/p.php?f=ibchat_bb&ib_entity=llc
- API customer service and technical support - api@interactivebrokers.com