IB-Matlab User Guide

Version 1.52

October 6, 2014

Fully compatible with IB version 9.71

http://UndocumentedMatlab.com/IB-Matlab

Undocumented Matlab

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

Table of Contents

DISCLAIMER

1 Introduction

Interactive Brokers (*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 *IB Gateway* is installed together with TWS. IB Gateway has no 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.

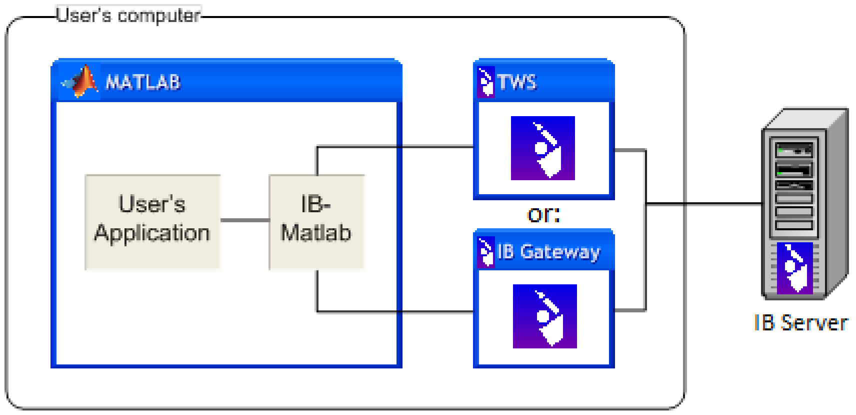
In fact, Java's cross-platform compatibility means that exactly the same **IB-Matlab** code runs on all platforms supported by both IB and Matlab, namely Windows (both 32 and 64 bit), Macs and Linux/Unix.

---

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

[2] http://www.interactivebrokers.com/en/software/api/apiguide/api/run_the_api_through_the_ib_gateway.htm

[3] http://www.interactivebrokers.com/en/software/ibapi.php



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**, your version may be missing some features discussed in this document. However, as part of your annual license renewal you will receive the latest **IB-Matlab** version, including all the functionality detailed here.

[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 and licensing

IB-Matlab requires the following in order to run:

1. An active account at IB

   Use of IB's Demo account is not recommended: it is limited in comparison with the functionalities of a live account. To test **IB-Matlab**, we recommend using the paper-trade (simulated trading) account that IB assigns when you register to IB. Paper-trade accounts resemble your live account more closely than the 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 in 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 the **IB-Matlab** files are in *C:\IB-Matlab\*, then the new line in the *classpath.txt* file should be as follows:

   ```
   C:\IB-Matlab\IBMatlab.jar
   ```

   You must use the full absolute filepath. So on MacOS, for example, enter */Users/Yair/IB-Matlab/IBMatlab.jar* rather than *~/IB-Matlab/IBMatlab.jar*. Similarly, you cannot use something like *$matlabroot/../../IB-Matlab.jar*.

When trying to save the *classpath.txt* file, you might get an error message saying the file is read-only. To solve this, enable write-access to this file: In Linux/Unix, run *chmod a+w classpath.txt*. In Windows, open Windows Explorer, right-click the *classpath.txt* file, select "Properties", and unselect the "Read-only" attribute. In Windows 7/8 you need to run Matlab as Administrator (right click the Matlab icon and select "Run as Administrator") in order to be able to save the file, even when it is not read-only.

If you cannot get administrator access to modify *classpath.txt*, place a copy of it in your Matlab startup folder. This is the folder used when you start Matlab – type the following command at the Matlab command prompt to get it:

```
>> 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.

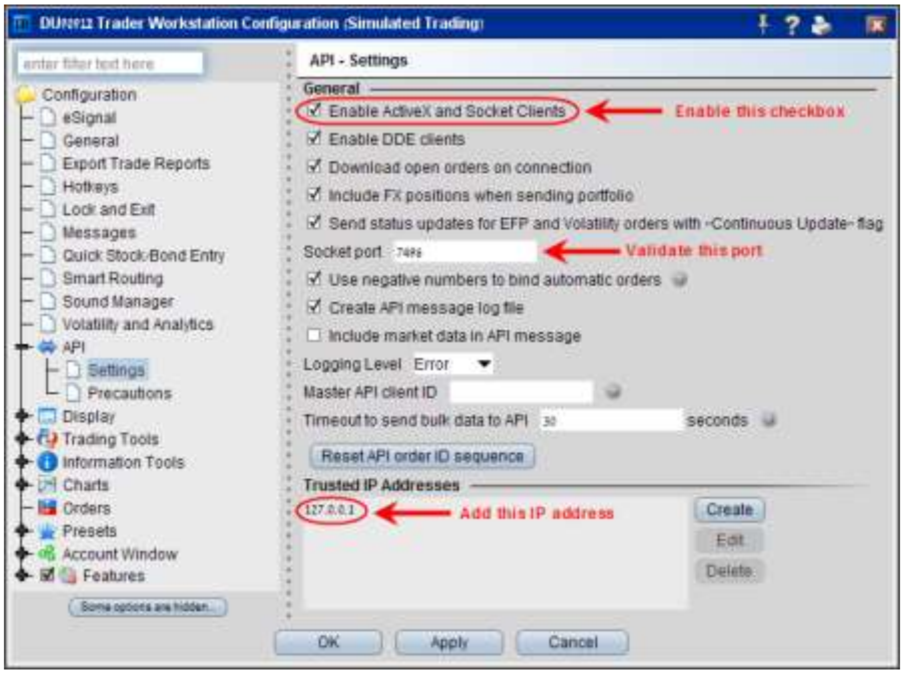
For these reasons, a much safer approach is to update the *classpath.txt* file in Matlab's default location, namely the %matlabroot%/toolbox/local/ folder.

6. Restart Matlab (no need to restart the computer or to run as administrator)
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. See §2.1 below for licensing details.
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) and 0:0:0:0:0:0:0:1 (the Ipv6 loopback address, used by IB-Gateway) 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 then you will need to specify the Port parameter when you use *IBMatlab* for the first time after each Matlab restart.[9]

[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**.

[9] See §13 below for more details

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]



API - Settings

General

- ✓ Enable ActiveX and Socket Clients
- ✓ Enable DDE clients
- ✓ Download open orders on connection
- ✓ Include FX positions when sending portfolio
- ✓ Send status updates for EFP and Volatility orders with «Continuous Update» flag

Socket port 7496

- ✓ Use negative numbers to bind automatic orders
- ✓ Create API message log file
- ☐ Include market data in API message

Logging Level Error

Master API client ID

Timeout to send bulk data to API 30 seconds

Trusted IP Addresses

127.0.0.1

e. If you have recently upgraded from a 32-bit system (e.g., Windows XP) to 64 bits (e.g., Windows 7), then if you encounter some problems running **IB-Matlab**, this could be due to a 32-64 bit mixup in TWS.[11]

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).[12]

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 §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.

[10] See §13 below for more details. Note: It has been reported that setting a value in this field may limit the ability to concurrently access multiple IB accounts, as may be the case for Financial Advisors or if you have child IB accounts.

[11] See http://www.elitetrader.com/vb/showthread.php?threadid=175081 for details on how to solve this.

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

2.1 Licensing alternatives

IB-Matlab's license uses an activation that is specific to the installed computer. This uses a unique fingerprint hash-code that is reported by the Operating System, which includes the Windows ID (on Windows systems), computer name, and the list of MAC addresses used by the computer.

There are two licensing variants – you can choose between either of them when you purchase **IB-Matlab**:

1. A standard license that is tied to a specific computer configuration. Once the computer's license is activated, the activation key is stored on the local hard-disk. This ensures fast performance of **IB-Matlab**, since it does not need to continuously check the license online.

   On the other hand, a possible drawback on Windows systems is that if your hard-disk crashes and you reinstall on a new hard-disk, then the Operating System might report a different hash-code that is not activated, and **IB-Matlab** will not run. In such a case you will need to purchase a new license. If run-time performance is less important to you than protection from such a potential disk crash, then consider using the online license (#2 below).

2. A license that is similarly tied to a specific computer but validates online whenever **IB-Matlab** connects to IB (i.e., at the beginning of an IB/TWS session). The user can always switch the activation to another computer, even if the disk crashed. The down-side is that checking the license online takes a second or two and therefore affects **IB-Matlab**'s run-time performance.

Note: you cannot switch between license variants after initial activation, so choose carefully. However, you can switch variants whenever you renew your annual license.

A corollary of the computer fingerprint is that whenever you modify any component that affects the fingerprint, **IB-Matlab** will stop working. This could happen if you reinstall the operating system (OS), modify the computer name, change network adapters (e.g., switch between wifi/cellular/wired connection, or use a new USB networking device), manually modify MAC addresses, or use VPN software that uses a dynamic MAC and does not property register as a VPN service. In such cases, you might see an error message such as the following when you try to use **IB-Matlab**:

```
Error using IBMatlab/ensureConnected (line 985)
IBMatlab is not activated on this computer (step 3).
Perhaps you are using a new network device or computer name.
To fix this, revert your changes, or send the following data to
Yair Altman (altmany@gmail.com): [*** (fingerprint code) ***]
```

To fix such cases, simply revert back to the original hardware/networking configuration, and then **IB-Matlab** will resume working. If you wish to make the configuration change permanent, then you can contact us for an activation switch to the new configuration (see the following section 2.2 for details).

The standard **IB-Matlab** license is for a single calendar year from date of purchase. Additional licensing options are available; contact us for pricing information:

- **3 or 6-months** usage: these short-term licenses can be repeatedly renewed. However, if and when you decide to get a full-year license, then you will need to purchase the full license (i.e., not the annual renewal).
- **Multi-year** license: 3-year, 5-year, 8-year or perpetual (49-year) extended maintenance. All these extended maintenance options include bug-fixing support for 3 years only, but they will work for much longer than the regular one year and will not require annual re-licensing.
- **Volume (multi-computer)** license: this is basically the same license as a single computer license, but when you purchase multiple licenses at the same time you get a volume discount.
- **Source-code** license: this license is unlimited in duration, can be installed on an unlimited number of computers within your organization, and requires no activation. This is naturally the most expensive license and requires signing a dedicated non-disclosure agreement (NDA).

2.2 Switching activated computers

You can switch the **IB-Matlab** license activation between computers or computer hardware configurations (i.e., fingerprint hash-code) up to 3 times at no cost. A small handling fee will be incurred for additional re-activations, following the initial 3.

In order to change activation to another computer, **IB-Matlab** needs to be deactivated on the existing computer, using a procedure that will be explained to you when you will request the activation switch. Email us for detailed instructions. The deactivation procedure will be run locally on your installed computer, and the reported confirmation code should be emailed back to us for validation. An activation switch can take up to 48 hours, depending on your request time (e.g., longer during the weekend), but is usually completed within a few hours during regular European business hours. During this time you will not be able to use **IB-Matlab**.

Deactivating an **IB-Matlab** license is only necessary if you use licensing variant #1 (i.e., activation on the local hard-disk, not online). In such cases, it is important that you do NOT format your disk or make other similar changes that will prevent the **IB-Matlab** deactivation. If you need to make such hardware changes, first contact us in order to receive the deactivation procedure, follow the procedure to deactivate **IB-Matlab**, wait for our confirmation, and then proceed with your hardware changes.

If you use the online activation variant (#2 in the preceding section), then it is not necessary to deactivate the existing **IB-Matlab** license. Just contact us with the new configuration's fingerprint hash-code and we will make the switch on the server-side.

3 Using IBMatlab

IB-Matlab uses the IB client (either TWS or IB Gateway) to connect to the IB server. Therefore, to use **IB-Matlab**, you must first ensure that either TWS or Gateway is active. If an active IB client is not detected, **IB-Matlab** will automatically attempt to start TWS and to connect to it. Note that this may not work for some TWS installations. You can always start TWS or IB Gateway manually, before running **IB-Matlab**. In any case, if an IB connection is unsuccessful, **IB-Matlab** will error.

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.[13]

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.

Note that if you choose to use the struct format and then to reuse this struct for different *IBMatlab* commands (by altering a few of the parameters), then the entire set of struct parameters is used, possibly including some leftover parameters from previous *IBMatlab* commands, that may lead to unexpected results. For example:

```
% 1st IBMatlab command - buy 10 GOOG at limit $600.00
>> params = [];  % initialize
>> params.Action      = 'buy';
>> params.Symbol      = 'GOOG';
>> params.Quantity    = 10;
>> params.Type        = 'LMT';
>> params.LimitPrice  = 600.00;
>> orderId1 = IBMatlab(params);
```

[13] See §15 below for more details

```
% 2nd IBMatlab command - sell 10 GOOG at limit $625.00
>> params.Action      = 'sell';   % reuse the existing params struct
>> params.LimitPrice  = 625.00;
>> orderId1 = IBMatlab(params);
```

The second *IBMatlab* command will sell all 10 units of GOOG, even if the user meant to sell just a single unit. This is because the `params` struct still contains the Quantity=10 field from the first *IBMatlab* command. To avoid unexpected results, I therefore advise to re-initialize the `params` struct (=[]) for each *IBMatlab* command.

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.

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:[14]

```
>> 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
```

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

```
                 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
```

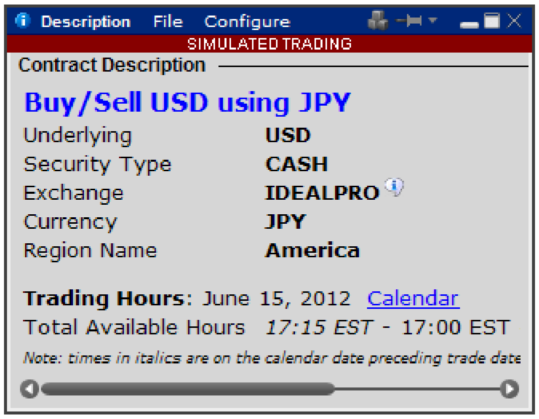
In other cases, we may also need to explicitly specify the Currency (default='USD'). For example, the Russell 2000 index (RUT) is listed on the Toronto Stock Exchange (TSE) and uses ‘CAD’ Currency.

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).

If you are uncertain about a security's contract details, try to test different parameter combinations. Alternately, use your TWS paper-trade (simulated trading) account to buy a virtual unit of the security, right-click the ticker in TWS and select "Contract Info / Description":

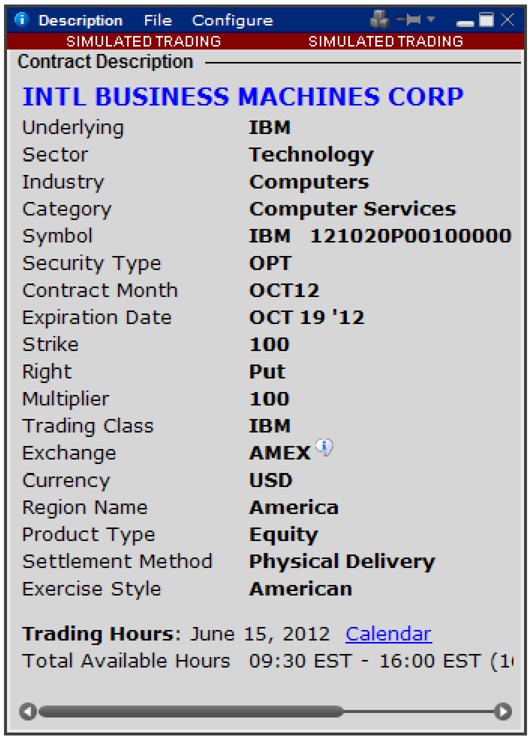


Contract Description

Buy/Sell USD using JPY

Underlying	**USD**
Security Type	**CASH**
Exchange	**IDEALPRO**
Currency	**JPY**
Region Name	**America**

Trading Hours: June 15, 2012 Calendar
Total Available Hours *17:15 EST* - 17:00 EST
Note: times in italics are on the calendar date preceding trade date


Description File Configure
SIMULATED TRADING SIMULATED TRADING

Contract Description

INTL BUSINESS MACHINES CORP

Underlying	**IBM**
Sector	**Technology**
Industry	**Computers**
Category	**Computer Services**
Symbol	**IBM 121020P00100000**
Security Type	**OPT**
Contract Month	**OCT12**
Expiration Date	**OCT 19 '12**
Strike	**100**
Right	**Put**
Multiplier	**100**
Trading Class	**IBM**
Exchange	**AMEX**
Currency	**USD**
Region Name	**America**
Product Type	**Equity**
Settlement Method	**Physical Delivery**
Exercise Style	**American**

Trading Hours: June 15, 2012 Calendar
Total Available Hours 09:30 EST - 16:00 EST (1

Contract descriptions for USD.JPY and an IBM option, as reported in TWS

This specific example shows that the LocalSymbol for the IBM OCT12 PUT option is 'IBM 121020P001000000' (Symbol remains 'IBM'). Note that this LocalSymbol has multiple spaces. For this reason it is better to copy-paste the value directly from the window, rather than to copy it manually.

Alternately, buy a virtual unit of the security as above, then use IB-Matlab to read the portfolio (see §4 below) and check the reported contract data. For example:

```
>> data = IBMatlab('action','portfolio');
>> data(3)
ans =
          symbol: 'EUR'
     localSymbol: 'EUR.USD'
        exchange: 'IDEALPRO'
         secType: 'CASH'
        currency: 'USD'
           right: '0'
             ...
```

As a last resort, contact IB's API customer support help-desk (see Appendix A.1 below) to request the necessary parameters for a particular security.

Here are some examples of IB symbols:[15]

Symbol	Type	Description
CO	Stock	Cisco Corporation, NASDAQ
GE	Stock	General Electric, NYSE
VOD-LSE	Stock	Vodafone Group, London Stock Exch.
ESM1-GLOBEX-FUT	Future	Emini ES Jun 2011 futures, Globex
QQQFJ-CBOE-OPT	Option	Jun 2004, 36.0 CALL option QQQFJ
SPX-CBOE-IND	Index	S&P-500 Index, CBOE
INDU-NYSE-IND	Index	Dow Jones Industrials Index, NYSE
YM JUN 11-ECBOT-FUT	Future	YM Jun 2011 future, ECBOT Note: 3 spaces between the symbol and month,1 space between month and year
QMN5-NYMEX-FUT	Future	QM (Crude) June 2005 future contract, NYMEX
FGBL DEC 11-DTB-FUT-EUR	Future	German Dec 2011 Bund future
XAUUSD-SMART-CMDTY	Commodity	London Gold Spot
EUR.USD-IDEAL-CASH	Cash Forex	EURUSD currency pair, IDEAL
EUR.USD-IDEALPRO-CASH	Cash Forex	EURUSD currency pair, IDEALPRO

Traders who wish to see the full option chain list, are referred to §11.4 below. While it is not possible to receive the entire list of option *prices* in a single command (each market price requires a separate request with a specific LocalSymbol), it is possible to extract the full list of option LocalSymbols in a single command.

[15] http://www.amibroker.com/ib.html (scroll down to the SYMBOLOGY section)

Here is the full list of Contract properties supported by *IBMatlab*:[16]

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' – stocks (default) • 'OPT' – options • 'FUT' – futures • 'IND' – indexes • 'FOP' – options on futures • 'CASH' – Forex • 'BAG'
Exchange	string	'SMART'	The exchange that should process the request. SMART uses IB's SmartRouting to optimize order execution time and cost.[17] To specify the primary exchange, use the : or / separator,[18] for example: 'SMART:ISLAND' or 'SMART/TSE'.
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. Because 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'

Important: 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 might get confused when trying to interpret the string `'5.20'` as a number, and very odd results might happen.

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

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

[18] This is supported by TWS 9.70 or newer: https://www.interactivebrokers.com/en/?f=/en/software/apiReleaseNotes/api970.php

4 Querying account and portfolio data

4.1 Account information

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

   (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 a specific field use standard Matlab dot notation:

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

Some account data fields have several variants. For example, different values may be returned by IB server for NetLiquidation, NetLiquidation_C and NetLiquidation_S. Until you trade a commodity, NetLiquidation_C=0 and NetLiquidation_S= NetLiquidation. After trading a commodity, NetLiquidation_C holds the value for commodities, NetLiquidation_S for securities, and NetLiquidation for the total. Several other fields also have these _S and _C variants.[19]

If your TWS account is linked to more than one IB account (as is common for Financial Advisors, for example), then you should specify the AccountName input parameter, so that IB would know which IB account to access:[20]

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

To get the account information for all accounts in a consolidated manner, set AccountName to 'All':[21]

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

[19] **IB-Matlab** versions prior to 5/5/2012 did not make a distinction between *Field*, *Field_S* and *Field_C*, and so the reported value in *Field* might be any of the three possibilities.

[20] If you don't specify the AccountName, you will get stale or empty account data.

[21] Note: The TWS/IB-Gateway API setting "Master API Client ID" may need to be empty (even if correct) for this to work (see installation step 9d in §2 above)

4.2 Portfolio data

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
    contract
```

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(2)
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
       contract: [1x1 struct]
```

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.[22] The workarounds are to simply re-request the information, and to force a programmatic reconnection to IB (more on the Java connector in §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
   pause(1);                                % let IB cool down a bit
   data = IBMatlab('action','portfolio');  % will automatically reconnect
end
```

As in the case of retrieving the account data, when requesting portfolio data we need to ensure that we are requesting the data for the correct account, if we have more than a single IB account connected to our IB login. Many frustrations can be avoided by specifically stating the requested AccountName parameter whenever we use *IBMatlab* in multi-account environments. If you are unsure of the account name, you can use AccountName='All'.

A suggested practice for safe/robust programming is to compare the sum of the non-cash portfolio `marketValue`s versus the account's `StockMarketValue`, as reported by the `IBMatlab('action','account_data')` command. Be careful to sum only non-cash securities (i.e., `~strcmpi(data.secType,'cash')`).

Shorted securities will appear with a negative `marketValue` in the portfolio struct array, while long securities will have a positive value. The sum of these values, which should be equal to the account's `StockMarketValue`, may be positive or negative, indicating whether the overall portfolio is long or short. If you are only interested in the total monetary amount of the security positions (i.e., their absolute `marketValue`s), then the sum in this case should equal the account's `GrossPositionValue`.

Note that small differences between the portfolio `marketValue` sums and the account `StockMarketValue` or `GrossPositionValue`, due to market changes that occurred between the time that the account data was collected, compared to the time that the portfolio data was requested. If you run these requests immediately one after another, then in the vast majority of the cases the data will match exactly.

[22] 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

5.1 Single-quote 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 00:47:24'
   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
        contract: [1x1 struct]
 contractDetails: [1x1 struct]
```

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
```

Note: the `reqTime`, `dataTime` and `dataTimestamp` fields reflect the local time. If the `lastPrice` field is returned with valid data (not -1) then it is usually accompanied by a `lastTimestamp` field that reflects the server time in Java units (seconds since midnight 1/1/1970[23] as a string, for example: '1348563149'). We can convert `lastTimestamp` into Matlab format by converting the string into a number and using ***datenum***:[24]

```
>> datestr(datenum([1970 1 1 0 0 str2num(data.lastTimestamp)]))
ans =
25-Sep-2012 08:52:29
```

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

1. The IB account is subscribed to the information service for the stated 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)

[23] http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Date.html (note the units [seconds/milliseconds] – this can be tricky…)

[24] http://www.mathworks.com/support/solutions/en/data/1-9B9H2S/

If any of these conditions is not met, the information returned by IB will be empty/invalid (the data fields will have a value of -1 or `[]`). In some cases, **IB-Matlab** automatically attempts to re-fetch the data from IB, to ensure that the data is in fact missing. If condition 3 is not met, the empty data will not be accompanied by any error message; if condition 1 and/or 2 (but not 3) is not met, an error message will be displayed in the Matlab command window,[25] as the following examples illustrate:

```
>> 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 discussion in §3).

In some cases, querying a contract may return some valid (>0) fields, but not others. For example, NIFTY50 is an index that can be queried but only returns valid close and lastPrice fields; other fields return -1. The reason is that NIFTY50 is not a tradable security by itself, so it does not have bid/ask/open/high/low/volume values. Only futures on the index are tradable securities, and these indeed return valid field values.

Another common reason for receiving -1 field values is when you query a security when the market is closed.

Additional market data about a security can be retrieved 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 '100,101'

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

or '236'.[26] Note that the 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. Only some securities and exchanges support this feature (mainly US stocks), and only for streaming quotes[27] (not for regular market queries). When the feature is supported, an additional `shortable` field is returned with basic information about the security's shortability.[28]

5.2 Scanner data

Scanner data returns a list of securities that match the specified scan criteria. IB includes a long list of predefined scanners,[29] including MOST_ACTIVE, TOP_PERC_GAIN, HOT_BY_VOLUME, HOT_BY_PRICE etc. The scan can be limited to security type and trading location,[30] as well as to a variety of criteria on the security attributes (price, volume, maturity date, market cap, Moody/S&P rating, coupon rate etc.).[31] This is an extensive list, which covers much of the demand.

Note: IB scanners are only limited to the predefined scanner types and options above. We cannot define a generic scan criteria based on attributes that are not on the predefined list. In such cases, we would need to use other means to scan the markets.

To use market scanners in **IB-Matlab**, set the Action parameter to 'Scanner', and either use the default criteria values (see table below) or override them. For example, to return the currently most active stock in NASDAQ (the default criteria):

```
>> data = IBMatlab('action','scanner')
data = 
          EventName: 'scannerData'
              reqId: 349661732
               rank: 0
    contractDetails: [1x1 struct]
           distance: []
          benchmark: []
         projection: []
            legsStr: []
             symbol: 'QQQ'
        localSymbol: 'QQQ'
           contract: [1x1 struct]
```

[26] http://www.interactivebrokers.com/en/software/api/apiguide/tables/generic_tick_types.htm

[27] See §7 for details on streaming quotes

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

[29] http://www.interactivebrokers.com/en/software/api/apiguide/tables/available_market_scanners.htm

[30] http://www.interactivebrokers.com/en/software/api/apiguide/tables/instruments_and_location_codes_for_market_scanners.htm

[31] http://www.interactivebrokers.com/en/software/api/apiguide/java/scannersubscription.htm

Additional information about the returned security (QQQ in this case) can be seen in the `contract` and `contractDetails` fields of the returned data structure.

By default, **IB-Matlab** only returns the top single security matching the scan criteria. We can change this using the NumberOfRows parameter. IB limits the amount of returned data, so it is quite possible that we will receive fewer results than requested:

```
>> data = IBMatlab('action','scanner', 'NumberOfRows',100)
data =
1x23 struct array with fields:
    EventName
    reqId
    rank
    contractDetails
    distance
    benchmark
    projection
    legsStr
    symbol
    localSymbol
    contract

>> data(end)
ans =
          EventName: 'scannerData'
              reqId: 349662602
               rank: 22
    contractDetails: [1x1 struct]
           distance: []
          benchmark: []
         projection: []
            legsStr: []
             symbol: 'AMZN'
        localSymbol: 'AMZN'
           contract: [1x1 struct]
```

The most important parameters for scanner requests are Instrument (default value: 'STK'), LocationCode (default value: ‘STK.NASDAQ’) and ScanCode (default value: ‘MOST_ACTIVE’). Additional parameters are listed at the end of this section.

Note: the IB documentation about the possible scanner parameter values is quite limited and incomplete. If you are unsure of the parameter values that are required for a specific scan, contact IB’s customer service and ask them for the specific set of *API ScannerSubscription parameters* that are required for your requested scan.

One feature that could help in determining the possible parameter values is an XML document that the IB server provides which describes the possible combinations. We can retrieve this document by specifying Type='parameters' in **IB-Matlab**:

```
>> xmlStr = IBMatlab('action','scanner', 'Type','parameters')
xmlStr =
<?xml version="1.0" encoding="UTF-8"?>
<ScanParameterResponse>
      <InstrumentList varName="instrumentList">
            <Instrument>
                  <name>US Stocks</name>
                  <type>STK</type>
                  <filters>PRICE,PRICE_USD,VOLUME,VOLUME_USD,AVGVOL
                  UME,AVGVOLUME_USD,HALTED,...,FIRSTTRADEDATE,HASOP
                  TIONS</filters>
                  <group>STK.GLOBAL</group>
                  <shortName>US</shortName>
            </Instrument>
            <Instrument>
                  <name>US Futures</name>
                  <type>FUT.US</type>
                  <secType>FUT</secType>
                  <filters>PRICE,PRICE_USD,VOLUME,VOLUME_USD,PRODCA
                  T,LEADFUT,CHANGEPERC,CHANGEOPENPERC,OPENGAPPERC,P
                  RICERANGE,TRADERATE</filters>
                  <group>FUT.GLOBAL</group>
                  <shortName>US</shortName>
            </Instrument>
            ... (~20K additional lines!)
```

This XML string is quite long (~1MB, ~20K lines). We can store it in a **.xml* file and open this file in an XML reader (for example, a browser). Alternatively, we can ask **IB-Matlab** to parse this XML and present us with a more manageable Matlab struct that we can then process in Matlab. This is done by setting ParametersType='struct'. Note that this XML parsing could take a long time (a full minute or even longer):

```
>> params = IBMatlab('action','scanner', 'type','parameters', ...
                     'ParametersType','struct')  %may take a long time!
params =
                        Name: 'ScanParameterResponse'
              InstrumentList: [1x1 struct]
                LocationTree: [1x1 struct]
                ScanTypeList: [1x2 struct]
                 SettingList: [1x1 struct]
                  FilterList: [1x2 struct]
           ScannerLayoutList: [1x1 struct]
         InstrumentGroupList: [1x1 struct]
     SimilarProductsDefaults: [1x1 struct]
    MainScreenDefaultTickers: [1x1 struct]
                  ColumnSets: [1x1 struct]
      SidecarScannerDefaults: [1x1 struct]

>> params.InstrumentList
ans =
          Name: 'InstrumentList'
    Attributes: [1x1 struct]
    Instrument: [1x23 struct]
```

```
>> params.InstrumentList.Instrument(2)
ans =
            Name: 'Instrument'
            name: 'US Futures'
            type: 'FUT.US'
         filters: [1x108 char]
           group: 'FUT.GLOBAL'
       shortName: 'US'
         secType: 'FUT'
    nscanSecType: []
     permSecType: []

>> params.InstrumentList.Instrument(2).filters
ans =
PRICE,PRICE_USD,VOLUME,VOLUME_USD,PRODCAT,LEADFUT,CHANGEPERC,CHANGEOPEN
PERC,OPENGAPPERC,PRICERANGE,TRADERATE
```

The parameters that affect scanner data retrieval closely mirror those expected by the Java API:[32]

Parameter	Data type	Default	Description
Type	String	'Scan'	One of: • 'Scan' – scanner data (default) • 'Parameters' – possible scanner param values
ParametersType	String	'XML'	One of: • 'XML' (default) • 'Struct' – Matlab struct
AbovePrice	Number	0.0	Filter out contracts with a price lower than this value
AboveVolume	Integer	0	Filter out contracts with a volume lower than this value
AverageOptionVolume Above	integer	0	Filter out contracts with avg. options volume lower than this
BelowPrice	number	Inf	Filter out contracts with a price higher than this value
CouponRateAbove	number	0.0	Filter out contracts with a coupon rate lower than this
CouponRateBelow	number	Inf	Filter out contracts with a coupon rate higher than this
ExcludeConvertible	string	'' (empty string)	Filter out convertible bonds

[32] http://www.interactivebrokers.com/en/software/api/apiguide/java/scannersubscription.htm

Parameter	Data type	Default	Description
Instrument	string	'STK'	Defines the instrument type[33]
LocationCode	string	'STK.NASDAQ'	Defines the scanned markets[34]
MarketCapAbove	number	0.0	Filter out contracts with a market cap lower than this
MarketCapBelow	number	Inf	Filter out contracts with a market cap above this value
MaturityDateAbove	string	'' (empty string)	Filter out contracts with a maturity date earlier than this
MaturityDateBelow	string	'' (empty string)	Filter out contracts with a maturity date later than this
MoodyRatingAbove	string	'' (empty string)	Filter out contracts with a Moody rating below this value
MoodyRatingBelow	string	'' (empty string)	Filter out contracts with a Moody rating above this value
NumberOfRows	integer	1	The number of rows of data to return for the query
ScanCode	string	'MOST_ACTIVE'	A long list.. – see the API doc[35]
ScannerSettingPairs	string	'' (empty string)	For example, a pairing 'Annual, true' used on the "Top Option Implied Vol % Gainers" scan returns annualized volatilities
SPRatingAbove	string	'' (empty string)	Filter out contracts with an S&P rating below this value
SPRatingBelow	string	'' (empty string)	Filter out contracts with an S&P rating above this value
StockTypeFilter	string	'ALL'	One of: • 'ALL' (default) • 'CORP' • 'ADR' • 'ETF' • 'REIT' • 'CEF'

[33] http://www.interactivebrokers.com/en/software/api/apiguide/tables/instruments_and_location_codes_for_market_scanners.htm (note: this is only a partial list!)

[34] http://www.interactivebrokers.com/en/software/api/apiguide/tables/instruments_and_location_codes_for_market_scanners.htm (note: this is only a partial list!)

[35] http://www.interactivebrokers.com/en/software/api/apiguide/tables/available_market_scanners.htm

6 Querying 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.[36] Note that these are IB server limitations, not **IB-Matlab** limitations. As of July 2013, these limitations include:

1. Historical data is limited to 2000 results (data bars) – if more than that is requested, the entire request is dropped.
2. Historical data can by default only be requested for the past year. If you have purchased additional concurrent real-time market data-lines from IB, then you can ask for historical data up to 4 years back. If you request data older than your allowed limit, the entire request is dropped.
3. Historical data requests that use a bar size of 30 seconds or less can only go back six months. If older data is requested, the entire request is dropped.
4. Requesting identical historical data requests within 15 seconds is prohibited. **IB-Matlab** will automatically return the previous results in such a case.
5. Requesting 6+ historical data requests for the same Contract, Exchange and Tick Type within 2 seconds is prohibited – the entire request will be dropped.
6. Requesting 60+ historical data requests of any type within 10-minutes is prohibited – the entire request will be dropped.
7. 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

[36] http://www.interactivebrokers.com/en/software/api/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 (=60 days) 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 =
         atenum: [1x7 double]
       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:[37]

- `atenum` – a numeric array of date/time values in Matlab's numeric format[38]
- `dateTime` – a cell-array of date strings, or a numeric array of date values in IB format (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 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

[37] http://www.interactivebrokers.com/en/software/api/apiguide/java/historicaldata.htm#HT_historicalDataw

[38] http://www.mathworks.com/help/matlab/ref/datenum.html

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:[39]

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[40])
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 – default) • '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 (default) • 2/3/5/15/30 mins • 1 hour • 1 day

[39] http://www.interactivebrokers.com/en/software/api/apiguide/java/reqhistoricaldata.htm

[40] The list of time zones accepted by IB is listed in §9.1 below

Parameter	Data type	Default	Description
WhatToShow	string (case insensitive)	'Trades'	Determines the nature of data being extracted. Valid values include: • Trades (default; 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 (default): 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' (the time part is omitted if barSize>=1d) 2) dates are returned as a long integer (# of seconds since 1/1/1970 GMT)
Timeout	number	Inf = unlimited	Maximal number of seconds to wait for an IB response to a request (note: the timeout is ignored in some cases, e.g. after partial data has been received etc.)

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 =
     datenum: [734909   734910   734913]
    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 was 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. This 1-day offset only occurs when UseRTH=0 (which is the default value): if you set UseRTH=1, then the correct dates will be reported, since regular trading hours end in the same day, not after midnight.

IB’s historical data mechanism enables retrieving data as recent as a minute ago, and as old as a year (or even older, if you purchase this option from IB). Some software vendors differentiate between intra-day and historical information, but as far as IB and **IB-Matlab** are concerned, this is merely a semantic difference and there is no actual difference. Subject to the available options in the Duration-vs.-BarSize table at the beginning of this section, we can select any date/time window that we wish.

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

7 Streaming data

Streaming data is a near-real-time mechanism, where IB sends ongoing information to **IB-Matlab** about quote ticks (bids and asks) and aggregated real-time bars.

7.1 Streaming quotes

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 latest 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 described in §5.1). To collect streaming quotes endlessly, simply set QuotesNumber to the value `inf`. Note that in Matlab, `inf` is a number not a string so do <u>not</u> enclose it in quotes (`'inf'`) when submitting requests.

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,[41] 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)
```

[41] QuotesBufferSize is a numeric parameter like QuotesNumber, so don't enclose the parameter value within string quotes ('')

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.[42]

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]
```

Streaming quotes are stored using a combination of the LocalSymbol, SecType, and Expiry date values that were specified in the initial request for the streaming quotes.

In most cases (as in the example above), we only need to specify the Symbol/LocalSymbol and the QuotesNumber in the subsequent requests.[43] Specifying all the other parameters is normally unnecessary, since **IB-Matlab** already knows about this symbol's parameters from the initial streaming request. We would only need to specify the SecType and possibly also the Expiry when there is a potential conflict between distinct streaming quotes (e.g., streaming quotes of both the underlying asset and some Futures index of it).

This is useful and easy to use, but also means that you cannot have two simultaneous streams for the same combination of LocalSymbol, SecType and Expiry, even if using different other 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

[42] 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).

[43] **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, for most stocks, Symbol = LocalSymbol so this distinction does not really matter.

- `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).
- `quotesReceived` – 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 and independently from 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=5000) 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.

Here is a summary of the *IBMatlab* parameters that directly affect streaming quotes:

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) – (default) • 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 in a cyclic buffer. Once this number of quotes has been received, the oldest quote is discarded whenever a new quote arrives.
GenericTickList	string	''	Used to request additional (non-default) information: volume, last trade info, etc.[44]
ReconnectEvery	integer	5000	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.
LocalSymbol	string	''	Used to identify and store streamed quotes.
SecType	string	'STK'	Used to identify and store streamed quotes.
Expiry	string	''	Used to identify and store streamed quotes.

Notes:

- IB does not send 'flat' ticks (quotes where price does not change). Also, IB streaming data is NOT tick-by-tick, but rather snapshots of the market (every 5ms for Forex, 10ms for Options, and 250ms for all other security types).
- By default, IB limits the streaming to 100 concurrent requests (contracts). Users can purchase additional 100-contract blocks from IB. The messages rate is not limited by IB, but a practical processing limit is ~100 quote events per second. This rate is ok for several dozen highly-traded stocks, but not the entire S&P, nor heavily-traded Forex.

[44] https://www.interactivebrokers.com/en/software/api/apiguide/tables/generic_tick_types.htm

7.2 Realtime bars

The realtime bars mechanism is similar to streaming quotes in the sense that it enables the user to receive information about a security every several seconds, until the specified QuotesNumber of bars have been received. The mechanism is also similar to historical data in the sense that the bars information is aggregated. Each bar contains the OHLCV information just as for historical data bars (see §6 for details).

Similarly to streaming quotes, the realtime bars mechanism has two distinct parts:

1. Request IB to start sending the stream of bars for a specified security. This is done by using Action='realtime_bars' and QuotesNumber with a positive (>0) value. If QuotesNumber=1 (the default value), then the data for the single bar is returned immediately; Otherwise, only the request ID is returned.
2. Later, whenever you wish to read the latest bar(s) data, simply use Action='realtime_bars' and QuotesNumber= -1 (minus one). This will return the latest information without stopping the background streaming.

Like streaming quotes, the streamed bars are stored based on a unique combination of their LocalSymbol, SecType and Expiry. As with streaming quotes, there is the ability to automatically reconnect to IB after every specified number of received bars.

Note that IB currently limits the realtime bars to 5-second bars only.[45] Also, only some combinations of securities, exchanges and data types (the WhatToShow parameter) are supported. If you have doubts about whether a specific combination is supported, ask IB customer service (the limitation is on the IB server, not *IBMatlab*).

Users can process realtime bars in one of two ways:

- use a Matlab timer to query the latest accumulated bars data (via QuotesNumber = -1), or:
- use the CallbackRealtimeBars parameter to set a dedicated Matlab callback function that will be invoked whenever a new bar is received (every 5 secs).[46]

Here is a simple example of using realtime bars for a single (snapshot) bar (QuotesNumber = 1), representing the previous 5 seconds:

```
>> data = IBMatlab('action','realtime', 'symbol','IBM')
data =
      atenum: 735551.017997685
    dateTime: {'13-Nov-2013 00:25:55'}
        open: 183
        high: 183
         low: 183
       close: 183
      volume: 0
         WAP: 183
       count: 0
```

[45] http://www.interactivebrokers.com/en/software/api/apiguide/java/reqrealtimebars.htm

[46] See §11 for details about setting up callback functions to IB events

And here is a slightly more complex example, with QuotesNumber=3. The data struct that is returned in this case is correspondingly more complex:

```
>> reqId = IBMatlab('action','realtime', 'symbol','AMZN', ...
                    'QuotesNumber',3, 'QuotesBufferSize',10);
reqId =
   345327051
```

```
(now wait 15 seconds or more for the 3 bars to be received)
```

```
>> dataStruct = IBMatlab('action','realtime', 'symbol','AMZN',
                         'QuotesNumber',-1);
dataStruct =
               reqId: 345327051
              symbol: 'AMZN'
         localSymbol: ''
            isActive: 0
      quotesReceived: 3
      quotesToReceive: 3
    quotesBufferSize: 10
          whatToShow: 'TRADES'
              useRTH: 0
                data: [1x1 struct]
            contract: [1x1 com.ib.client.Contract]
```

```
>> dataStruct.data
ans =
      atenum: [735551.008912037 735551.008969907 735551.009027778]
    dateTime: {1x3 cell}
        open: [349.97 349.97 349.97]
        high: [349.97 349.97 349.97]
         low: [349.97 349.97 349.97]
       close: [349.97 349.97 349.97]
      volume: [0 0 0]
         WAP: [349.97 349.97 349.97]
       count: [0 0 0]
```

```
>> dataStruct.data.dateTime
ans =
  '13-Nov-2013 00:12:50'  '13-Nov-2013 00:12:55'  '13-Nov-2013 00:13:00'
```

You may sometimes see warning messages of the following form:

```
[API.msg2] Can't find Eid with tickerId:345313582 {345313582, 300}
```

These messages can safely be ignored. They represent harmless requests by *IBMatlab* to IB, to cancel realtime bar requests that were already cancelled on the IB server.

Realtime bar requests are subject to both historical data pacing limitations (see §6 for details) and streaming data pacing limitations (§7.1). You may be able to loosen the limitations by purchasing additional data slots from IB. Discuss your alternatives with IB customer service, if you encounter pacing violation messages:

```
[API.msg2] Invalid Real-time Query: Historical data request pacing
violation {8314, 420}
```

Here is a summary of the *IBMatlab* parameters that directly affect realtime bars:

Parameter	Data type	Default	Description
Action	string	''	Needs to be 'realtime_bars' for this feature
QuotesNumber	integer	1	One of: • inf – continuous endless streaming bars for the specified security • N>1 – stream only N bars • 1 – get only a single bar (i.e., non-streaming snapshot) – (default) • 0 – stop streaming quotes • -1 – return latest accumulated bars data while continuing to stream data
QuotesBufferSize	integer	1	Controls the number of streaming bars stored in a cyclic buffer. Once this number of bars has been received, the oldest bar is discarded whenever a new bar arrives.
GenericTickList	string	''	Used to request additional (non-default) information: volume, last trade info, etc.[47]
LocalSymbol	string	''	Used to identify and store streamed bars.
SecType	string	'STK'	Used to identify and store streamed bars.
Expiry	string	''	Used to identify and store streamed bars.
WhatToShow	string (case insensitive)	'Trades'	Determines the nature of data being extracted. Valid values include: • 'Trades' (default) • 'Midpoint' • 'Bid' • 'Ask'
UseRTH	integer or logical flag	0 = false	Determines whether to return all data in the requested time span, or only data that falls within Regular Trading Hours: • 0 or false (default): all data is returned, even outside of regular trading hours • 1 or true: only data within the regular trading hours is returned, even if the requested time span falls outside RTH.
ReconnectEvery	integer	5000	Number of quotes (total of all securities) before automated reconnection to IB and re-subscription to the realtime bars. • inf – accept realtime bars without automated reconnection • N>0 – automatically reconnect and re-subscribe to realtime bars after N bars are received.

[47] https://www.interactivebrokers.com/en/software/api/apiguide/tables/generic_tick_types.htm

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'.[48]

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 datenum 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.[49] You need to carefully ensure that the order type is accepted by IB before using it in *IBMatlab*. Here is the list of order types supported by *IBMatlab*, which is a subset of the list in IB's documentation:[50]

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

[48] SSHORT is only used by non-cleared iBroker accounts: since their orders are not being cleared by IB, they need to distinguish between shorting a position or selling. This is typically relevant only for large institutional users. In most cases, your orders are being cleared by IB you would only specify SELL for both shorting and selling.

[49] http://www.interactivebrokers.com/en/index.php?f=4985

[50] http://www.interactivebrokers.com/en/software/api/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 (the AuxPrice parameter).
	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 (the AuxPrice parameter).
	Trailing Limit if Touched	TRAIL LIT	A Trailing Limit-If-Touched sell order sets a trigger price at a fixed amount (AuxPrice parameter) or % (TrailingPercent parameter) 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 market 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 (AuxPrice parameter) or % (TrailingPercent parameter) above the market price. If the market price falls, the trigger price falls by the same amount; if it rises, the trigger price remains unchanged. If the market 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 (AuxPrice parameter) or % (TrailingPercent parameter) below the market price. If the market price rises, stop trigger price rises by the same amount; if it falls, the trigger price remains unchanged. If market 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 (TrailStopPrice parameter) at a fixed amount (AuxPrice parameter) or % (TrailingPercent parameter) below 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*)

Class	Order type full name	Order type abbreviation	Description
Execution 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.
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 executes as a Market order during closing 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 automatically changes to Limit order if it does not execute immediately at market.
	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 amt.
	TWAP - best efforts	TWAP	Achieves the Time-Weighted Average Price on a best-effort basis (see details in §9.2 below).
	VWAP - best efforts	VWAP	Achieves the Volume-Weighted Average Price on a best-effort basis (see details in §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, expiration times using 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*:[51]

Parameter	Data type	Default	Description
Action	string	(none)	One of: 'Buy', 'Sell', or 'Sshort' (note the double-S in Sshort)[52]
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 and TRAIL)
TrailingPercent	number	0	Trailing percent for TRAIL order types
TrailStopPrice	number	0	The stop price used when Type='Trail Limit'
TIF	string	'GTC'	Time-in-Force. Not all TIFs are available for all orders. Can be one of: • 'Day' – Good until end of trading day • 'DTC' – Day Till Cancelled • 'GTC' – Good Till Cancelled (default) • 'GTD' – Good Till Date (uses the GoodTillDate parameter below) [53] • 'GAT' – Good After Time/date (uses GoodAfterTime parameter below) [54] • 'IOC' – Immediate or Cancel • 'FOK' – Fill or Kill[55] • 'OPG' – Open Price Guarantee[56] • 'AUC' – Auction, submitted at the Calculated Opening Price[57]

[51] Also see the corresponding API documentation: http://www.interactivebrokers.com/en/software/api/apiguide/java/order.htm

[52] SSHORT is only used by non-cleared iBroker accounts: since their orders are not being cleared by IB, they need to distinguish between shorting a position or selling. This is typically relevant only for large institutional users. In most cases, your orders are being cleared by IB you would only specify SELL for both shorting and selling.

[53] GTD requires enabling Advanced Time In Force Attributes in the Preferences page of TWS / IB Gateway (http://www.interactivebrokers.com/en/software/webtraderguide/webtrader.htm#webtrader/orders/advanced_time_in_force_attributes.htm)

[54] GAT requires enabling Advanced Time In Force Attributes in the Preferences page of TWS / IB Gateway (http://www.interactivebrokers.com/en/software/webtraderguide/webtrader.htm#webtrader/orders/advanced_time_in_force_attributes.htm). For additional information on GAT see http://www.interactivebrokers.com/en/software/tws/usersguidebook/ordertypes/good_after_time.htm

[55] FOK is not officially supported, according to IB's API documentation (http://www.interactivebrokers.com/en/software/api/apiguide/tables/supported_order_types.htm). FOK requires the entire order to be filled, as opposed to IOC that allows a partial fill. For additional information on FOK see http://www.interactivebrokers.com/en/software/tws/usersguidebook/ordertypes/fill_or_kill.htm

[56] OPG is not officially supported, according to IB's API documentation (http://www.interactivebrokers.com/en/software/api/apiguide/tables/supported_order_types.htm). An OPG is used with a Limit

Parameter	Data type	Default	Description
GoodTillDate	string	''	Format: 'YYYYMMDD hh:mm:ss TMZ' (TMZ is optional[58])
GoodAfterTime	string	''	Format: 'YYYYMMDD hh:mm:ss TMZ' (TMZ is optional[59])
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
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.
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.
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 §9.6 below for additional details.
TriggerMethod	integer	0	One of:[60] • 0=Default • 1=Double-Bid-Ask • 2=Last • 3=Double-Last • 4=Bid-Ask • 7=Last-or-Bid-Ask • 8=Mid-point

order to indicate a Limit-on-Open order, or with a Market order to indicate a Market-on-Open order (http://www.interactivebrokers.com/en/software/webtraderguide/webtrader/orders/creating_an_order.htm)

[57] AUC is not officially supported, according to IB's API documentation (http://www.interactivebrokers.com/en/software/api/apiguide/tables/supported_order_types.htm). For additional information on AUC see http://www.interactivebrokers.com/en/software/tws/usersguidebook/ordertypes/auction.htm

[58] The list of time zones accepted by IB is listed in §9.1 below

[59] The list of time zones accepted by IB is listed in §9.1 below

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

Parameter	Data type	Default	Description
BracketDelta	number	[]=empty	Price offset for stop-loss and take-profit bracket child orders (see §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 related BracketDelta parameter above, and §9.3 below for additional details.

ParentId	integer	0	Useful for setting child orders of a parent order: these orders are only active when their parent OrderId is active or gets triggered. This is normally used in bracket orders (see §9.3 below), but can also be used otherwise.
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 §10 below).

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, automated orders and combo 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.[61] *IBMatlab* enables specifying the algo strategy's properties, as follows:

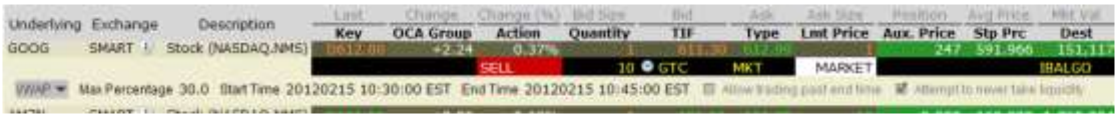
Parameter	Data type	Default	Description
MaxPctVol	number	0.1=10%	Percent of participation of average daily volume up to 0.5 (=50%).
StartTime	String	'9:00:00 EST'	Format: 'YYYYMMDD hh:mm:ss TMZ' (TMZ is optional)
EndTime	String	'16:00:00 EST'	(*same as StartTime above*)
AllowPastEndTime	integer or logical 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);
```

[61] http://www.interactivebrokers.com/en/trading/orders/vwapAlgo.php; http://www.interactivebrokers.com/en/software/tws/usersguidebook/algos/vwap.htm

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



```
Underlying  Exchange  Description           Key  OCA Group  Action  Quantity  TIF  Type  Lmt Price  Aux. Price  Stp Prc  Dest
GOOG        SMART     Stock (NASDAQ.NMS)                    SELL    10        GTC  MKT   MARKET                          IBALGO
VWAP  Max Percentage 30.0  Start Time 20120215 10:30:00 EST  End Time 20120215 10:45:00 EST  Allow trading past end time  Attempt to never take liquidity
```

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: [62]

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.

[62] http://www.interactivebrokers.com/en/software/api/apiguide/tables/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.[63] *IBMatlab* enables specifying the algo strategy’s properties, as follows:

Parameter	Data type	Default	Description
StrategyType	string	'Marketable'	One of: • 'Marketable' (default) • '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 §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.

[63] http://www.interactivebrokers.com/en/trading/orders/twapAlgo.php;
http://www.interactivebrokers.com/en/software/tws/usersguidebook/algos/twap.htm

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.[64] 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.

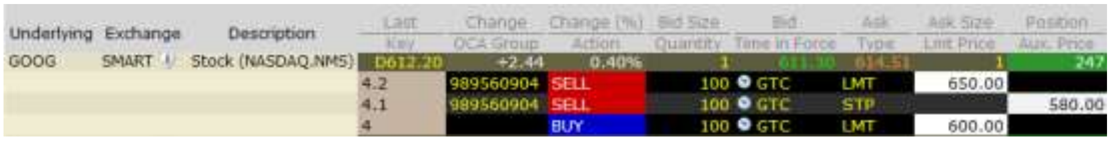
In IB-Matlab, brackets can only be assigned to parent Buy or Sell orders having Type=LMT or STPLMT. Specifying bracket orders is very simple, using the BracketDelta parameter. This parameter (default=[] = empty) accepts a single number value or an array of two numeric values, which specify the offset 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	Limit 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	

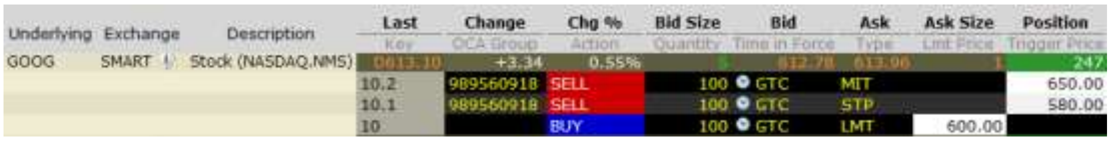
In this screenshot, notice that the parent order is shown as active (blue; IB status: "*Order is being held and monitored*") at the bottom. This order has a Last-Key value of "4" and is a simple Buy at Limit 600 order.

[64] 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 Key	Change OCA Group	Chg % Action	Bid Size Quantity	Bid Time in Force	Ask Type	Ask Size Lmt Price	Position 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 §10.2 below for details).

The parameters that specifically affect bracket orders include:

Parameter	Data type	Default	Description
BracketDelta	number	[]=empty	Price offset for stop-loss and take-profit bracket child orders (see §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 related BracketDelta parameter above, and §9.3 below for additional 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 for 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 useful.

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

Parameter	Data type	Default	Description
LimitBasis	string	(*none*)	Either 'BID' or 'ASK'. LimitBasis cannot be used together with LimitPrice.
LimitDelta	integer	0	Units of the security's minimal tick value
LimitBounds	[number, number]	[0,inf]	The LimitPrice will only fluctuate between the specified lower & upper bounds
LimitRepeatEvery	number	0	Update timer period in seconds
LimitPause	number	0	Update timer suspend time in seconds
LimitUpdateMode	number	0	Mode of the periodic LimitPrice update: • 0: LimitPrice increases or decreases based on the latest market bid/ask price • 1: LimitPrice can only **increase**; if market price decreases, LimitPrice remains as-is • -1: LimitPrice can only **decrease**; if the price increases, LimitPrice remains as-is
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 value in market queries (see §5.1).

IBMatlab uses Matlab timers for the implementation of automated orders having LimitRepeatEvery > 0. These timers invoke their callback function once every LimitRepeatEvery seconds. In each invocation, the current market data for the security is checked against the specifications (LimitUpdateMode, LimitBounds etc.). If it is determined that the trade order should be modified, then an update command is sent to the IB server with the new LimitPrice (see §10.2 below). This process could take some time and therefore it is strongly suggested to use a LimitRepeatEvery value larger than 5 or 10 [secs], otherwise Matlab might use a large percent of its CPU time in these timer callbacks. Each automated order uses an independent timer, so having multiple concurrent automated orders would only exasperate the situation. Therefore, the more such concurrent orders you have, the longer LimitRepeatEvery should be.

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 order Type requested by the user. LimitPrice cannot be used together with LimitBasis.

For example, the tick value for GOOG is 0.01. To send a Limit BUY order, which 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', 'LimitBasis','BID',...
                 'LimitDelta',-2, 'LimitRepeatEvery',15);
```

When trying to use the automated orders feature, you may discover that the limit price is not updated although the market price moves up or down. In most likelihood, this is due to the tick price not being available for some reason, and the simple solution is to specify it directly using the Tick parameter:

```
orderId=IBMatlab('action','BUY', 'symbol','GOOG', 'quantity',100,...
                 'type','LMT', 'LimitBasis','BID', 'tick',0.01,...
                 'LimitDelta',-2, 'LimitRepeatEvery',15);
```

The LimitPause parameter enables a suspension of the order for the specified duration between each timer invocation. At the beginning of each suspension, the order is cancelled. At the end of each suspension, the order is resubmitted with updated LimitPrice and Quantity (depending on the number of executed Quantity until that time). For example, if LimitRepeatEvery=15 and LimitPause=3, then the order will be active between t=0 and t=15, then again between t=18 and t=23, then again between t=26 and t=41, and so on.

High frequency traders often game REL and various types of pegged orders, e.g., by temporarily causing price to move up or down such that these orders trigger at less than optimal prices. Order delays reduce this possibility, as temporary price movements may revert before the order is re-released. The regular periodic update feature (LimitRepeatEvery) helps in this regard, but using LimitPause would increase the possibility of price improvement (e.g., for a buy order the price could drop below the original bid).

9.5 Combo orders

IB enables traders to trade a spread of securities as a single atomic combination (combo) order. For example, a trader might trade a calendar spread of some security options or futures, e.g. Sell November, Buy December. Each of these securities (*legs*) is treated separately, but the combination is treated as a single entity. Combo-orders typically improve trading certainty, reduce risk of partial or mis-executions, and reduce the trading costs significantly compared to trading the securities separately.

To use combo-trades in *IBMatlab*, specify the leg parameters (Symbol, LocalSymbol, SecType, Exchange, Currency, Expiry, Strike, and Right) in a cell array wherever the different legs have different values. In addition, you must specify the ComboActions parameter. You can also specify the ComboRatios parameter, but this is optional:[65]

```
orderId = IBMatlab('action','buy', 'exchange','CFE', 'quantity',1, ...
                   'SecType','FUT', 'LocalSymbol',{'VXZ2','VXX2'}, ...
                   'ComboActions',{'Buy','Sell'}, 'AccountName','D123')
```

Alternately, you could use cell arrays also for the fields that are the same for all legs. The following is equivalent to the command above:

```
orderId = IBMatlab('action','buy', 'exchange',{'CFE','CFE'}, ...
                   'quantity',1, 'SecType',{'FUT','FUT'}, ...
                   'LocalSymbol',{'VXZ2','VXX2'}, ...
                   'ComboActions',{'Buy','Sell'}, 'AccountName','D123')
```

The same syntax can be used for querying the market data of a specific combo:

```
data = IBMatlab('action','query','exchange','GLOBEX','secType','FUT',...
                'localSymbol',{'ESZ2','ESH3'}, ...
                'ComboActions',{'Sell','Buy'}
```

Note that querying market data for a combo might well return negative prices. For example, in the specific query example above, the following data was received:

```
data =
              reqId: 230455081
            reqTime: '26-Oct-2012 04:24:22'
           dataTime: '26-Oct-2012 04:24:23'
      dataTimestamp: 7.3517e+05
             ticker: ''
           bidPrice: -6.8500
           askPrice: -6.7500
            bidSize: 748
            askSize: 287
               open: -1
              close: -1
                low: -1
```

[65] By default, *IBMatlab* uses ratios of [1,1], i.e. the same ratio for all legs. You can specify any array of positive values that shall be used to determine the relative weights of the legs.

```
               high: -1
          lastPrice: -1
             volume: -1
               tick: 0.2500
           contract: [1x1 struct]
    contractDetails: [1x1 struct]
```

Note that only instantaneous market bid/ask data is returned – the open, close, low, high, volume and lastPrice information are not returned.

IB only supports combo trades for a small subset of securities (generally speaking, US options and futures). FOREX, for example, is currently NOT supported.

Unfortunately, IB does not report an informative error message when a combo trade order or market query is rejected. We are left guessing as to the reason: perhaps we specified one or more of the legs incorrectly; or maybe we are not subscribed for real-time data for any of the legs; or perhaps IB does not support the combo for the specific legs that we requested.

The combo legs must all use the same exchange and generally also the same currency. However, combo legs do not need to have the same underlying security Symbol. If you wish to use a combo spread of two securities with a different symbol, you could use the internal Symbol for the spread using the ComboBagSymbol parameter. For example, here's the 2-vs-10-year US Treasury-Notes spread:[66]

```
IBMatlab('action','query', 'SecType','FUT', 'exchange','ECBOT', ...
         'ComboBagSymbol','TUT', ...  % the spread's symbol is TUT
         'LocalSymbol',{'ZN    MAR 13','ZT    MAR 13'}, ...
         'ComboActions',{'Sell','Buy'}, ...
         'ComboRatios',[3,5])
```

Sometimes IB fails to return snapshot query data for combos (as for TUT above), due to IB server limitations/bugs. In such cases, using streaming quotes (see Chapter 7) may be a good workaround.

```
IBMatlab('action','query', 'SecType','FUT', 'exchange','ECBOT', ...
         'ComboBagSymbol','TUT', ...  % the spread's symbol is TUT
         'LocalSymbol',{'ZN    MAR 13','ZT    MAR 13'}, ...
         'ComboActions',{'Sell','Buy'}, ...
         'ComboRatios',[3,5], 'QuotesNumber',2);
% wait a bit for data to be received from IB server
data = IBMatlab('action','query', 'QuotesNumber',-1, ...
                'LocalSymbol',{'ZN    MAR 13','ZT    MAR 13'});
```

When specifying combo legs, we need to be aware of exchange limitations. In general combos use the default ratio of 1:1, but in some cases some other ratio is needed. For instance, in the above example, the ComboRatios for the ZT/ZN spread was set to 3:5

[66] A list of other similar predefined CME spreads can be found in http://cmegroup.com/trading/interest-rates/files/TreasurySwap_SpreadOverview.pdf and http://cmegroup.com/trading/interest-rates/intercommodity-spread.html

since the ECBOT exchange does not support any other ratio. Note that this ratio may change from time to time: the ZT/ZN spread ratio was 1:2 in early 2013, and changed to 3:5 later that year.[67] If you specify an incorrect ratio, or when the market is closed, IB will send an ICS (*Inter-Commodity Spread*) error message. For example:

```
[API.msg2] Invalid ICS spread {360280114, 318}
```

When specifying the spread's LocalSymbol, be careful to enter all the spaces in there. For example, the ZN LocalSymbol has 4 spaces between "ZN" and "MAR". IB is super sensitive about this and if you specify a LocalSymbol that is even slightly incorrect then IB will complain that it cannot find the specified contract.

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

Parameter	Data type	Default	Description
Symbol	string or cell-array of strings	(none)	The symbol(s) of the underlying leg assets.
LocalSymbol	string or cell-array of strings	''	The local exchange symbol of the underlying leg asset. When left empty, IB sometimes tries to infer it from Symbol and the other properties.
SecType	string or cell-array of strings	'STK'	One of: 'STK', 'OPT', 'FUT', 'IND', 'FOP' (but not 'CASH' or 'BAG') for the legs.
Exchange	string or cell-array of strings	'SMART'	The exchange that should process the request for the corresponding legs.
Currency	string or cell-array of strings	'USD'	The currency for the corresponding legs.
Expiry	string or cell-array of strings	''	'YYYYMM' or 'YYYYMMDD' format, for each of the combo legs.
Strike	number or numeric array	0.0	The strike price (for options) of the corresponding legs.
Right	string or cell-array of strings	''	One of: 'P', 'PUT', 'C', 'CALL' for each of the combo legs.
ComboActions	cell-array of strings	{}	Array of corresponding leg actions. For example: {'Sell', 'Buy'}
ComboRatios	numeric array of positive numbers	[1,1]	Array of corresponding leg weights. Any number is accepted – just the relative values matter, so [1,1.5]=[2,3]=[4,6].
ComboBag Symbol	string	''	The exchange symbol of the combo-bag spread. When left empty, *IBMatlab* will use the last leg's Symbol and LocalSymbol for the parent bag contract.

[67] The latest spread ratios on CME can be found here: http://cmegroup.com/trading/interest-rates/intercommodity-spread.html

9.6 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[68]), or to specify the All-or-None flag (via the order object's m_allOrNone property[69]).

These properties are not available as *IBMatlab* parameters, but they can still be specified in **IB-Matlab** using the `ibConnectionObject` Java object returned by *IBMatlab* as a second output value, as explained in §15 below. There are several ways in which we can create and update the contract:

- We can use `ibConnectionObject` to create the initial contract and order objects, modify their requested properties, then use `ibConnectionObject` again to send the order to IB. §15.3 shows a usage example of this.
- We can use *IBMatlab*'s Hold parameter (see §8) 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.

In all cases, we would use the `ibConnectionObject.placeOrder` function to send the updated contract and order to IB for processing. Here is a typical usage example:

```
% Prepare initial contract and order objects using the Hold mechanism
[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;
order.m_orderRef = 'abra kadabra';

% 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'), but not for other *IBMatlab* actions.

[68] http://www.interactivebrokers.com/en/software/api/apiguide/java/contract.htm

[69] http://www.interactivebrokers.com/en/software/api/apiguide/java/order.htm

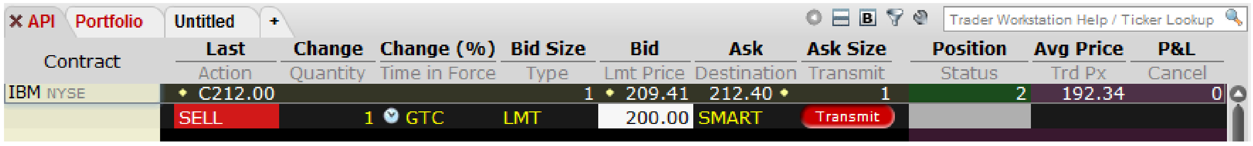
Some additional order fields that can be set in this manner include:

- `m_allOrNone` – true for an All-or-None order
- `m_sweepToFill` – true for a Sweep-to-Fill order
- `m_primaryExchange` – to clarify ambiguities with SMART-routed orders[70]
- `m_hidden` – true for a hidden order routed via the INet (Island) exchange[71]
- `m_displaySize` – >0 for an Iceberg order[72]
- `m_volatility` – >0 for specifying option limit price in terms of volatility [percent], typically used together with `m_volatilityType` [1=daily, 2=annual]
- additional contract and order fields are described in IB's API documentation[73]

The Hold mechanism can be used to programmatically prepare an order for later manual approval and transmission in TWS:

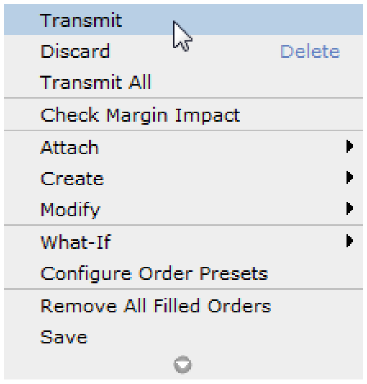
```
% Prepare initial contract and order objects using the Hold mechanism
[orderId, ibConnectionObject, contract, order] = ...
    IBMatlab('action','BUY', 'Hold',true, ...);

% Modify the order to NOT transmit immediately from TWS to IB server
order.m_transmit = false;

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

This will create the order in TWS without transmitting it. You will see the order in TWS's API tab with a button to transmit:



Contract	Last / Action	Change / Quantity	Change (%) / Time in Force	Bid Size / Type	Bid / Lmt Price	Ask / Destination	Ask Size / Transmit	Position / Status	Avg Price / Trd Px	P&L / Cancel
IBM NYSE	C212.00			1	209.41	212.40	1	2	192.34	0
	SELL	1	GTC	LMT	200.00	SMART	Transmit			

Right-clicking anywhere in the row will present a menu with additional options:



[70] As an alternative, you can specify the primary exchange directly in the Exchange parameter, for example: 'SMART:ISLAND'

[71] http://www.interactivebrokers.com/en/trading/orders/hidden.php

[72] http://www.interactivebrokers.com/en/trading/orders/iceberg.php

[73] https://www.interactivebrokers.com/en/software/api/apiguide/java/contract.htm, https://www.interactivebrokers.com/en/software/api/apiguide/java/order.htm, https://www.interactivebrokers.com/en/software/api/apiguide/tables/extended_order_attributes.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 §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:[74]

- `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 §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.

[74] http://www.interactivebrokers.com/en/software/api/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[75] 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 OrderId and/or Symbol. For example:

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

         (etc.)

% Filter by symbol: 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 =
     []
```

We can use the returned data to filter the results by any of the order/contract fields:

```
data = IBMatlab('action','query', 'type','open', 'symbol','goog');
for idx = length(data):-1:1   % only report orders having lmtPrice<600
    if data(idx).order.m_lmtPrice>=600, data(idx)=[]; end
end
```

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. If this limitation affects your work, use a static ClientID of 0, thereby enabling access to all open orders placed by any **IB-Matlab** session (since they will all have the same ClientID 0) as well as directly on TWS (which uses the same ClientID 0). See §13 for additional details on ClientID.

[75] http://www.interactivebrokers.com/en/software/api/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,ibConnectionObject]=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
```

Note: Orders placed manually via TWS all have an OrderId of 0, unless we run the following command in Matlab. TWS orders placed from then on will get unique IDs. Naturally, we also need to set ClientId=0 to access the TWS orders (see §10.1):

```
ibConnectionObject.reqAutoOpenOrders(true);  % see §15 for details
```

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 §10.1 above.

Alternately, we can use the Java connector object for this (see §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.[76]

Matlab has built-in support for asynchronous events, called *Callbacks* in Matlab jargon.[77] 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 <u>any</u> 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,[78] so IB's event name is misleading in this specific case).

[76] http://www.interactivebrokers.com/en/software/api/apiguide/java/java_eclientsocket_methods.htm

[77] http://www.mathworks.com/help/matlab/creating_guis/writing-code-for-callbacks.html

[78] http://www.interactivebrokers.com/en/software/api/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 §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:[79]

```
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.[80]

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','');
```

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

[79] http://www.mathworks.com/help/matlab/creating_guis/writing-code-for-callbacks.html#brqow8p

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

Using Matlab callbacks with the Java connector object can be used, for example, to implement combo trades,[81] as an alternative to the built-in mechanism described in §9.5 above. In this case, separate contracts are created for the separate combo legs, then submitted to IB via the Java connector's *reqContractDetails()* method, awaiting the returned IDs via the Matlab callback to the ContractDetails event (see **CallbackContractDetails** in the table below). Once the IDs for all the legs are received, `com.ib.client.ComboLeg` objects[82] are created. The completed order can then be submitted to IB for trading via the Java connector's *placeOrder()* method. All this may appear a bit difficult to implement, but in fact can be achieved in only a few dozen lines of code. This example illustrates how Matlab callbacks can seamlessly interact with the underlying Java connector's methods.

Here is the list of currently-supported callback events in *IBMatlab* (for additional information about any of the callbacks, follow the link in the "IB Event" column):

***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
CallbackAccountSummary	accountSummary	No	in response to calling *reqAccountSummary()* on the Java connector
CallbackAccountSummaryEnd	accountSummaryEnd	No	when all AccountSummary events have been sent, to indicate end of data
CallbackBondContractDetails	bondContractDetails	Yes	in response to market queries; not really used in *IBMatlab*
CallbackCommissionReport	commissionReport	Yes	immediately after a trade execution, or when requesting executions (see §12.1 below)
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

[81] http://www.interactivebrokers.com/en/software/api/apiguide/java/placing_a_combination_order.htm

[82] http://www.interactivebrokers.com/en/software/api/apiguide/java/comboleg.htm#XREF_interoperability_socket66

***IBMatlab* parameter**	**IB Event**	**Triggered by *IBMatlab*?**	**Called when?**
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 in response to calling *reqManagedAccts ()*on the Java connector
CallbackMarketDataType	marketDataType	No	when the market type is set to Frozen or RealTime, to announce the switch, or in response to calling *reqMarketDataType()* on the Java connector
CallbackMessage	error	Yes	whenever IB wishes to send the user an error or informational message. See §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
CallbackPosition	position	No	in response to calling *reqPositions()* on the Java connector
CallbackPositionEnd	positionEnd	No	After all Position events have been sent for a request, to indicate end of data
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)

***IBMatlab* parameter**	**IB Event**	**Triggered by *IBMatlab*?**	**Called when?**
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 of an option or its atenum moves. TWS's option model volatilities, prices, and deltas, along with the present value of dividends expected on that atenum 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	Yes	in response to a realtime bars request, for each of the result bars separately
CallbackReceiveFA	receiveFA	No	in response to calling *requestFA()* on the Java connector
CallbackScannerData	scannerData	Yes	in response to a user query for scanner data, for each result row
CallbackScannerDataEnd	scannerDataEnd	Yes	when the last scannerData event has been sent, to indicate end of data
CallbackScannerParameters	scannerParameters	Yes	in response to a user query for scanner parameters XML
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 §4) or updated
CallbackUpdateMktDepth	updateMktDepth	No	when 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 to track executions

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 §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*):[83]

```
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/en/software/api/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/en/software/api/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
```

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

11.3 Example – using CallbackTickGeneric to check if a security is shortable

In this example, we attach a user callback function to *tickGeneric* events in order to check whether a security is shortable[84] (also see §5.1 above).

Note: according to IB,[85] "*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:[86]

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

    % Only check the shortable tick type =46, according to
    % http://www.interactivebrokers.com/en/software/api/apiguide/tables/tick_types.htm

    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
```

[84] http://www.interactivebrokers.com/en/software/api/apiguide/tables/using_the_shortable_tick.htm

[85] http://www.interactivebrokers.com/en/software/api/apiguide/tables/generic_tick_types.htm

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

11.4 Example – using CallbackContractDetails to get a contract's full options chain

In this example, we attach a user callback function to *contractDetails* events in order to receive the full list of LocalSymbols and associated contract properties of an underlying security's options chain.

As noted in §3 above, it is not possible to receive the entire list of option *prices* in a single command – each market price requires a separate request with a specific LocalSymbol.

However, we can use the *contractDetails* event to extract the full list of option LocalSymbols in a single command. This relies on the fact that when the Right and Strike parameters of a contract are empty, IB returns the full list of contracts matching the other specifications.

We first define our callback function for the event:

```
function IBCallbackContractDetails(ibConnector, eventData)

    contract = eventData.contractDetails.m_summary;

    fprintf([char(contract.m_localSymbol) '\t' ...
             char(contract.m_secType)     '\t' ...
             char(contract.m_symbol)      '\t' ...
             char(contract.m_expiry)      '\t' ...
             char(contract.m_right)       '\t' ...
             char(contract.m_multiplier)  '\t' ...
             num2str(contract.m_strike)   '\n']);

end  % IBCallbackContractDetails
```

Now we ask IB for the current market data of the contract with empty Right and Strike. We can safely ignore the IB warning about ambiguous or missing security definition:

```
>> data=IBMatlab('action','query', 'symbol','CL', 'secType','FOP',...
                 'exchange','NYMEX', 'currency','USD', ...
                 'expiry','201306', 'right','', 'strike',0.0, ...
                 'CallbackContractDetails',@IBCallbackContractDetails)

[API.msg2] The contract description specified for CL is ambiguous;
you must specify the multiplier. {286356018, 200}

LOM3 P6650    FOP    CL     20130516     P      1000   66.5
LOM3 P8900    FOP    CL     20130516     P      1000   89
LOM3 P11150   FOP    CL     20130516     P      1000   111.5
LOM3 C6400    FOP    CL     20130516     C      1000   64
LOM3 C8650    FOP    CL     20130516     C      1000   86.5
LOM3 C10900   FOP    CL     20130516     C      1000   109
LOM3 C6650    FOP    CL     20130516     C      1000   66.5
LOM3 C8900    FOP    CL     20130516     C      1000   89
... (over 400 different contracts)
```

The returned `data` struct will naturally contain empty market data, but its contractDetails field will contain useful data about the requested security:

```
>> data
data =
              reqId: 286356019
            reqTime: '30-Apr-2013 12:55:28'
           dataTime: '30-Apr-2013 12:55:31'
      dataTimestamp: 735354.538562743
      lastEventTime: 735354.538562743
             ticker: 'CL'
           bidPrice: -1
           askPrice: -1
               open: -1
              close: -1
                low: -1
               high: -1
          lastPrice: -1
             volume: -1
               tick: 0.01
           contract: [1x1 struct]
    contractDetails: [1x1 struct]
```

```
>> data.contract  % these are the details for only one of the options
ans =
              m_conId: 50318947
             m_symbol: 'CL'
            m_secType: 'FOP'
             m_expiry: '20130516'
             m_strike: 111.5
              m_right: 'P'
         m_multiplier: '1000'
           m_exchange: 'NYMEX'
           m_currency: 'USD'
        m_localSymbol: 'LOM3 P11150'
                  ...
```

```
>> data.contractDetails
ans =
              m_summary: [1x1 com.ib.client.Contract]
           m_marketName: 'LO'
         m_tradingClass: 'LO'
              m_minTick: 0.01
       m_priceMagnifier: 1
           m_orderTypes: [1x205 char]
       m_validExchanges: 'NYMEX'
           m_underConId: 43635367
             m_longName: 'Light Sweet Crude Oil'
        m_contractMonth: '201306'
             m_industry: []
             m_category: []
          m_subcategory: []
           m_timeZoneId: 'EST'
         m_tradingHours: '20130430:1800-1715;20130501:1800-1715'
          m_liquidHours: '20130430:0000-1715,1800-2359;20130501:0000-
1715,1800-2359'
                    ...
```

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, §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:[87]

- `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

[87] http://www.interactivebrokers.com/en/software/api/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','executions','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','executions','symbol','goog')
```

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

```
>> data = IBMatlab('action','query','type','executions','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 §12.1 above, excluding the sub-structs `contract` and `execution`.

It should be noted that using these log files, which is done by default, can have a significant performance impact in cases of rapid partial executions. For example, if we buy 1000 shares of a security whose normal ask size is 5 shares, then we should expect about 200 separate execution messages when the order is filled. This in turns translates into 200 separate file saves, for each of the two log files (CSV, MAT). This could cause MATLAB to appear frozen for quite a long time until all this I/O is done.

To solve the performance issue in cases where the execution logs are not really needed, set the LogFileName parameter to the empty string ('') to prevent logging.

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. §11.2 above contains a working example of such a function.

As noted in §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 §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.[88]

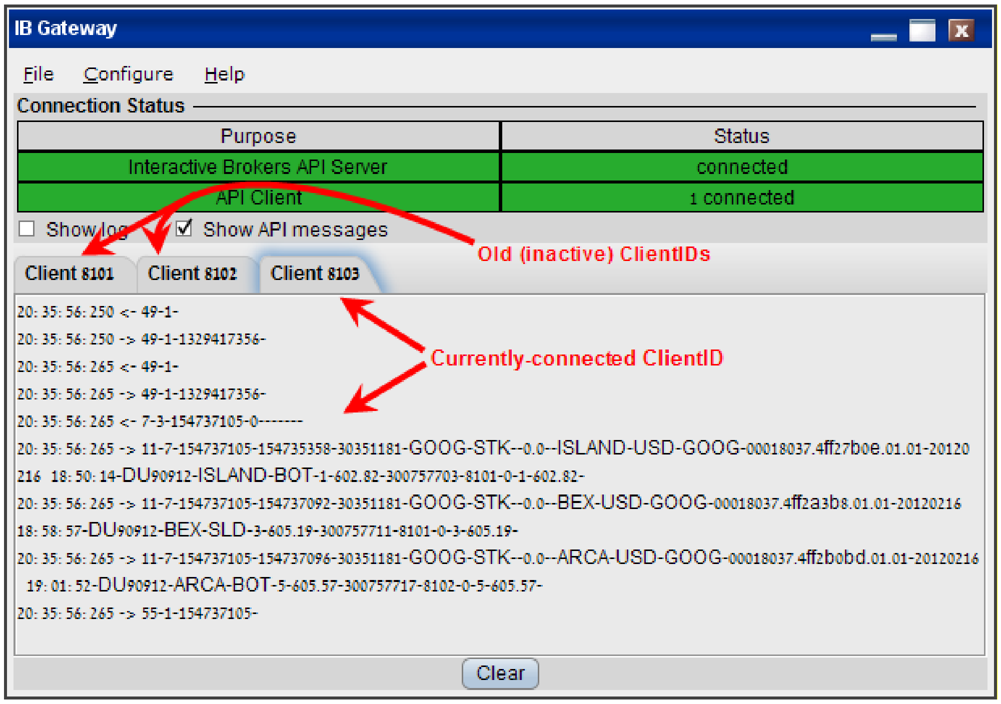
In reconnections of any kind, **IB-Matlab** automatically tries to reuse the same ClientID as in the previous connection.

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

[88] See the ReconnectEvery parameter (§7.1 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)
```



Purpose	Status
Interactive Brokers API Server	connected
API Client	1 connected

```
20: 35: 56: 250 <- 49-1-
20: 35: 56: 250 -> 49-1-1329417356-
20: 35: 56: 265 <- 49-1-
20: 35: 56: 265 -> 49-1-1329417356-
20: 35: 56: 265 <- 7-3-154737105-0-------
20: 35: 56: 265 -> 11-7-154737105-154735358-30351181-GOOG-STK--0.0--ISLAND-USD-GOOG-00018037.4ff27b0e.01.01-20120
216  18: 50: 14-DU90912-ISLAND-BOT-1-602.82-300757703-8101-0-1-602.82-
20: 35: 56: 265 -> 11-7-154737105-154737092-30351181-GOOG-STK--0.0--BEX-USD-GOOG-00018037.4ff2a3b8.01.01-20120216
18: 58: 57-DU90912-BEX-SLD-3-605.19-300757711-8101-0-3-605.19-
20: 35: 56: 265 -> 11-7-154737105-154737096-30351181-GOOG-STK--0.0--ARCA-USD-GOOG-00018037.4ff2b0bd.01.01-20120216
 19: 01: 52-DU90912-ARCA-BOT-5-605.57-300757717-8102-0-5-605.57-
20: 35: 56: 265 -> 55-1-154737105-
```

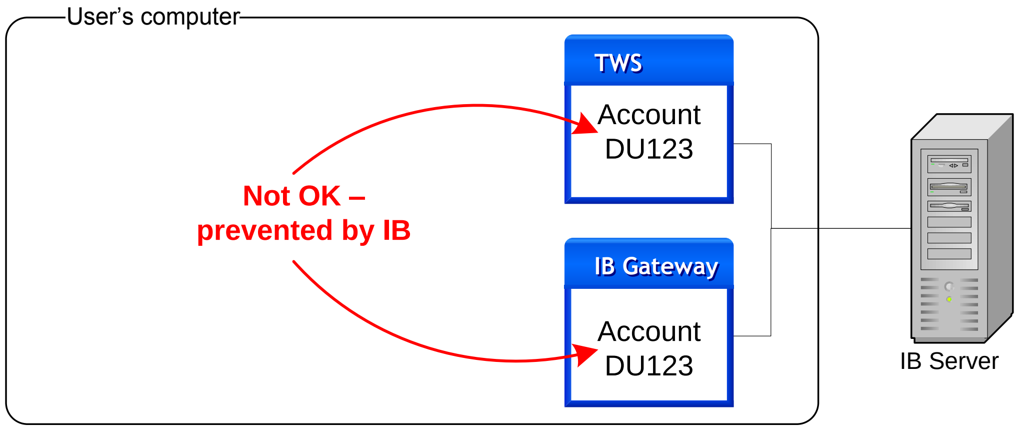
While specifying a new ClientID automatically disconnects and reconnects to IB, you can also force a disconnection/reconnection for the same ClientID, by using the Java connector object (discussed in §15 below). Following a disconnection from IB, **IB-Matlab** will automatically reconnect to IB upon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

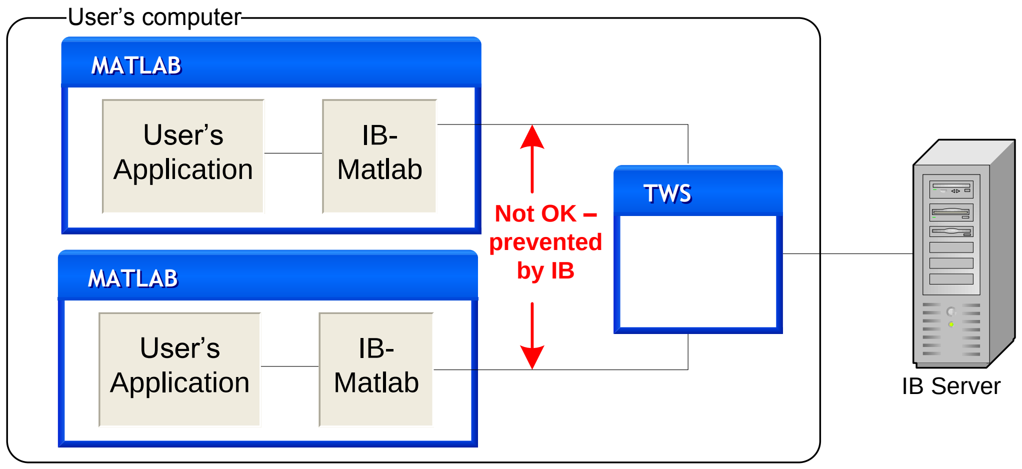
```
[data, ibConnectionObject] = IBMatlab('action',...);  % do whatever
ibConnectionObject.disconnectFromTWS;  % disconnect from IB
data = IBMatlab('action','portfolio');  % will automatically reconnect
```

ClientID 0 is special: it simulates the TWS and enables **IB-Matlab** to receive, modify and cancel 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 §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.

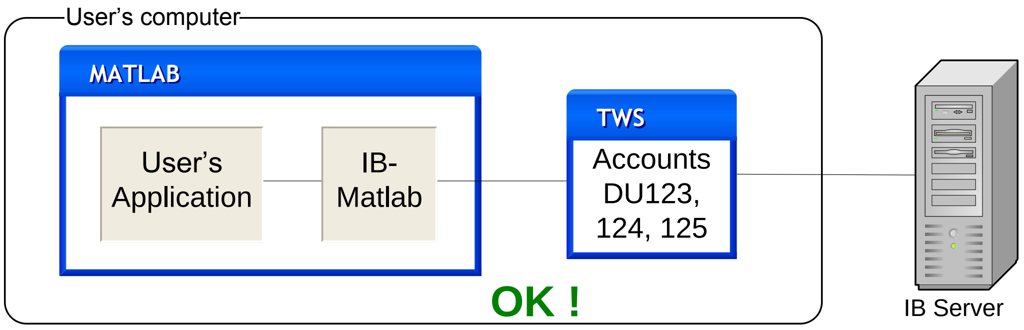
Another use is to connect **IB-Matlab** on one computer (which has Matlab installed) to TWS/Gateway on another computer, which may not necessarily have Matlab. In this case, simply set the Host and possibly also the Port parameters.

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.

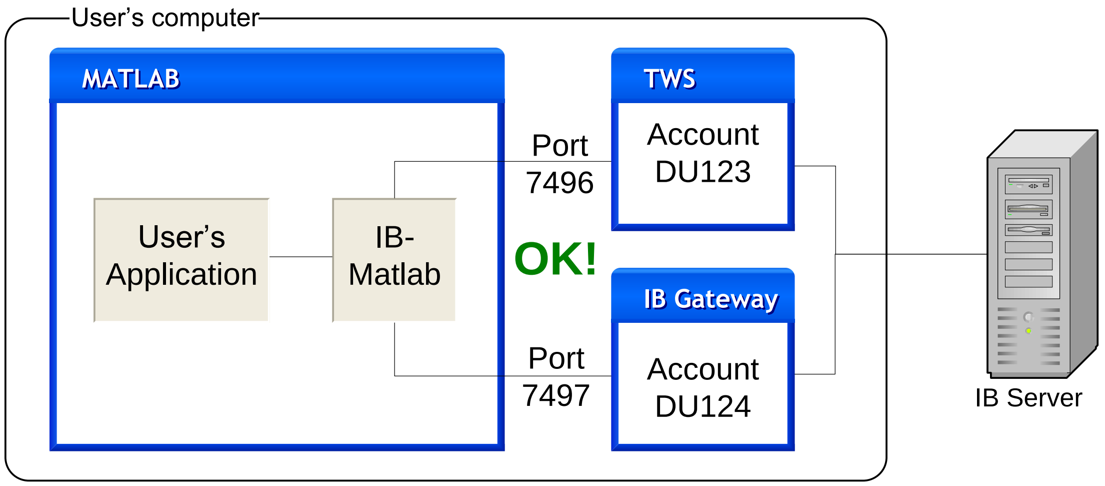




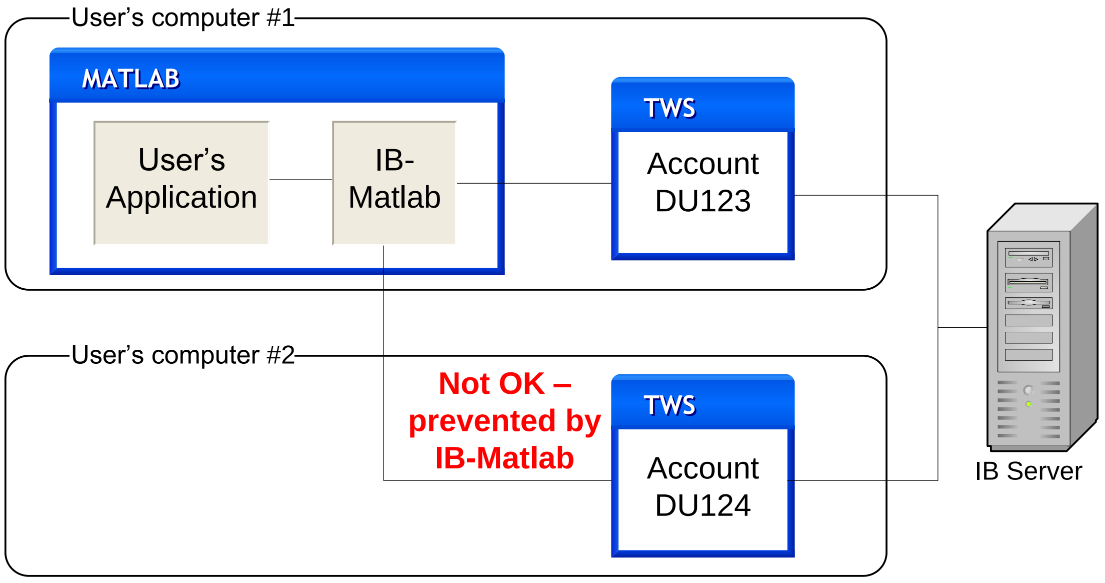
On the other hand, it is possible to control multiple IB accounts from the same TWS application, and in such a case **IB-Matlab** can access all of these accounts when it connects to TWS, using the AccountName parameter. Please refer to your TWS documentation (or IB's customer service) to set up your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



It is also possible to run TWS with one IB account, and IB Gateway with another account, either on the same computer as **IB-Matlab**, or on another machine. 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 both on the same Host.

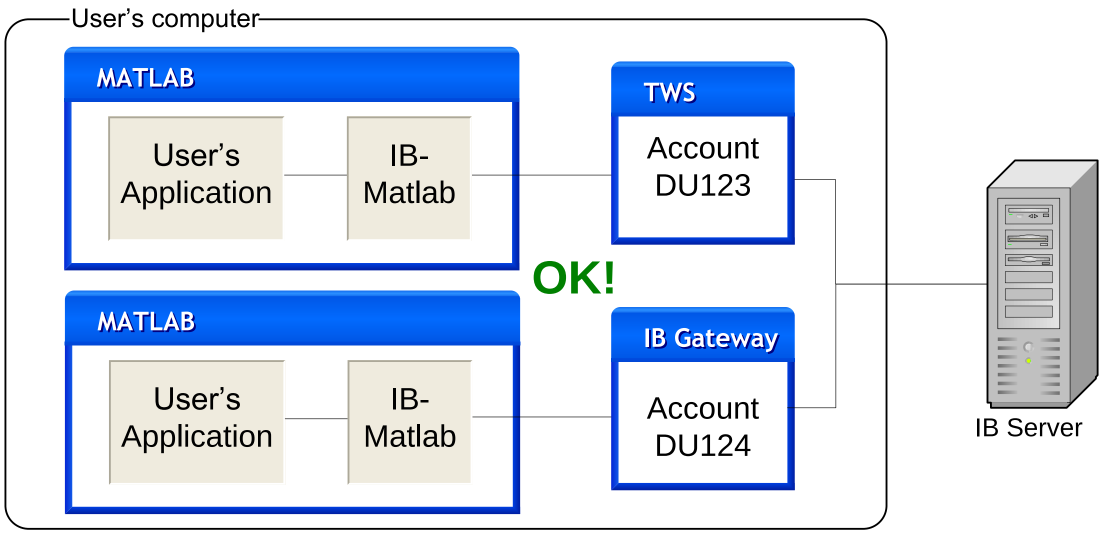


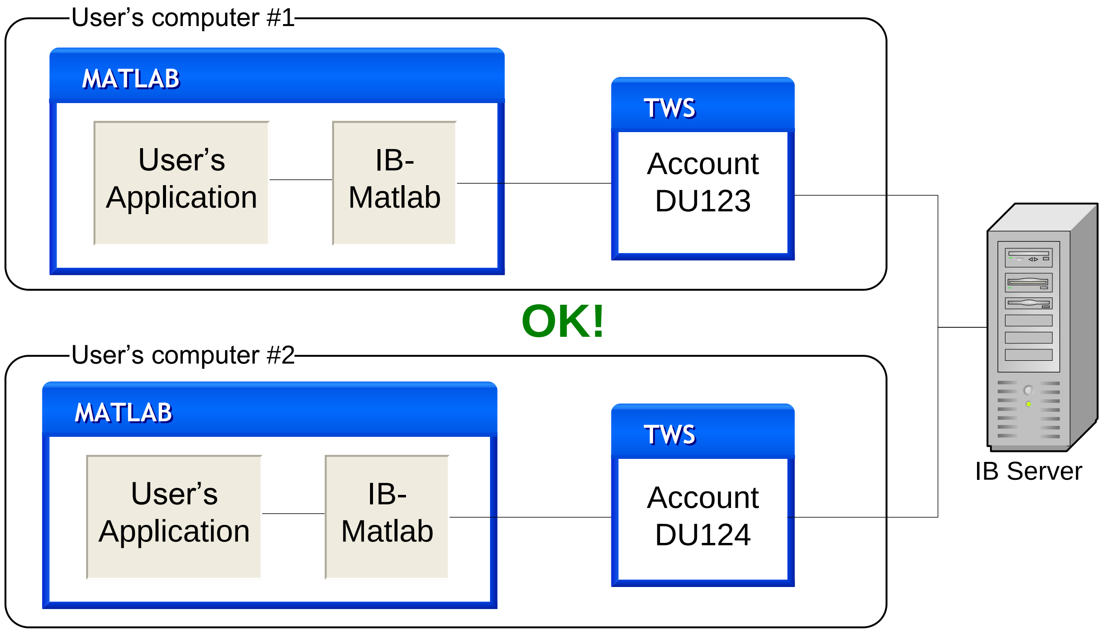
TWS & IB Gateway are on the same host (computer) and can be controlled by a single **IB-Matlab**



TWS & IB Gateway are on separate hosts (computers) and cannot be controlled by a single **IB-Matlab**

To control two or more TWS/Gateways, it is better to use distinct **IB-Matlab** instances, i.e., distinct Matlab sessions, each running its own **IB-Matlab** instance and connecting to a single IB client (TWS or Gateway). This helps prevent mix-ups in the AccountName or Port that might occur when **IB-Matlab** controls separate IB clients:





Note that if you wish to use **IB-Matlab** on separate computers, then you will need a separate **IB-Matlab** license for each computer.

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 §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.[89] 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 < 2
message, id (data1), code (data2)	errors and informational messages	[API.msg2]	MsgDisplayLevel < 1 or: data2 < 2000[90]
message, exception object	severe IB errors (exceptions)	[API.msg3]	MsgDisplayLevel < 2

Note: no IB message (regardless of data1, data2) is displayed if MsgDisplayLevel>=2

The full list of message codes (data2) for API.msg2 (which is the most common message type) is listed online.[91] It is sub-divided into three groups:

- Error messages (data2 codes between 100-999)
- System messages (data2 codes between 1000-1999)
- Warning messages (data2 codes between 2000-2999)

[89] http://www.interactivebrokers.com/en/software/api/apiguide/java/error.htm

[90] IB-Matlab versions prior to 2013-05-10 had a different implementation, in which MsgDisplayLevel=1 had an effect only on very few messages while most messages were displayed. This was fixed to be consistent with the original intention.

[91] http://www.interactivebrokers.com/en/software/api/apiguide/tables/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=200) happened for the specific request ID 153745227. We can therefore correlate between the error and the trade order 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.[92] Several mechanisms can help us with this detective work:

- We could set *IBMatlab*’s MsgDisplayLevel parameter to -1 or -2 (see above).
- We could set *IBMatlab*’s Debug 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.[93] 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).[94] 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,[95] 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.[96] 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”.

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.

[92] See examples in §3 above

[93] See §2, installation step 9d

[94] http://www.interactivebrokers.com/en/software/api/apiguide/tables/api_logging.htm

[95] 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/en/software/api/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/en/software/api/apiguide/api/api_logging.htm

[96] See the screenshot in §13 above

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).[97] Users of **IB-Matlab** should take extra precaution in their programs to ensure that these common mistakes do not occur.

A different type of error occurs in user applications that rely on valid data being returned from the IB server in response to account, portfolio or market-data queries. Unfortunately, the IB server data feed (or perhaps only the interface) is not as reliable as it could be. IB sometimes returns empty or invalid data field values, typically -1. This issue, together with some workarounds, was discussed in §3 and §4 above. In general, user applications should implement sanity checks on the returned data, and retry to send the request until receiving valid data. Failing to do so may result in applicative errors, and might even lead to bad automated trading decisions, so please be extra careful about this.

One final type of error may be due to out-of-memory errors, either directly in Matlab or in Java. Matlab "`out of memory`" errors might occur when receiving and storing a huge amount of streaming/historic data. They can be fixed by running IB-Matlab on a computer having more memory, or by reducing the amount of stored data.[98]

Java memory errors are recognized by the message "`java.lang.OutOfMemoryError: Java heap space`". They can be solved by running Matlab with more allocated Java heap memory than the default value of 64 or 128MB (depending on Matlab release). This value can be increased in Matlab's preferences, or via a *java.opts* file.[99]

[97] Both of these were discussed in §3 above

[98] Also see: http://www.mathworks.com/help/matlab/matlab_prog/resolving-out-of-memory-errors.html

[99] http://www.mathworks.com/support/solutions/en/data/1-18I2C/

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 `IBMatlab.IBConnection`. You can call its publicly-accessible methods (functions) just like any 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[100] and responses[101]). This was done on purpose, to enable easy integration with IB. `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, `currentTime()` is automatically being called by the underlying Java engine as a new event. However, as noted in §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.

Note: All trade (buy/sell/short) orders must be placed exclusively through either the `ibConnectionObject` interface (the *placeOrder()* method) or the *IBMatlab* name-value pair interface. Placing trade orders via both interfaces in a single **IB-Matlab** session will result in request ID mixup and failed trades: the IB server will reject trades that

[100] http://www.interactivebrokers.com/en/software/api/apiguide/java/java_eclientsocket_methods.htm

[101] http://www.interactivebrokers.com/en/software/api/apiguide/java/java_ewrapper_methods.htm

have duplicate or non-sequential IDs. Using duplicate and non-sequential IDs is not critical in many other IB requests, but is critical in the specific case of trade orders.

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 (may be called directly)
     **********************************/

    // Check if connected to TWS
    public  atenum 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()  // same as reqCurrentTime()

    // 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,
                            atenum 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,  atenum snapshotFlag)

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

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

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

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

    // Place an order
    public void placeOrder(int id, Contract contract, Order 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,  atenum outsideRTH)

    // 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,  atenum outsideRTH)

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

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

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

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

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

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

    // Request IB news bulletins
    public void reqNewsBulletins ( atenum allMsgsFlag)
```

```
// Cancel request for IB news bulletins
public void cancelNewsBulletins ()

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

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

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

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

// Request market scanner results
public void reqScannerSubscription (int tickerId,
                                    ScannerSubscription scannerSubscription)
// Cancel request for a scanner subscription
public void cancelScannerSubscription (int tickerId)

// Requests real-time bars (only barSize=5 is currently supported by IB)[102]
public void reqRealTimeBars(int tickerId, Contract contract, int barSize,
                            String whatToShow,  atenum useRTH)
// Cancel request for real-time bars
public void cancelRealTimeBars(int tickerId)

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

// Request 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)
// Cancel request for Reuters global fundamental data
public void cancelFundamentalData(int id)

// Request the next available reqId
public void reqNextValidId() // same as reqId() below
public void reqId() // a single ID
public void reqIds(int numIds) // multiple IDs

// Market Data requests:
// Calculate the implied volatility of a contract
public void calculateImpliedVolatility(int tickerId, Contract contract,
                                       double optionPrice, double  atenum p )
// Cancel request to calculate the implied volatility of a contract
public void cancelCalculateImpliedVolatility(int tickerId)

// Calculate an option price
public void calculateOptionPrice(int tickerId, Contract contract,
                                 double volatility, double  atenum p )
// Cancel request to calculate an option price
public void cancelCalculateOptionPrice(int tickerId)

// Cancel all open API orders - 9.65
public void reqGlobalCancel()

// Request market data type - 9.66
// (1 for real-time streaming market data or 2 for frozen market data)
public void reqMarketDataType(int marketDataType)

// Request reception of the data from the TWS Account Window Summary tab - 9.69
public void reqAccountSummary(int reqId, String group, String tags)
// Cancel request for TWS Account Window Summary data - 9.69
public void cancelAccountSummary(int reqId)
```

[102] http://www.interactivebrokers.com/en/software/api/apiguide/java/reqrealtimebars.htm

```
// Request reception of real-time position data for an all accounts - 9.69
public void reqPositions()
// Cancel request for real-time position data - 9.69
public void cancelPositions()

// Set 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( atenum flag)
// Get the Done flag
public  atenum isDone()


/****************************
 * IB Callbacks (invoked automatically - should NOT be called directly!)
 ****************************/

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

// Receives indication that the TWS connection has closed
public void connectionClosed()

// Receives 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,  atenum hasGaps)

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

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

```
// Receives 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  atenum p)

// 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()

    // Receives market data type information - 9.66
    public void marketDataType(int reqId, int marketDataType)

    // Receives commission report information - 9.67
    public void commissionReport(CommissionReport commissionReport)

    // Receives real-time position for an account - 9.69
    public void position(String account, Contract contract, int pos)

    // Indicates end of position messages - 9.69
    public void positionEnd()

    // Receives the data from the TWS Account Window Summary tab - 9.69
    public void accountSummary(int reqId, String account, String tag, String value,
                               String currency)

    // Indicates end of account-summary messages - 9.69
    public void accountSummaryEnd(int reqId)

}
```

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.[103]

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);
```

[103] http://www.interactivebrokers.com/en/trading/orders/arrivalprice.php, http://www.interactivebrokers.com/en/software/api/apiguide/tables/ibalgo_parameters.htm

And now for the corresponding Matlab code (notice how closely it resembles the original Java code):[104]

```
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 §9.6 above.

[104] 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 [105]

1. Download http://www.spatial-econometrics.com/ (jplv7.zip) and unzip into a new folder (this is a free alternative to Matlab's Econometrics Toolbox[106]).

2. Add the new toolbox function to your Matlab path. Naturally, use the actual folder name where you've unzipped the toolbox rather than my C:\... example:

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

3. Use IB-Matlab to get last year's daily history data for both securities:[107]

```
IBM_history  = IBMatlab('action','history', 'Symbol','IBM',   ...
                        'DurationValue',1,'DurationUnits','Y',...
                        'BarSize','1 day');

GOOG_history = IBMatlab('action','history', 'Symbol','GOOG',  ...
                        'DurationValue',1,'DurationUnits','Y',...
                        'BarSize','1 day');

% Transform row vectors => column vectors for the adf,cadf tests
IBM_close_prices  = IBM_history.close';   % array of 252 values
GOOG_close_prices = GOOG_history.close';  % array of 252 values
```

4. Ensure that both time series are not already stationary (do not have a unit root and tend to mean-revert). This is done with the ***adf*** (Augmented Dickey-Fuller test) function in the Spatial Econometrics Toolbox, which outputs a `results` struct that enables us to determine at varying confidence levels if this is in fact the case. ADF has the null hypothesis that the time series has a unit root and is non-stationary. To reject this, the absolute value of `results.adf` (the t-statistic) needs to be greater than the desired absolute value of `results.crit` – a vector that contains 6 confidence level values, corresponding to 99%, 95%, 90%, 10%, 5% and 1% confidence. Only if the null hypothesis of a unit root and non-stationarity **cannot be rejected** for **both** time series in a possible pair, will it be worth proceeding to the cointegration test in step 5.

```
adfResults = adf(IBM_close_prices,0,1);
if abs(adfResults.adf) < abs(adfResults.crit(3))
   % failed test - bail out
end
```

…and similarly test the second security (`GOOG_close_prices` here). Note that ***adf*** is a necessary but weak test: most securities will pass this test.

[105] In practice, many traders do this test much more rarely, for practical reasons. However, they risk a chance that the pair of securities have stopped being cointegrated, so trading based on their cointegration assumption could prove to be very costly…

[106] Also see MathWorks webinar "Cointegration & Pairs Trading with Econometrics Toolbox": http://www.mathworks.com/wbnr55450; More cointegration examples using MathWorks Econometrics Toolbox: http://www.mathworks.com/help/econ/identify-cointegration.html

[107] See §6 for details

5. The ***cadf*** (Cointegrating Augmented Dickey-Fuller test) function in the Spatial Econometrics Toolbox tests for cointegration between a dependent time series (y) and an explanatory time series (x).[108] The `results` struct output by ***cadf*** also includes a `results.adf` (t-statistic) value and six `results.crit` values. The null hypothesis here is that the time series are **not** cointegrated, so to reject this in favor of the pair possibly being cointegrated, the absolute value of `results.adf` needs to be greater than the desired `results.crit` value.

   ```
   cadfResults = cadf(IBM_close_prices, GOOG_close_prices, 0,1);
   if abs(cadfResults.adf) < abs(cadfResults.crit(3))
      % failed test - bail out
   end
   ```

   NOTES:
   There are numerous ways of calculating the beta of the relationship (and how frequently to adjust this) between the y and x time series (IBM and GOOG respectively in the example above) in order to determine the correct relative number of shares to buy/sell in each stock in a cointegrating pair.[109]

   Also, the above tests do not indicate which stock informationally leads the other and which is the dependent/independent variable. Tests such as Granger Causality are intended for this purpose.[110]

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

7. Compute the correlation factor β between the time series (e.g., using ***ols***):

   ```
   res = ols(IBM_close_prices, GOOG_close_prices);  % β == res.beta
   ```

8. Store STD = *std*(Close$_{sec1}$ – β*Close$_{sec2}$) for later use in the runtime part below

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

9. Use IB-Matlab to stream quote data for the two securities:[111]

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

10. Use IB-Matlab to attach our user-defined callback processing function:[112]

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

[108] In fact, ***cadf*** will test against a matrix of multiple explanatory variables but in the case of a pairs strategy only two vectors – the two historic time series of prices x and y – are required as inputs

[109] For more information on some of the possibilities see http://www.ljmu.ac.uk/Images_Everyone/Jozef_1st(1).pdf

[110] See http://maki.bme.ntu.edu.tw/codes/granger_tool/etc_granger.m

[111] See §7 for details

[112] See §11 for details

16.2 Runtime – process TickPrice streaming-quote events

```
% Callback function to process IB TickPrice events
function IBMatlab_CallbackTickPrice(ibConnector, eventData, modelData)

   persistent lastPrice1 lastPrice2

   % If this is an event about the BID field of the new tick
   if (eventData.field == 1)  % == com.ib.client.TickType.BID

      if (eventData.price > 0)  % disregard invalid values

         % Update the security's stored last price with the new info
         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 pair of symbols
            ignoreFlag = true;
         end

         % Check whether the monitored securities have diverged
         % from their steady-state prices in either direction
         if ~ignoreFlag && ~isempty(lastPrice1) && ~isempty(lastPrice2)

            % Compute the divergence from the steady-state model
            deltaPrice = lastPrice1 - modelData.beta*lastPrice2;
            meanSpread = 0; % see footnote[113]

            % If the securities diverged too much, start trading
            if deltePrice < meanSpread -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 > meanSpread + 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  % if any price has changed
      end  % if this is a valid price event
   end  % if this is the tick's BID-field event
end  % IBMatlab_CallbackTickPrice
```

[113] The simplistic code here assumes that the long-term mean spread between the securities is 0. In practice, the mean spread has to be calculated in run-time. One way of doing this is to use Bollinger Bands: simply check whether we are outside the second band for the overbought/oversold signal. Bollinger band functions are available in the Financial Toolbox (***bollinger*** function), or can be freely downloaded from: http://www.mathworks.co.uk/matlabcentral/fileexchange/10573-technical-analysis-tool

17 Frequently-asked questions (FAQ)

1. Can IB-Matlab be used with other brokers?

IB-Matlab only connects to Interactive Brokers. It can be adapted for other brokers, but some development is obviously required since other brokers have different APIs. Contact me by email and I'll see if I can help.

2. Does IB-Matlab impose limitations on historical data or streaming quotes?

IB-Matlab does not impose any limitations, but the IB server does impose limitations on the frequency of the requests and the amount of returned data.[114] The limitations depend on your specific IB subscription. The basic IB subscription allows 2000 historical data bars, once every 10 seconds, going back up to one year. If you request more bars then IB returns nothing, and if you request more frequently then IB returns a pacing violation error. Additional data, going back up to 4 years, can be requested from IB based on your trading volume and subscription level. Again, the limitations are imposed by the IB server based on your account; IB-Matlab supports whatever subscription your account has, and does not limit the information in any manner.

3. Can I see a demo of IB-Matlab?

You can see a webinar showing a demo of IB-Matlab (along with presentation slides and the demo's source code).[115] In addition, you are most welcome to request a fully-functional trial version of IB-Matlab, that you can use to run the demo yourself, or to test your own trading strategies.

4. How does IB-Matlab compare to alternative products?

IB-Matlab is currently the market leader in the niche of Matlab-to-IB integration. There are other alternatives available, but IB-Matlab provides by far the best functionality, value and cost-effectiveness. Consider the following points when you test IB-Matlab's free trial:

1. IB-Matlab works on Matlab 7.1 (R14 SP3) onward; other alternatives do not
2. IB-Matlab works on all Matlab platforms (Windows, Mac, Linux); other alternatives (that use ActiveX) only work on Windows
3. IB-Matlab provides access to 100% of the IB API; other alternatives only to a small subset. Some examples of important features that are unique to IB-Matlab: streaming quotes, scanner data, bracket orders, current portfolio, account data.
4. IB-Matlab provides numerous features and functionality configurations; other alternatives are much more limited. Some important features that are unique to IB-Matlab include connecting to a remote TWS, connecting to multiple IB accounts, multiple parameter configurations etc.

---

[114] https://www.interactivebrokers.com/en/software/api/apiguide/api/historical_data_limitations.htm

[115] http://undocumentedmatlab.com/ib-matlab/real-time-trading-system-demo

5. IB-Matlab provides an easy-to-use Matlab wrapper function with numerous settable parameters, in addition to providing direct access to the raw API connector object; other alternatives only provide the raw connector option, which is much more difficult to program and debug.
6. IB-Matlab provides detailed logging options (with different levels of outputs from very detailed to errors-only); other alternatives provide none.
7. IB-Matlab enables to connect to multiple IB client accounts (FA) from within a single Matlab script; other alternatives can only access a single account.
8. IB-Matlab can connect to TWS on the same computer or on remote computers; other alternatives only offer a local connection.
9. IB-Matlab uses a Java API connector that is superior, more stable, and faster than the ActiveX connector used by other alternatives. In addition, IB-Matlab was specifically optimized for best possible performance.
10. IB-Matlab's User Guide contains 100 full pages packed with details, examples, real-life advice and numerous online references, with additional content available online; other alternatives' documentation is much more limited and does not contain much detail or examples.
11. IB-Matlab has a track record of several years and hundreds of clients (ranging from self-traders to large banks). It's been a very long time since anyone reported a bug – it's rock solid. On the other hand, some other alternatives are either very recent products or have stopped being supported some years ago.
12. IB-Matlab is actively supported, maintained and improved, as evidenced by the online change-log;[116] other alternatives are not currently maintained and improved, certainly not at IB-Matlab's pace.
13. IB-Matlab received numerous positive testimonials by active traders on IB's marketplace,[117] where IB-Matlab is one of the top-rated products. Other alternatives do not even have a single positive review.
14. Two highly-positive reviews of IB-Matlab were published in the Automated Trader magazine (2011,[118] 2012;[119] downloadable PDF versions: 2011,[120] 2012[121]); other alternatives received no such reviews.
15. IB-Matlab costs significantly less than other alternatives, for both the initial license and the annual maintenance. You could directly connect to IB's API connector for free, but this would require hundreds of development work-

[116] http://undocumentedmatlab.com/files/IB-Matlab_User_Guide.pdf#Page=98 (see Appendix B in this document)

[117] https://gdcdyn.interactivebrokers.com/Universal/servlet/MarketPlace.MarketPlaceServlet?action=softwareTools&selectedClient=tradersInvestors

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

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

[120] http://undocumentedmatlab.com/files/IB-Matlab_Review.pdf

[121] http://undocumentedmatlab.com/files/IB-Matlab_Review2.pdf

hours in order to get corresponding functionality.

5. How do you know that IB-Matlab trades $100M daily?

A few of the IB-Matlab users have chosen to tell me how they use the product, since they were very proud of how it enabled them to scale-up their trading. I have no way of verifying this information, because IB-Matlab does not send any information except to IB. The actual number from all IB-Matlab users may well be much higher than $100M.

6. Does IB-Matlab send you any information?

No – IB-Matlab only communicates with IB. The only communication that is done with this server is a verification of the computer's IB-Matlab activation (a single hash-code).

7. How can I be sure IB-Matlab does not contain bugs that will affect my trades?

Well, there is never a 100% guarantee. I rigorously test the product. IB-Matlab has been live since 2010 and is actively used by hundreds of users on a daily basis. So far nothing major has been reported. IB-Matlab is a very stable and robust product, despite the fact that new functionality is being added on a constant basis.

8. Is IB-Matlab being maintained? supported?

Yes, actively. Features and improvements are added on a regular basis, and I support the users personally. You can see the list of ongoing improvements in IB-Matlab's change-log, listed in Appendix B of the IB-Matlab User Guide (this document). You can see the very latest updates in the online version of this guide.[122]

9. I saw in the online User Guide that IB-Matlab added a nice new feature – can I get it?

You get the very latest version of IB-Matlab, including all the latest additions and improvements, when you purchase a new license or renew an existing one. If you do not wish to wait for the end of your license year, you can always renew immediately for $99. However, the new license year will start from that moment onward (in other words, you will lose the unused portion of your current annual license). For example, if you purchased the license on 1/1/2014 then it is good until 1/1/2015. If you choose to renew and get the latest IB-Matlab version on 9/9/2014, then your new license will expire on 9/9/2015, so you gain the latest IB-Matlab version but you lose about 4 months of your current annual license (worth about ~$30). The choice is yours.

10. What happens when the licensed year is over?

When you purchase the IB-Matlab license (or renewal), it will work for a full year. A short time before the year is over, you will start seeing a notification message in your

[122] http://undocumentedmatlab.com/files/IB-Matlab_User_Guide.pdf

Matlab console (Command Window) alerting you about this. This message will only appear during the initial connection to IB, so it will not affect your regular trading session. When the year is over, IB-Matlab will stop working. You can renew the license for additional years for only $99 per year. If you wish to be independent of such annual renewals, you can purchase multi-year renewals in advance, for a reduced cost.

11. I purchased a new computer – can I transfer my IB-Matlab license?

Yes: you will need to first deactivate IB-Matlab on your existing computer (I will walk you through this procedure) and then activate it on the new computer. In other words, don't format your old disk before you deactivate IB-Matlab, otherwise you will not be able to activate the new computer… You can make up to 3 such switches at no extra cost; additional switches incur a small handling fee.

12. I have a laptop and desktop, and I use them interchangeably – can I use IB-Matlab on both?

Yes, but you will need to purchase two separate IB-Matlab licenses. IB-Matlab's license is tied to a specific computer.

13. Can IB-Matlab be compiled and deployed?

Yes, IB-Matlab can indeed be compiled. However, it will require purchasing a separate license for each deployed computer other than the development computer. You do not need a separate license for the compiled application on your dev computer, since this is already licensed.

14. Is IB-Matlab provided in source-code format?

IB-Matlab is provided in encrypted binary form, like any other commercial software. If you wish to get the source-code, then this is possible, subject to signing a separate agreement and a higher cost. The benefit is that the source-code version has no license fees and is not tied to any specific computer – you can install it on as many computers as you wish <u>within your organization</u>. Contact me for details.

15. Do you provide an escrow service for IB-Matlab's source-code?

Yes. There are two alternative levels of escrow that you can select:

1. At safe-keeping with a Wall-Street lawyer
2. Using NCC Group's[123] independent escrow service

Escrow services incur a non-negligible usage fee, but you may decide that it may be worth it for ensuring business continuity. The choice is entirely yours.

[123] http://nccgroup.com/en/our-services/software-escrow-and-verification/software-escrow

If you wish to ensure business continuity, consider purchasing multi-year renewals in advance, for a reduced cost. This will ensure that your license will be independent of annual renewals for as many years as you select.

16. Is feature ABC available in IB-Matlab?

IB-Matlab supports the entire IB API. This means that all the functionality that IB exposes in its API, is available in IB-Matlab. In most cases, this functionality is available using an easy-to-use Matlab wrapper function. This includes all the important trading and query functionalities. Some additional functionalities, which are less commonly used, are supported by a more capable underlying connector object that IB-Matlab provides. To check whether a specific feature is available in the IB API (and by extension, in IB-Matlab), please refer to IB-Matlab's User Guide (this document), IB's online reference,[124] or contact IB customer service.

17. Can you add feature ABC for me?

I will be happy to do so, for a reasonable development fee that will be agreed with you in advance. After the development, this feature will be available to all others who purchase (or renew) the latest version of IB-Matlab, at no extra cost. Contact me by email if you have such a request, to get a proposed quote.

18. Can you develop trading strategy XYZ for me?

I will be happy to do so, for a reasonable development fee that will be agreed with you in advance. Unlike development of IB-Matlab features, strategy development will never be disclosed to others, and will not be integrated in IB-Matlab. They will be developed privately for you, and will be kept secret. Contact me by email if you have such a request, to get a proposed quote.

19. Does IB-Matlab include a back-testing engine?

No. IB-Matlab is only used for communication with the IB server (data from IB server; trade orders to IB server), it does not include any data analysis or back-testing functionalities. This is what makes the integration with Matlab so powerful, since Matlab is great at data analysis and visualization. So you can easily develop your own analysis programs in Matlab, that will get the data from IB-Matlab, analyze it, and send corresponding orders back to the IB server (again through IB-Matlab). If you'd rather use an off-the-shelf back-testing engine, then I could recommend TA Developer's backtesting application[125] or Walk-Forward Analysis Toolbox for MATLAB[126] – both of these work in Matlab and can easily be connected to the data from IB-Matlab.

[124] http://www.interactivebrokers.com/en/software/ibapi.php

[125] http://www.tadeveloper.com/docs/80-matlab-algo-trading

[126] https://secure.payproglobal.com/r.ashx?s=51029&a=52198&u=http://wfatoolbox.com

20. Does IB-Matlab work with the IB demo account?

Yes. However, note that IB's demo account is limited in data, functionality and performance compared to a live or paper-trading account. Therefore, it is best to trial IB-Matlab and to test your strategies using your paper-trading account (which you automatically get with your live account).

Appendix A – resources

A.1 Official IB resources

- API download page – http://www.interactivebrokers.com/en/software/ibapi.php
- All IB API guides –
  http://individuals.interactivebrokers.com/en/software/api_guides.php
- API Online Reference Guide –
  http://individuals.interactivebrokers.com/en/software/api/api.htm
- API Online Reference Guide – Java –
  http://www.interactivebrokers.com/en/software/api/apiguide/java/java.htm
- Java API Quick Reference (PDF) –
  http://www.interactivebrokers.com/download/inst/JavaAPIQuickReference.pdf
- Full API Reference Guide (PDF) –
  http://www.interactivebrokers.com/download/newMark/PDFs/APIprintable.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
- IB Gateway User's Guide (PDF) –
  http://individuals.interactivebrokers.com/download/newMark/PDFs/gateway.pdf
- Java API Samples for the Getting Started Guide (ZIP) –
  http://www.interactivebrokers.com/download/JavaAPIExamples.zip
- API Webinars -
  http://www.interactivebrokers.com/en/general/education/webinars.php?p=a&ib_entity=llc
- IB discussion forum – http://www.interactivebrokers.com/smf/
- API chat-room – http://chatsrv1.interactivebrokers.com/fchat/chat/flashchat.php
- API customer service and technical support – api@interactivebrokers.com
  (please let them know that you are using IB-Matlab, which in turn uses IB's Java API)
- Yahoo!'s IB discussion forum (not an official IB resource) –
  http://finance.groups.yahoo.com/group/twsapi/

A.2 MathWorks webinars

- Real-time trading in MATLAB – http://www.youtube.com/watch?v=e4bdKcFl4GA
  This is a recording of a presentation that I gave at the MATLAB Computational Finance Virtual Conference in September 2013, explaining IB-Matlab and showing how it can be used for real-time trading using a demo trading application.
  Also available: presentation slides and the demo application source-code.

Real-time trading in MATLAB

Yair Altman

Undocumented Matlab.com

altmany@gmail.com

- MathWorks algorithmic-trading portal – http://www.mathworks.com/discovery/algorithmic-trading.html, http://www.mathworks.com/financial-services/algorithmic-trading.html (includes the webinar "*Real-Time Trading System in MATLAB*" that features IB-Matlab)

- Algorithmic trading with MATLAB for financial applications – http://www.mathworks.com/wbnr52491

- Cointegration and Pairs Trading with the Econometrics Toolbox – http://www.mathworks.com/wbnr55450

- Energy trading & risk management with MATLAB – http://www.mathworks.com/wbnr50145

- Automated Trading with MATLAB – http://www.mathworks.com/wbnr68672

- Commodities Trading with MATLAB – http://www.mathworks.com/wbnr78374

A.3 Additional open-source Matlab resources

- Quantitative Finance Matlab Code Library – http://www.quantcentral.com/quantitative-finance-matlab-code-library/
  (click one of the "Matlab Code" options in the website's main toolbar)

- Spatial Econometrics Toolbox for Matlab – http://www.spatial-econometrics.com/

- Algorithmic trading code in the Matlab File Exchange – http://www.mathworks.com/matlabcentral/fileexchange/?term=trading

Appendix B – change log

The following table lists changes done to this document. Depending on the date that you have installed **IB-Matlab**, your version may be missing some features discussed in this document. Whenever you renew your annual license, you will receive the latest **IB-Matlab** version, including all the functionality detailed here.

Version	Date	Section(s)	Description
1.09	2012-04-16		Baseline version for this change-log
1.10	2012-05-05	4	Some account-data fields have several variants starting with this IB-Matlab version
		15.1	Clarification about mixing Matlab and Java trade orders
1.11	2012-05-27	16	Major overhaul of the Pairs Trading sample model
		A.2	Added this new section with references to online MathWorks webinars
1.12	2012-05-30	Cover	Added note about compatibility with the IB API version 9.67
		15.2	Added Java method interfaces *commissionReport(), marketDataType(), reqMarketDataType()*
1.13	2012-06-15	3	Added currency usage example (RUT→'CAD'); added explanation how to retrieve contract info in TWS via Contract Info / Description menu
		5	Clarified what happens if any of the pre-conditions for querying market data are not met
		11.1	Clarified to follow the hyperlinks in the table
		12.1	Fixed usage example code: 'open' → 'executions'
		14	Typo fix: "code=200**0**" → "code=200**)**"
1.14	2012-06-19	4	Clarified about requesting portfolio information (multi-accounts, safe programming, shorts, sums)
		8	Added references to the **AuxPrice** parameter where relevant; added TWAP order type; rearranged the parameters in the order-params table; clarified **TIF** ='GTD' usage; clarified **ParentID** usage
1.15	2012-06-28	9.4	Clarified that **LimitBasis** cannot be used with **LimitPrice**; fixed the code snippet accordingly
		14	Added discussion data errors leading to -1 values
		16.1	Added reference link to the MathWorks Econometrics Toolbox
1.16	2012-07-18	11.1	Added usage example of the Java connector object for implementing combo-trades
		14	Added discussion of out-of-memory errors
1.17	2012-08-07	9.3	Fixed LMTSTP → STPLMT

Version	Date	Section(s)	Description
1.18	2012-09-13	Disclaimer	Changed paragraph ordering (no change in text)
		3	Expanded the explanation how to retrieve contract info in TWS via Contract Info / Description menu
		4	Added the contract field in the portfolio-query results struct; fix in code snippet (sleep→pause)
		5	Clarified that if any of the pre-conditions for querying market data are not met, the returned info can be empty or error
		9.5	Clarified on using the ibConnectionObject
		A.1	Clarified about telling IB support that IB-Matlab uses the Java API when contacting them
1.19	2012-09-25	5	Clarified that all time-stamp fields except lastTimestamp reflect the local (not server) time.
1.20	2012-09-27	11.1, 14, 16.1	Updated some reference links following changes on the www.mathworks.com website
1.21	2012-10-03	2	Clarified on setting the path in ***classpath.txt***
1.22	2012-10-06	3	Clarified on reusing the struct input format
		4	Split §4 into §4.1 (account information) and §4.2 (portfolio information)
		A.2	Added resource: added reference to Automated Trading online webinar
1.23	2012-10-25	Cover	Updated IB API compatibility: 9.67 → 9.68
		4.1	Clarified that on a multi-account, not specifying the **AccountName** parameter can lead to stale or empty account data
		4.2	The returned contract field in the portfolio query results struct is now a Matlab struct, not a Java object; clarification that **AccountName** ='all' can be used on a multi-account
		5	Added the contractDetails field in the market-query results struct
		9.6 (now: 9.5)	New section and functionality (**ComboActions**, **ComboRatios** params) for combo-orders
1.24	2012-11-19	2	Clarified (item 9e) about upgrades in Windows 32-bit to 64-bit
		9.6 (now: 9.5)	Switched usage example from ECBOT to GLOBEX exchange
1.25	2012-11-27	9.5, 9.6	Switched the relative order of these sections
		11.1	Clarified that the Java connector object can be used to implement combo trades as an alternative to the built-in mechanism (§9.5)

Version	Date	Section(s)	Description
1.26	2012-12-22	9.5	Added **ComboBagSymbol** parameter; added usage example of **ComboBagSymbol**; clarifications regarding combo legs limitations (need to use same exchange/currency, default ratio)
1.27	2013-01-10	15.2	Replaced Boolean→boolean in Java method interfaces; added methods *reqNextValidId()*, *reqId()*, *reqIds()*, *calculateImpliedVolatility()*, *cancelCalculateImpliedVolatility()*, *calculateOptionPrice()*, *cancelCalculateOptionPrice()*
1.28	2013-02-08	-	(formatting changes only; no change in text)
1.29	2013-02-10	A.2, A.3	Added a few online resources
1.30	2013-03-08	2	Clarified editing the *classpath.txt* file
1.31	2013-04-11	8	Fixed description for **Hold** parameter (refer to §9.6, not §9.5)
		9.6	Added usage example for preparing an order for manual approval/transmission
		B	Added this change log in a new Appendix B
1.32	2013-04-19	Cover	Updated IB API compatibility: 9.68 → 9.69
		1, A.1	Updated IB links, following changes on IB website
		15.2	Added new methods supported by 9.69: *reqAccountSummary()*, *cancelAccountSummary()*, *reqPositions()*, *cancelPositions()*, *accountSummary()*, *accountSummaryEnd()*, *position()*, *positionEnd()*
1.33	2013-04-30	3	Clarified usage of Symbol and SecType options
		8	Added additional supported TIF options
		11	Added new sub-section §11.4; updated section titles
1.34	2013-05-10	7	Clarified streaming quotes limitations
		14	Clarified behavior of MsgDisplayLevel=1
		A.1	Added references for IB API chat-room and Yahoo! Forum; updated IB's forum web-address
1.35	2013-06-21	2, 3	Clarified that IB's paper-trade account is the simulated trading account
		6	Added the datenum field in the historical-data query results struct; clarified that the time part is omitted in the dateTime field when barSize>=1d
		12.2	Clarified setting LogFileName='', to prevent real-time execution logging, for improved performance
1.36	2013-07-15	5, 9.5, 11.4	Addition of contract field to market query results
		6	Updated historical data limitations imposed by IB

Version	Date	Section(s)	Description
1.37	2013-09-11	11.1	Added callbacks for IB's *AccountSummary, AccountSummaryEnd, CommissionReport, MarketDataType, Position* and *PositionEnd* events.
		15.2	Clarified the function interface of `reqRealTimeBars()`
		A.2	Added MathWorks' Commodities Trading webinar
		(all)	Fixed online IB references (the IB server modified the URLs of its online API documentation)
1.38	2013-09-27	9.4	Added the **LimitUpdateMode** parameter; Added clarification about the **Tick** value
		A.2	Added Real-time trading in MATLAB webinar/ presentation (+demo application source code)
1.39	2013-10-10	6	Added the **Timeout** parameter
		9.4	Added clarification about **LimitRepeatEvery** value
		9.5	Added clarifications about **ComboRatios** volatility and **LocalSymbol** sensitivity
1.40	2013-10-23	Disclaimer	Clarified the Disclaimer text e.g., clarified that any mention of a trading symbol or trading order does not constitute a recommendation by the author etc.
		1	Replaced outline image with a clearer version
		3	Clarified that an IB client (TWS or Gateway) needs to be active in order for **IB-Matlab** to work; Added: **IB-Matlab** will automatically start TWS if an IB client (TWS or IB Gateway) is not active
		8	Fixed reference URL to IB's order-types webpage
		A.2	Added screenshot of my Real-time Trading webinar; Added reference to MathWorks Algorithmic Trading portal that features a webinar on using **IB-Matlab**
1.41	2013-11-19	7.1	Added clarification about using **GenericTickTypes** parameter to get extra info in streaming quotes
		7.2	Added this section on the new realtime bars feature
		11.1	Indicated that the *realtimeBars* event is triggered by **IB-Matlab**, accessible via **CallbackRealtimeBars**
		11.4	Clarified that the returned data structure only contains contract info for a single option contract
		13	Clarified the options of multi-client connection(s); Minor clarification on use of **ClientID** parameter
		A.3	Added reference to Spatial Econometrics Toolbox
		B	Clarified that you will get the latest **IB-Matlab** version whenever you renew your annual license

Version	Date	Section(s)	Description
1.42	2013-11-27	5.2	Added this section on the new scanner data feature
		7.1	Added minor clarification about not enclosing numeric parameters in Matlab string quotes ("); Clarified that streaming quote fields are independent of each other
		11.1	Indicated that the *scanner** events are triggered by **IB-Matlab**, in response to user scanner requests
1.43	2014-01-02	8	Clarified that SSHORT order action is typically used only by some institutional traders; most users should only use BUY and SELL order actions.
		9.4	Added the **LimitPause** parameter for automated orders
		9.5	Clarified that IB sends an error message when the combo ratio is incorrect
1.44	2014-01-25	6	Clarified the effect of **UseRTH** on the reported historical data dates; clarified that the historical data mechanism is also used for retrieving intra-day data.
		7.1	Modified streaming quotes **ReconnectEvery** default value (2000→5000)
		10.1	Added usage example of filtering open orders by their limit price
1.45	2014-02-03	5.1	Added automatic re-fetch attempt upon missing data
		7.2	Added the serverTime field to realtime-bars data
		9.6	Added usage example of setting the orderRef field
1.46	2014-03-03	7.2	Added the **ReconnectEvery** feature to realtime bars
1.47	2014-03-14	-	Added the ability for **IB-Matlab** to work with VPN
		9.4	Added the **LimitBounds** parameter for automated orders
		9.5	Clarified the table explanation of the **ComboRatios** parameter
1.48	2014-05-26	2.1	Added this new section on licensing alternatives
		2.2	Added this new section on switching activated computers
		5.2	Clarified importance of some scanner parameters
		10.2	Clarified how to get unique OrderIds for manual orders placed within TWS
1.49	2014-08-05	9.6	Explained using order fields for Iceberg, hidden, all-or-none, sweep-to-fill and other special order types
		14	Clarified that no IB message is displayed if **MsgDisplayLevel** >= 2
1.50	2014-09-05	17	Added this new FAQ section to this document

Version	Date	Section(s)	Description
1.51	2014-10-01	5.1	Clarified possible reasons for receiving partial query data when querying current market data.
		9.5	Clarified usage of CME inter-commodity spreads
		3, 9.6	Clarified that the primary exchange can be specified in the **Exchange** parameter (e.g., 'SMART:NYSE')
1.52	2014-10-06	8	Added the **TrailingPercent** parameter for use by various TRAIL order types. Clarified the use of **AuxPrice** parameter for trailing amounts in TRAIL.