IBC (Interactive Brokers Connector)
User Guide

Version 2.04

July 17, 2021

Fully compatible with:

Windows, Linux, Mac OS

IB TWS versions 962 - 977

MATLAB R2006a - R2020a

https://Matlab-trader.com

Table of Contents

DISCLAIMER

THIS IBC SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND/OR NONINFRINGEMENT.

THIS SOFTWARE IS NOT OFFICIALLY APPROVED OR ENDORSED BY ANY REGULATORY, GOVERNING OR COMMERCIAL BODY, INCLUDING SEC, FINRA, MATHWORKS AND/OR INTERACTIVE BROKERS.

MUCH EFFORT WAS INVESTED TO ENSURE THE CORRECTNESS, ACCURACY AND USEFULNESS OF THE INFORMATION PRESENTED IN THIS DOCUMENT AND THE SOFTWARE. HOWEVER, THERE IS NEITHER A GUARANTEE THAT THE INFORMATION IS COMPLETE OR ERROR-FREE, NOR THAT IT MEETS THE USER’S NEEDS. THE AUTHOR AND COPYRIGHT HOLDERS TAKE ABSOLUTELY NO RESPONSIBILITY FOR POSSIBLE CONSEQUENCES DUE TO THIS DOCUMENT OR USE OF THE SOFTWARE.

THE FUNCTIONALITY OF THE SOFTWARE DEPENDS, IN PART, ON THE FUNCTIONALITY OF OTHER SOFTWARE, HARDWARE, SYSTEMS AND SERVICES BEYOND OUR CONTROL. SUCH EXTERNAL COMPONENTS MAY CHANGE OR STOP TO FUNCTION AT ANY TIME, WITHOUT PRIOR NOTICE AND WITHOUT OUR CONTROL. THEREFORE, THERE CAN BE NO ASSURANCE THAT THE SOFTWARE WOULD WORK, AS EXPECTED OR AT ALL, AT ANY GIVEN TIME.

IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES, LOSS, OR OTHER LIABILITY, WHETHER IN ACTION OF CONTRACT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE, REGARDLESS OF FORM OF CLAIM OR WHETHER THE AUTHORS WERE ADVISED OF SUCH LIABILITIES.

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

ALL TRADING SYMBOLS AND TRADING ORDERS DISPLAYED IN THE DOCUMENTATION ARE FOR ILLUSTRATIVE PURPOSES ONLY AND ARE NOT INTENDED TO PORTRAY A TRADING RECOMMENDATION.

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 trade 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.

IBC uses IB's Java API to connect Matlab to IB, providing a seamless interface within Matlab to IB's Java API functionality. Users can activate IB's API using simple Matlab commands, without any need to know Java.

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

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

[2] http://interactivebrokers.github.io/tws-api/initial_setup.html

[3] https://www.interactivebrokers.com/en/index.php?f=5041

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

1. A Java package (IBC*.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 (IBC*.p*) that provides the most often-used API functionality in an easy-to-use Matlab function. This wrapper is a pure Matlab implementation that provides access IB's most important functionalities, without needing to know anything about Java or the underlying connector.

Active trading actions (buy, sell, short, close, modify, cancel, exercise, lapse) 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 (IBC*.p*). Additional trading features (the full range of IB's Java API) can be accessed using the connector object that is provided by **IBC**.

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 executes, or a specified price is reached, for example.

This document explains how to install and use **IBC**. Depending on the date that you have installed **IBC**, your version may be missing some features discussed in this document. Whenever you renew your annual license you will receive the latest **IBC** version, including all the functionality detailed here.

2 Installation and licensing

IBC requires the following in order to run:

1. An active account at IB

   IBC will also work with IB's Demo account, but this is not recommended: it is limited in comparison with the functionalities of a live account. To properly test **IBC**, we recommend using the paper-trade (simulated trading) account that you get when you open an IB account. Paper-trade accounts resemble a 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 R2006a or a newer release

   If you have an earlier release of Matlab, some API functionality may still be available on your system. Contact us support@Matlab-Trader.com for details.

The installation procedure for **IBC** is as follows:

1. Ensure that you have read and accepted the IBC license agreement. This step is required even for **IBC** trials. If you do not accept the license agreement, you cannot use **IBC**.

2. Extract the files within the installation zip file (esp. IBC*.jar, IBC.p, IBC.m*) into a dedicated folder (for example: *C:\IBC\*). Do not place the files in one of Matlab's installation folders.

3. Add the new **IBC** folder to your Matlab path using the path tool (in the Matlab Desktop's toolstrip, click HOME / ENVIRONMENT / Set path… and save).[9] The folder needs to be in your Matlab path whenever you run *IBC*.

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

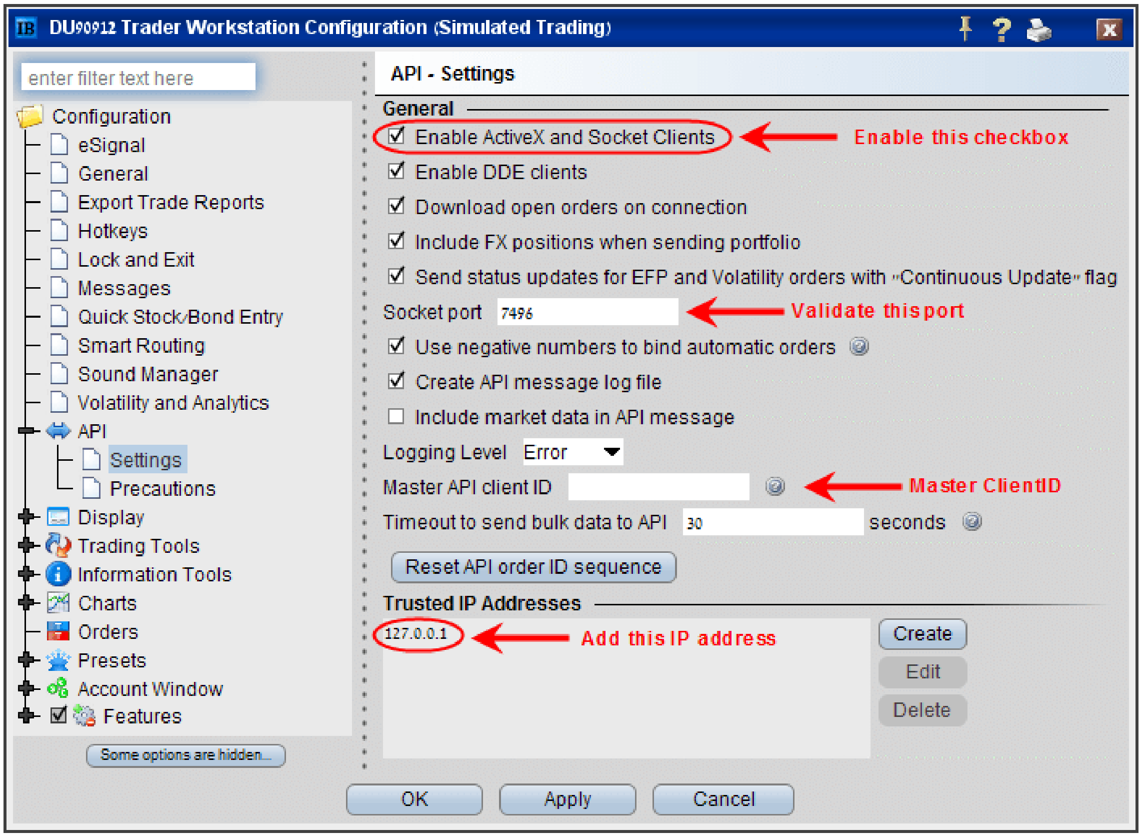
5. In TWS, go to the main menu's Edit / Global Configuration... / API / Settings and make the following changes (the specifics depend on your TWS version):

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

   b. Validate the Socket port used by IB for API communication: normally 7496 for TWS, 4001 for Gateway. Other values can be set, but then you will need to specify the **Port** parameter when you use *IBC* for the first time after each Matlab restart (see §13 below for details).

c. Decide whether to specify a Master ClientID (positive integer number) that will provide access to all orders created by any API client. You can then use this in *IBC* by specifying the **ClientId** parameter.

d. 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 but not TWS) 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.



API - Settings

General

☑ Enable ActiveX and Socket Clients — Enable this checkbox

☑ 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 — Validate this port

☑ 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 — Master ClientID

Timeout to send bulk data to API 30 seconds

Trusted IP Addresses

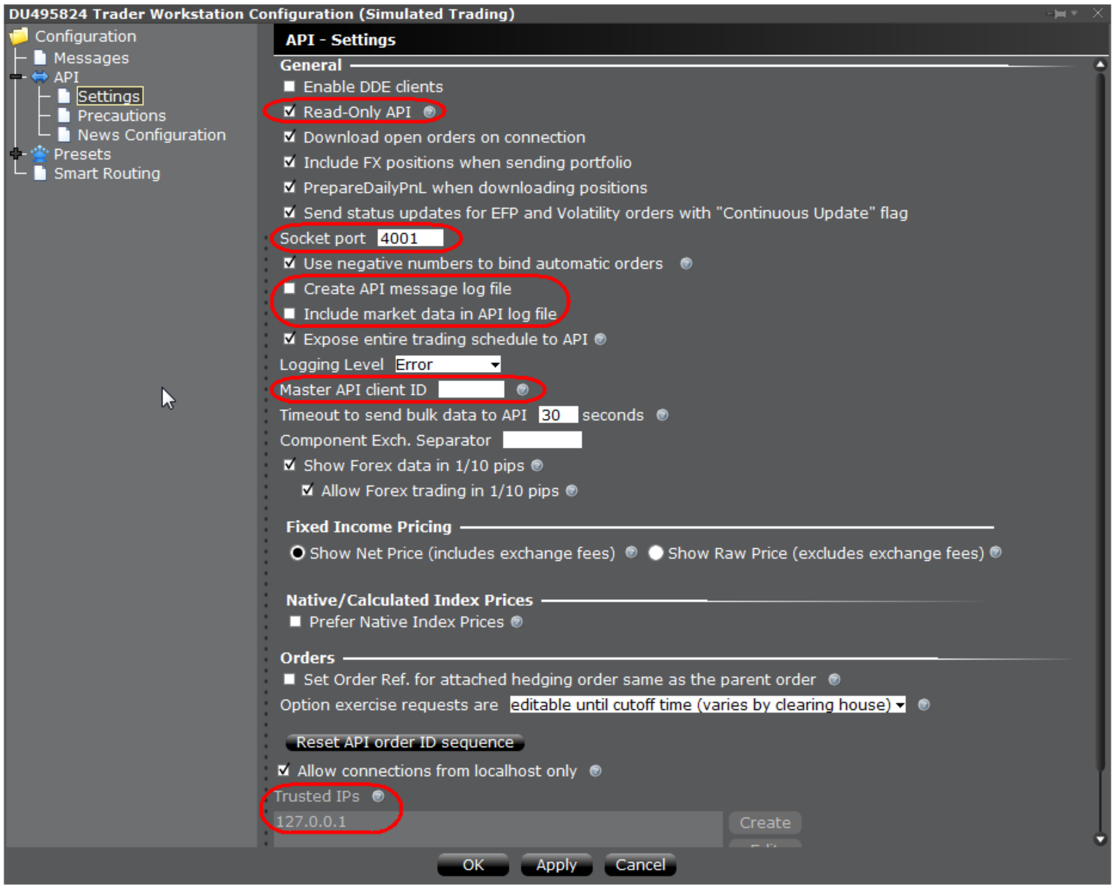
127.0.0.1 — Add this IP address

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 **IBC**, this could be due to a 32-64 bit mixup in TWS.

f. Unset the "Read-Only API" checkbox if you wish IBC to be able to send trade orders to IB (rather than just access IB's data-feed).

6. 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 main 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).[13]



DU495824 Trader Workstation Configuration (Simulated Trading)

API - Settings

General

- ☐ Enable DDE clients
- ☑ Read-Only API
- ☑ Download open orders on connection
- ☑ Include FX positions when sending portfolio
- ☑ PrepareDailyPnL when downloading positions
- ☑ Send status updates for EFP and Volatility orders with "Continuous Update" flag
- Socket port 4001
- ☑ Use negative numbers to bind automatic orders
- ☐ Create API message log file
- ☐ Include market data in API log file
- ☑ Expose entire trading schedule to API
- Logging Level Error
- Master API client ID
- Timeout to send bulk data to API 30 seconds
- Component Exch. Separator
- ☑ Show Forex data in 1/10 pips
  - ☑ Allow Forex trading in 1/10 pips

Fixed Income Pricing

○ Show Net Price (includes exchange fees) ● Show Raw Price (excludes exchange fees)

Native/Calculated Index Prices

- ☐ Prefer Native Index Prices

Orders

- ☐ Set Order Ref. for attached hedging order same as the parent order
- Option exercise requests are editable until cutoff time (varies by clearing house)

Reset API order ID sequence

- ☑ Allow connections from localhost only

Trusted IPs

127.0.0.1

7. If you are running the non-trial version of **IBC**, you will need to activate your license at this point, by sending us the output of the `IBC('license')` command. See §2.1 below for licensing details.

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

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

9. If you get an error such as "IBC *is not activated on this computer. To fix this, send the following data* ...", email us this message as plain text (not a screenshot) so we could activate your license on this computer, then retry step #8.

10. If you get an error such as "IBC*.jar not found in static Java classpath. Please add IBC.jar to classpath.txt*", then follow the steps in §2.3 below carefully, restart Matlab and retry step #8. If you still get the error, contact us.

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

2.1 Licensing and activation

IBC'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.

Once the computer's license is activated, the activation key is stored on the authorization webserver. This activation key validates online whenever **IBC** connects to IB (i.e., at the beginning of an IB/TWS session), and once every few hours while connected. Validating the license online only takes a second or two. Since it is only done at the initial connection to TWS and once every few hours, it does not affect **IBC**'s run-time performance. If you have a special concern regarding the online activation, please contact us for clarifications.

A corollary of the computer fingerprint is that whenever you modify any component that affects the fingerprint, **IBC** 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 software that creates dynamic MACs. In such cases, you might see an error message such as the following when you try to use **IBC**:

```
Error using IBC/ensureConnected
IBC is not activated on this computer.
```

Some additional information may be presented to help you diagnose the problem.

To fix such cases, simply revert back to the original hardware/networking configuration, and then **IBC** 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).

Computer fingerprints are typically stable, and are not supposed to change dynamically. However, some software programs (especially on MacOS, but also sometimes on Linux/Windows) create dynamic MAC addresses and/or dynamically modify the computer name (hostname). This is then reflected in the OS-reported fingerprint, possibly causing **IBC** to stop working. The solution in such cases is to find a way to keep the MAC addresses and computer name static, with the same values as the activated fingerprint. The hostname can be set using the OS's *hostname* command, and you can determine the nature of the OS-reported MACs as follows:

```
>> IBC('license', 'debug',1);  % partial sample output below
84A6C8EEAFED - net5 Intel(R) Centrino(R) Wireless-N 2230
B888E3E1EDD4 - eth4 Realtek PCIe GBE Family Controller
```

Using this output, you can determine which MAC address was changed / added / deleted, and then take the appropriate action to fix it so that the reported MACs will match the activated fingerprint. If you decide that the MACs/hostname changes are permanent, contact us to change the activated fingerprint (see §2.2 below).

The standard **IBC** license is for a single year from date of purchase. Additional licensing options are available; contact us at support@Matlab-Trader.com for pricing information:

- **Multi-year** license: 2-year, 3-year, or 5-year extended license terms will work for much longer than the standard license year, as long as you keep your hardware and software stable and IB continues to provide its API service.
- **Volume (multi-computer)** license: the same license as for a single computer, but when you purchase multiple licenses at once, you get a volume discount.
- **Site** license: enables to run **IBC** on an unlimited number of computers within the same Windows Domain. This license does not require activation by end-users, only a single centralized activation. It supports cloud deployment, where computer hardware fingerprints (but not the domain) often change.
- **Deployment (compiled)** license: enables to use **IBC** within a compiled program that runs on an unlimited number of computers. This license does not require separate activations by end-users, only a single centralized activation.
- **Source-code** license: unlimited in duration, can be installed on an unlimited number of computers in the organization, and requires no activation. This license requires signing a dedicated NDA (non-disclosure agreement).
- **Bundle** license: a discounted bundle of **IBC** and **IQML** (IQFeed-Matlab connector) or **EODML** (EODHistoricalData-Matlab connector) licenses.

2.3 Updating the static Java classpath

In **IBC** versions 1.97 and earlier, the installation included steps to add the *IBC.jar* file to Matlab's static Java classpath. This is no longer necessary in **IB-Matlab** version 1.98 (1/1/2018), but in some rare cases, a situation might arise where modifying the classpath might be necessary. For this reason, this step is detailed here:

1. 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\*).

2. Add the full path to the *IBC.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 **IBC** files are in *C:\IBC\*, then the new line in the *classpath.txt* file should be as follows:

   C:\IBC\IBC.jar

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

   When saving *classpath.txt*, you may 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 *classpath.txt*, select "Properties", and unselect the "Read-only" attribute. In Windows 7/8/10 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 `pwd` command at the Matlab command prompt to get it). Note that placing the modified *classpath.txt* file in your Matlab startup folder enables you to run **IB-Matlab** whenever you use this startup folder – so if you ever use a different Matlab startup folder, you'd 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, it is safer to update the *classpath.txt* file in Matlab's default location, namely the %matlabroot%/toolbox/local/ folder.

   As an alternative on some Matlab releases, create a *javaclasspath.txt* file in the startup folder, which just contains a single line, IBC*.jar*'s full path.

   ⚠ Note: **IBC** cannot receive IB data if Java's static classpath is not set.

3. Restart Matlab (no need to restart the computer or to run as administrator)

3 Using IBC

3.1 General usage

IBC uses the IB client (either TWS or IB Gateway) to connect to the IB server. Therefore, to use **IBC**, you must first ensure that either TWS or Gateway is active. If an active IB client is not detected, **IBC** 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, **IBC** will error.

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

```
>> help IBC
>> doc IBC
```

The *IBC* 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.[14]

IBC can accept input parameters in several formats:

- As name-value pairs – for example:

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

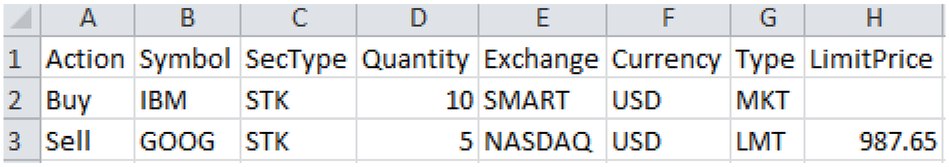
- As a struct (or struct array) of parameters – for example:

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

- As a table of parameters, with the parameter names as the table field names
- As field-separated rows in an input file (CSV or XLS) – for example:

  ```
  >> data = IBC('C:\MyData\inputFile.xlsx');
  ```

  Where row #1 of the file contains the parameter names, and rows 2+ contain the parameter values, one row per IBC command. For example:

	A	B	C	D	E	F	G	H
1	Action	Symbol	SecType	Quantity	Exchange	Currency	Type	LimitPrice
2	Buy	IBM	STK	10	SMART	USD	MKT	
3	Sell	GOOG	STK	5	NASDAQ	USD	LMT	987.65

  In CSV files you can add comments following the % character, for example:

  ```
  Action,Symbol,SecType,Quantity,Exchange,Currency,Type,LimitPrice
  Buy,IBM,STK,10,SMART,USD,MKT  % A Christmas gift for dad
  ```

All these input formats are equivalent. You can use whichever format that you feel comfortable with.

[14] See §15 below for more details

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

```
% 1st IBC 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 = IBC(params);

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

The second *IBC* 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 *IBC* command. To avoid unexpected results, I therefore advise to re-initialize the `params` struct (=[]) for each *IBC* command.

IBC is quite tolerant of user input: parameter names are case-insensitive (whereas most IB values are case-sensitive), parameter order does not matter, non-numeric parameters can be specified as either char arrays ('abc') or strings ("abc"), and some of these can be shortened. For example, the following are all equivalent:

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

The full list of acceptable input parameters is listed in the sections below, grouped by usage classification (action). Each action has a specific set of acceptable parameters.

When using *IBC*, there is no need to worry about connecting/disconnecting from IB: *IBC* automatically connects to whichever TWS/Gateway is currently active. If you logged-in to TWS/Gateway using a paper-trading username then *IBC* will work on the simulated account; if you login to a live account then *IBC* will connect to that account. The TWS account type is transparent to *IBC*: the only way to control whether *IBC* will use simulated or live trading is to login to TWS using the appropriate username. Refer to §13 below for additional details.

When using *IBC*, you may receive various types of error messages. Refer to §14 below for details about handling these messages. As a specific example, IB limits the rate of messages sent to the IB server, to 50 messages / second. If you exceed this rate, you will receive an error message from IB:

```
[API.msg2] Max rate of messages per second has been exceeded: max=50 rec=55
```

3.2 Contract properties

The following contract (security/ticker) properties can be specified in *IBC*:

Parameter	Data type	Default	Description
Symbol	string	(none)	The symbol of the underlying asset[16]
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' – stock equity and ETF (default) • 'OPT' – option • 'FUT' – future (CONTFUT=continuous) • 'IND' – index • 'FOP' – option on future • 'CASH' – Forex • 'WAR' – warrant • 'BOND' – bond • 'FUND' – mutual fund • 'IOPT' – structured (Dutch Warrant) • 'SSF' – single-stock future • 'CMDTY' – commodity • 'BAG' – combo-legs (see §9.5 below)
Exchange	string	'SMART'	The exchange that should process the request.[17] SMART uses IB's SmartRouting to optimize order execution time and cost.[18] To specify the primary exchange, use the : or / separator,[19] for example: 'SMART:ISLAND' or 'SMART/TSE'.
Currency	string	'USD'	The trading currency. This field can often be specified to avoid ambiguities (see §14.2).
Expiry	string	''	'YYYYMM' or 'YYYYMMDD' format Note: indicates last trading date not expiry date
Multiplier	number	[]	The contract multiplier (for options)
Strike	number	0.0	The strike price (for options)
Right	string	''	One of: 'P', 'PUT', 'C', 'CALL' (for options)
IncludeExpired	integer or logical flag	0=false	If true, expired options/futures are considered, otherwise they are not.
ConId	integer	[]	The contract ID (if known)
SecId	string	''	The security ID e.g. 'US0378331005' (Apple's ISIN). Must be specified with **SecIdType**.
SecIdType	string	''	The type of **SecID**. Either 'ISIN' or 'CUSIP'

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 = IBC('action','account_data')  % or: 'action','account'
data =
           AccountCode: 'DU12345'
           accountName: 'DU12345'
          AccountReady: 'true'
           AccountType: 'INDIVIDUAL'
           AccruedCash: [1x9 struct]
         AccruedCash_C: [1x1 struct]
         AccruedCash_S: [1x1 struct]
       AccruedDividend: [1x1 struct]
     AccruedDividend_C: [1x1 struct]
     AccruedDividend_S: [1x1 struct]
        AvailableFunds: [1x1 struct]

   (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:

```
>> myDividends = data.AccruedDividend
myDividends =
       value: 12345.67
    currency: 'AUD'
```

When the account has information in various currencies, the corresponding data field is an array of Matlab structs.[20] For example:

```
>> data.AccruedCash(1)  % A Matlab struct of a specific currency
ans =
       value: 1040
    currency: 'AUD'

>> data.AccruedCash(2)  % A Matlab struct of a specific currency
ans =
       value: 1039
    currency: 'BASE'

>> [data.AccruedCash.value]  % A numeric array of all currency values
ans =
    1040    1039      0      0      0     -7      0      0      0

>> {data.AccruedCash.currency}  % A corresponding Matlab cell array
ans =
    'AUD'  'BASE'  'CAD'  'CHF'  'DKK'  'NOK'  'NZD'  'SEK'  'USD'
```

[20] **IBC** versions prior to Oct 20, 2014 did not make a distinction between various currencies, so the reported value might be misleading if your account holds values in various currencies.

Some account data fields have several variants, for example, AccruedCash, AccruedCash_C and AccruedCash_S. Until you trade a commodity, AccruedCash_C=0 and AccruedCash_S=AccruedCash. After trading a commodity, AccruedCash_C holds the value for commodities, AccruedCash_S for securities, and AccruedCash for the total. Several other fields also have these _S and _C variants.

If your TWS account is linked to multiple IB accounts (as is common for financial advisors), then you should specify the **AccountName** input parameter, so that IB would know which IB account to access:[21]

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

To get data for all accounts in a consolidated manner, set **AccountName** to 'All':[22]

```
>> data = IBC('action','account', 'AccountName','All')
data =
            DF12344: [1x1 struct]
            DU12345: [1x1 struct]
            DU12346: [1x1 struct]
        SummaryData: [1x1 struct]
    ManagedAccounts: {'DF12344'  'DU12345'  'DU12346'}
```

where the returned struct fields contain the account data for each specific account, as shown at the beginning of this section, plus a `SummaryData` struct field for all accounts. The final field, `ManagedAccounts`, is a cell array of all the managed account names.

Note: IB has changed the behavior for **AccountName**='All' in 2015. The description above is accurate as of November 2015, but with your IB accounts you might still see the previous behavior, receiving a single unified data struct, as for a single account.

The **AccountName** parameter is only used when managing multiple accounts. If you manage just a single account, then the **AccountName** parameter is ignored - you will always receive the detailed data struct for the account, as shown at the beginning of this section, even if you specify an invalid **AccountName**, or omit it altogether.

In some cases, IB might return empty data in response to account requests. Two workarounds have been found for this, although they might not cover all cases. The workarounds are to simply re-request the information, and to force a programmatic reconnection to IB (more on connection issue in §13 below):

```
data = IBC('action','account');
if isempty(data)   % empty data - try to re-request the same data
   data = IBC('action','account');
end
if isempty(data)   % still empty data - try to disconnect/reconnect
   IBC('disconnect');               % disconnect from IB
   pause(1);                             % let IB cool down a bit
   data = IBC('action','account'); % will automatically reconnect
end
```

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

[22] TWS/IB-Gateway API setting "Master API Client ID" may need to be empty (even if correct) for this to work (see installation step 5d in §2 above).

4.2 Portfolio data

To retrieve an IB account's portfolio (list of held securities), use 'portfolio' **action**:

```
>> data = IBC('action','portfolio')
data =
1x12 struct array with fields:
    symbol
    localSymbol
    exchange
    secType
    currency
    right
    expiry
    strike
    position
    marketValue
    marketPrice
    averageCost
    realizedPnL
    unrealizedPnL
    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: 920
    marketValue: 171580
    marketPrice: 186.5
    averageCost: 169.03183335
    realizedPnL: 7513.78
  unrealizedPnL: 16070.71
       contract: [1x1 struct]
```

The `marketPrice` value is reflected in TWS's Quote Monitor as the "Mark Price". It is defined as the last price, clamped to ask (if ask<last) or bid (if bid>last) as needed.[23]

It is highly advisable for robustness to compare the account's `StockMarketValue`[24] to the sum of non-cash portfolio `marketValue`s. 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

[23] http://interactivebrokers.com/en/software/tws/usersguidebook/thetradingwindow/price-based.htm

[24] As reported by the `IBC('action','account')` command – see §4.1 for details

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 value of the security positions (i.e., their absolute `marketValue`s), then the sum in this case should equal the account's `GrossPositionValue`. Note that there may be 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, and 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.

In the returned data struct, the `contract` field is itself a struct, which contains basic information about the security.[25] The only important piece of information that is not already included in the main struct is the contract Id stored in `data.contract.m_conId`:

```
>> data(2).contract
ans =
        m_conId: 3691937
       m_symbol: 'AMZN'
      m_secType: 'STK'
     m_currency: 'USD'
  m_primaryExch: 'NASDAQ'
            ...
```

As with account data requests,, if multiple IB accounts are connected to our IB login, then we need to ensure that we request data for the correct account. Many frustrations can be avoided by specifically stating the **AccountName** parameter whenever we use *IBC* in a multi-account environment. If you are unsure of the account name, set **AccountName** to 'All' (read the related discussion at the end of §4.1):

```
>> data = IBC('action','portfolio', 'AccountName','All')
data =
            DF12344: [0x0 struct]
            DU12345: [1x7 struct]
            DU12346: [1x3 struct]
```

As with account data requests, IB might return empty data in response to portfolio requests. Two workarounds have been found for this, although they might not cover all cases.[26] The workarounds are to simply re-request the information, and to force a programmatic reconnection to IB (more on the connection issue in §13 below):

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

[25] Use §5.4 below to retrieve detailed contract information; the fields are explained here: http://interactivebrokers.com/php/whiteLabel/Interoperability/Socket_Client_Java/java_properties.htm

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

In some cases, even with the retry workaround above, IB still returns empty portfolio data. A more reliable (and much faster) mechanism for retrieving portfolio data is to limit the request only to the portfolio positions, by setting the **Type** parameter to 'positions'. IB will return the data much faster and more reliably, except for the `marketValue`, `marketPrice`, and `averageCost` fields, which are returned empty:

```
>> data = IBC('action','portfolio', 'type','positions');
>> data(5)
ans =
         symbol: 'ZL'
    localSymbol: 'ZL   DEC 15'
       exchange: ''
        secType: 'FUT'
       currency: 'USD'
          right: ''
         expiry: '20151214'
         strike: 0
       position: -1
    marketValue: []
    marketPrice: []
    averageCost: []
    realizedPnL: []
  unrealizedPnL: []
       contract: [1x1 struct]
```

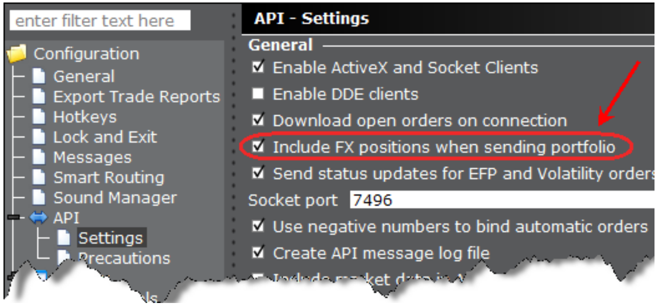
In this example, we have a short position of -1 for the ZL 12/2015 future, and no market information is included in the returned data.

Here is a short Matlab code example showing how to retrieve the position (number of portfolio shares) of a specific security ('GOOG' in this example):

```
portfolioData = IBC('action','portfolio', 'type','positions');
symbols = {portfolioData.localSymbol};
idx = strcmpi('GOOG', symbols);
position = portfolioData(idx).position;
if isempty(position)
    position = 0;
end
```

The position information is often sufficient. For example, an automated trading algorithm may need to determine if a position is currently open, and to compute the trade-order quantity required to open/reverse/close it. In such cases, limiting the portfolio request to position-only data is advisable. If you also need market data, you can use a standard portfolio request, or to retrieve the market data in a separate query.

Finally, note that IB will only send Forex (cash) position in the portfolio data if the relevant option is selected in the API settings:



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

Socket port 7496

☑ Use negative numbers to bind automatic orders

☑ Create API message log file

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 = IBC('action','query', 'symbol','GOOG')
data = 
          reqId: 22209874
        reqTime: '02-Dec-2010 00:47:23'
       dataTime: '02-Dec-2010 00:47:24'
  dataTimestamp: 734474.032914491
  lastEventTime: 734474.032914512
         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]
```

Here is another example, this time for a future asset:

```
>> data = IBC('action','query', 'LocalSymbol','YI   JUL 17', ...
                   'SecType','FUT', 'Exchange','NYSELIFFE')
data = 
            reqId: 727929834
          reqTime: '11-May-2017 10:23:11'
         dataTime: '11-May-2017 10:23:12'
    dataTimestamp: 736826.432780035
    lastEventTime: 736826.432780521
           ticker: ''
         bidPrice: 16.263
         askPrice: 16.271
             open: 16.197
            close: 16.207
              low: 16.18
             high: 16.285
        lastPrice: 16.285
           volume: 25
             tick: 0.01
          bidSize: 3
          askSize: 3
         lastSize: 1
```

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: `reqTime`, `dataTime`, `dataTimestamp` and `lastEventTime` fields reflect local time. If `lastPrice` 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[27] as a string, for example: ‘1348563149’). We can convert`lastTimestamp` into Matlab format by converting the string into a number using Matlab’s ***datestr*** and ***datenum*** functions:[28]

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

To retrieve live and historic market data, several 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)
4. There is no other TWS with live data running on a different computer[29]
5. If you manage several accounts, they should be associated with the main account for live data, using the Manage Accounts window in TWS.

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,[30] as the following examples illustrate:

```
>> data = IBC('action','query', 'symbol','GOOG')
[API.msg2] Requested market data is not subscribed.Error&BEST/STK/Top
{153745220, 354}
data =
              reqId: 779564761
            reqTime: '30-Oct-2017 13:27:49'
           dataTime: '30-Oct-2017 13:27:53'
```

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

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

[29] IB only sends live/historic data to a single computer, so retrieving such data requires IBC to be connected to the TWS that gets the live data, not to another TWS on a different computer. So if you connect TWS to your live account on computer A, and another TWS to your paper-trading account on computer B, then **IBC** can retrieve data only via computer A.

[30] The error messages can be suppressed using the **MsgDisplayLevel** parameter, and can also be trapped and processed using the **CallbackMessage** parameter – see §14.1 below for details

```
  dataTimestamp: 736998.561032893
  lastEventTime: 736998.561035394
         ticker: 'GOOG'
       bidPrice: -1
       askPrice: -1
           open: -1
          close: -1
            low: -1
           high: -1
      lastPrice: -1
         volume: -1
           tick: 0.01
       contract: [1×1 struct]
contractDetails: [1×1 struct]
```

This illustrates a situation where we are not subscribed to data for this specific security type and/or exchange, and delayed market data is not available. When delayed market data is available (depending on **Exchange** and **SecType**), the delayed quotes are shown in dedicated fields, together with an explanatory warning message:

```
>> data = IBC('action','query', 'symbol','AAPL')
[API.msg2] Requested market data is not subscribed. Displaying delayed
market data... {779573827, 10167}
data =
                reqId: 779573829
              reqTime: '30-Oct-2017 14:17:10'
             dataTime: '30-Oct-2017 14:17:11'
        dataTimestamp: 736998.595277546
        lastEventTime: 736998.595283414
               ticker: 'AAPL'
             bidPrice: -1
             askPrice: -1
                 open: -1
                close: -1
                  low: -1
                 high: -1
            lastPrice: -1
               volume: -1
                 tick: 0.01
             contract: [1×1 struct]
      contractDetails: [1×1 struct]
     delayed_bidPrice: 163.5
     delayed_askPrice: 163.59
    delayed_lastPrice: 163.5
       delayed_volume: 991
        delayed_close: 163.05
         delayed_open: 0
```

Note that some of the delayed data fields may be missing or invalid. Specifically, the `delayed_close` field is very often not provided, and the `delayed_open` field is often 0.

A similar yet different error message is returned when we try to get historical data for a security type or exchange that is not subscribed:

```
>> data = IBC('action','history', 'symbol','GOOG')
[API.msg2] Historical Market Data Service error message:
No market data permissions for ISLAND STK {153745241, 162}
data =
    dateTime: {1x0 cell}
        open: []
        high: []
         low: []
       close: []
      volume: []
       count: []
         WAP: []
     hasGaps: []
```

In this case, there is no delayed data: IB only provides delayed quotes for single-quote and streaming data. IB will not return historical data if you do not have the necessary market perrmissions/subscription.

If we specify an incorrect security name or classification properties, then the data is similarly not returned, and an error message is displayed (see §14.2).

In some cases, querying a contract may return some invalid (<0) field values. For example, querying the NIFTY50 index 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 has no bid/ask/open/high/low/volume values. Only NIFTY50 futures are tradable securities, and these indeed return valid field values. Another common reason for receiving -1 field values is querying when the market is closed. To prevent **IB-Matlab** from waiting a long time for the missing fields, set the **Timeout** parameter.

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' or '236'.[31] Note that the value should be a string ('236'), not a number (236).

```
>> data = IBC('action','query', 'symbol','GOOG', ...
                   'GenericTickList','236');  % Note: '236', not 236
```

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 [32] (not for regular market queries). When the feature is supported, an additional `shortable` field is returned with basic information about the security's shortability. [33] Multiple tick types can be specified, separated by comma. For example: `'GenericTickList','233,236,258'` .[34]

[31] https://interactivebrokers.github.io/tws-api/tick_types.html

[32] See §7 for details on streaming quotes

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

[34] https://interactivebrokers.github.io/tws-api/md_request.html#genticks

5.2 Market depth (Level II) data

For some exchanges and security types, and possibly also depending on your IB account subscription, IB enables the user to retrieve market depth (Level II order book) information. In **IBC**, this information can either be retrieved as a one-time data snapshot, or in continuous streaming mode. To get snapshot data, set the **QuotesNumber** parameter to 1 (=default); to get streaming data set **QuotesNumber** to a larger value. This section describes the one-time data snapshot mechanism; see §7.3 below for a description of the streaming data mechanism, and §11.5 for a usage example of a continuously-updating order-book GUI.

In general, retrieving market depth is very similar to retrieving single-quote data, the only difference being that for market depth, the **NumberOfRows** parameter should be set to a positive number between 2-5 (i.e., 2, 3, 4 or 5).[35]

Here is a simple example of retrieving the top 3 rows of the EUR.USD order-book:

```
>> dataStruct = IBC('action','query', 'symbol','EUR', ...
                    'LocalSymbol','EUR.USD', 'SecType','CASH', ...
                    'Exchange','IDEALPRO', 'NumberOfRows',3)
dataStruct =
                reqId: 464868253
              reqTime: '16-Dec-2014 14:03:48'
     lastEventDateNum: 735949.585989664
    lastEventDateTime: '16-Dec-2014 14:03:49'
               symbol: 'EUR'
          localSymbol: 'EUR.USD'
             isActive: 0
       quotesReceived: 6
      quotesToReceive: 6
                  bid: [1x3 struct]
                  ask: [1x3 struct]
             contract: [1x1 struct]

>> dataStruct.bid(1)
ans =
          price: 1.25345
           size: 1000000
    marketMaker: ''
        dateNum: 735949.585989618
       dateTime: '16-Dec-2014 14:03:49'

>> dataStruct.ask(1)
ans =
          price: 1.2535
           size: 8320000
    marketMaker: ''
        dateNum: 735949.585989653
       dateTime: '16-Dec-2014 14:03:49'
```

[35] The default value of **NumberOfRows** (=1) in indicates a single-quote query rather than a market-depth query.

Note that in this case, 6 quotes were received, corresponding to the requested 3 rows for both the bid and the ask sides. You can request up to 10 market-depth rows (some exchanges may limit the number of available market-depth rows to a lower number). Note that in some cases, the market depth of the bid/ask sides may not be the same. For example, it is possible that at some time there are 5 bid rows, but only 4 ask rows.

Naturally, `dataStruct.bid(1)` is the highest bid, `dataStruct.ask(1)` is the lowest ask:

```
>> [dataStruct.bid.price]
ans =
         1.25345          1.2534         1.25335

>> [dataStruct.ask.price]
ans =
          1.2535         1.25355          1.2536
```

In some cases (again, depending on the market, security and your IB subscription level), Level 2 market-maker date may be available. In such cases, the `marketMaker` field will contain the name of the exchange hosting the order for that row.

As noted above, market depth is not always available. Only certain combinations of **Exchange**, **SecType** and **Currency** are supported for Level 2, even if you have the necessary market-data subscription. For example, the 'SMART' exchange is typically not supported, only specific exchange names. In all such cases, you may receive an applicable error message from the IB server, and the returned data will be empty:

```
>> data = IBC('Action','query', 'Symbol','IBM', 'NumberOfRows',3)
[API.msg2] Deep market data is not supported for this combination of
security type/exchange {464879606, 10092}
data =
                reqId: 464879606
              reqTime: '16-Dec-2014 14:22:06'
     lastEventDateNum: -1
    lastEventDateTime: ''
               symbol: 'IBM'
          localSymbol: ''
             isActive: 1
       quotesReceived: 0
      quotesToReceive: 6
                  bid: [0x0 struct]
                  ask: [0x0 struct]
             contract: [1x1 struct]
```

5.3 Scanner data

IB's scanner data functionality returns a list of securities that match the specified scan criteria. IB provides a long list of predefined scanners, [36] including MOST_ACTIVE, TOP_PERC_GAIN, HOT_BY_VOLUME, HOT_BY_PRICE etc. The scan can be limited to security type and trading location, 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.).[37] 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. For example, consider using finviz.com, which provides a very detailed online scanner (free for online browsing; premium service to download CSV data file)[38]. Many additional screeners are available online.[39]

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

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

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, **IBC** 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:

[36] http://interactivebrokers.com/en/software/webtrader/webtrader.htm#webtrader/marketscanners/about market scanners.htm

[37] http://interactivebrokers.github.io/tws-api/classIBApi_1_1ScannerSubscription.html

[38] http://finviz.com/screener.ashx

[39] For example: http://nasdaq.com/reference/stock-screener.aspx, http://caps.fool.com/Screener.aspx, http://finance.google.com/finance#stockscreener, http://screener.finance.yahoo.com/stocks.html, http://stockscreener.us.reuters.com/Stock/US/index, http://marketwatch.com/tools/stockresearch/screener

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

>> dataStruct(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: You will only receive scan data that corresponds to your paid IB market subscription. For example, if you are only subscribed to NASDAQ data but not to NYSE or other exchanges, then you will only receive NASDAQ scan results, regardless of your specified **LocationCode**. In other words, scanner parameters only narrow down (filter) scan results; they cannot be used to provide unsubscribed data.

IB’s 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 **IBC**:

```
>> xmlStr = IBC('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,AVGVOLUME,AVGVOLU
         ME_USD,HALTED,...,FIRSTTRADEDATE,HASOPTIONS</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,PRODCAT,LEADFUT,C
         HANGEPERC,CHANGEOPENPERC,OPENGAPPERC,PRICERANGE,TRADERATE</f
         ilters>
         <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 **IBC** 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 = IBC('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 IB's Java API:[40]

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

[40] https://interactivebrokers.github.io/tws-api/classIBApi_1_1ScannerSubscription.html

Parameter	Data type	Default	Description
Instrument	string	'STK'	Defines the instrument type
LocationCode	string	'STK.NASDAQ'	Defines the scanned markets
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 maximal number of rows of data to return for the query
ScanCode	string	'MOST_ACTIVE'	A long list... - see the API doc[41]
ScannerSettingPairs	string	" (empty string)	For example, a pairing of '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'

[41] https://www.interactivebrokers.com/en/software/webtrader/webtrader/marketscanners/about%20market%20scanners.htm. The latest version of this webpage only contains a description of the scanners, not their scan codes (e.g. "Most active" instead of MOST_ACTIVE). To get the scan codes, see the Reference chapter, "Available Market Scanners" section (around p. 566) in IB's downloadable API documentation: http://institutions.interactivebrokers.com/download/newMark/PDFs/APIprintable.pdf. Alternatively, you could ask IB support to tell you the specific scan codes for your requested market scanners.

5.4 Contract details and options chain

Contract details for any security can be retrieved using the parameters **Action**='query' with **Type**='contract'. If the security is well-defined, then **IBC** will return a data struct containing various details on the contract, which is basically a merge of the contract and contractDetails structs that are returned by the single-quote query (§5.1). For example:

```
>> dataStruct = IBC('action','contract', 'symbol','IBM')
dataStruct =
                m_conId: 8314
               m_symbol: 'IBM'
              m_secType: 'STK'
               m_expiry: []
               m_strike: 0
                m_right: []
           m_multiplier: []
             m_exchange: 'SMART'
             m_currency: 'USD'
          m_localSymbol: 'IBM'
          m_primaryExch: 'NYSE'
       m_includeExpired: 0
            m_secIdType: []
                m_secId: []
     m_comboLegsDescrip: []
            m_comboLegs: [0 java.util.Vector]
            m_underComp: []
              m_summary: [1x1 com.ib.client.Contract]
           m_marketName: 'IBM'
         m_tradingClass: 'IBM'
              m_minTick: 0.01
       m_priceMagnifier: 1
           m_orderTypes: 'ACTIVETIM,ADJUST,ALERT,ALGO,ALLOC,AON,
AVGCOST,BASKET,COND,CONDORDER,DARKPOLL,DAY,DEACT,DEACTDIS,DEACTEOD,DIS,
GAT,GTC,GTD,GTT,HID,IBDARK,ICE,IMB,IOC,LIT,LMT,LOC,MIT,MKT,MOC,MTL,...'
       m_validExchanges: 'SMART,NYSE,CBOE,ISE,CHX,ARCA,ISLAND,VWAP,
IBSX,DRCTEDGE,BEX,BATS,EDGEA,LAVA,CSFBALGO,JEFFALGO,BYX,IEX,TPLUS2,PSX'
           m_underConId: 0
             m_longName: 'INTL BUSINESS MACHINES CORP'
        m_contractMonth: []
             m_industry: 'Technology'
             m_category: 'Computers'
          m_subcategory: 'Computer Services'
           m_timeZoneId: 'EST'
         m_tradingHours: '20150325:0400-2000;20150326:0400-2000'
          m_liquidHours: '20150325:0930-1600;20150326:0930-1600'
                    ...
(many additional data fields, some of them empty)
```

Note that the data is returned even outside market trading hours, unlike the single-quote query that typically returns empty pricing data outside trading hours. Also note that no pricing information is returned, only the contract information.

Retrieving the options chain of a security uses the same mechanism. In this case, when **SecType**='OPT' (stock options) or 'FOP' (futures options), multiple option contracts are returned in an array of data structs similar to the above, for those options that match our query. For example, to retrieve all futures options for the 10-year US Treasury Note (ZN), which have a contract month of December 2015:

```
>> dataStruct = IBC('action','contract', 'symbol','ZN', ...
                    'secType','FOP', 'expiry','201512', ...
                    'exchange','ecbot')
dataStruct =
224x1 struct array with fields:
    m_conId
    m_symbol
    ...
>> dataStruct(1)
ans =
               m_conId: 168043528
              m_symbol: 'ZN'
             m_secType: 'FOP'
              m_expiry: '20151120'
              m_strike: 128.5
               m_right: 'P'
          m_multiplier: '1000'
            m_exchange: 'ECBOT'
            m_currency: 'USD'
         m_localSymbol: 'P OZN  DEC 15  12850'
                   ...
>> dataStruct(2)
ans =
               m_conId: 168043533
              m_symbol: 'ZN'
             m_secType: 'FOP'
              m_expiry: '20151120'
              m_strike: 131
               m_right: 'P'
          m_multiplier: '1000'
            m_exchange: 'ECBOT'
            m_currency: 'USD'
         m_localSymbol: 'P OZN  DEC 15  13100'
                   ...
```

Note: we need to specify the **SecType** and **Exchange** for options, since IB cannot find the security using the default parameter values ('STK' and 'SMART', respectively):[42]

```
>> dataStruct = IBC('action','contract', 'symbol','ZN', ...
                    'secType','FOP', 'expiry','201512')
[API.msg2] No security definition has been found for the request
{494601749, 200}
dataStruct =
     []
```

[42] **Exchange** can be omitted (or set to 'SMART') for stock options but not for futures options; **SecType** must be set in either case.

Also note that the reported `m_expiry` field is the last trading date for the contract (in this case, November 20, 2015), not the contract's actual expiration date (Dec' 2015).

We can limit the results by specifying a combination of the **Expiry**, **Strike**, **Multiplier** and/or **Right** parameters. For example, to limit ZN options only to Calls:

```
>> dataStruct = IBC('action','contract', 'symbol','ZN', ...
                         'secType','FOP', 'expiry','201512', ...
                         'exchange','ecbot', 'right','Call')
dataStruct =
112x1 struct array with fields:
    m_conId
    m_symbol
    m_secType
    ...
```

Similarly, to get all options (Calls & Puts), in all expiry dates, that have **Strike**=$130:

```
>> dataStruct = IBC('action','contract', 'symbol','ZN', ...
                         'secType','FOP', 'strike',130, ...
                         'exchange','ecbot')
dataStruct =
22x1 struct array with fields:
    m_conId
    m_symbol
    m_secType
    ...
```

To retrieve the full options-chain without any filtering, we can just remove the limiting parameters. Note that it takes a few seconds for all the thousands of possible contract details to be sent from IB:

```
>> dataStruct = IBC('action','contract', 'symbol','IBM', ...
                         'secType','OPT', 'exchange','CBOE2')
dataStruct =
900x1 struct array with fields:
    m_conId
    m_symbol
    m_secType
    ...
```

Note that the options are not necessarily ordered in any way: you should test the field values, not rely on the order of the contracts in the returned dataStruct.

A different way of retrieving the options chain is explained in §11.4 below, using IB event callabacks. The difference between the mechanism here and in §11.4 is that the the command here is synchronous (i.e., Matlab waits for all the data to be received from IB before returning a unified dataStruct). In §11.4, the contracts data are received and processed in parallel (asynchronously) to the main Matlab program.

Finally, note that 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**). We can only extract the full option chain list of contract names and details in a single command, as shown above.

5.5 Fundamental data

IB’s fundamental data functionality returns Reuters global fundamental data for stocks. You must have a subscription to Reuters Fundamental set up in your IB Account Management before you can receive most of the reports. The following data reports are available using this functionality:[43]

- ReportSnapshot (company overview)
- ReportsOwnership (company ownership; note: may be large in size)
- ReportRatios (financial ratios; note: may not be available in some cases)
- ReportsFinSummary (financial summary)
- ReportsFinStatements (financial statements)
- RESC (analyst estimates)
- CalendarReport (company calendar)
- Ratios (fundamental ratios – different from other reports: see below)

Note: Fundamental data is only available for stocks, not for any other security type.

To use IB’s market scanner in **IBC**, set the **Action** parameter to 'Fundamental', and the **Type** parameter to one of the report names above, and specify the requested contract information (**Symbol**, **Exchange**, **Currency** etc.). For example:

```
>> xmlStr = IBC('Action','fundamental','Type','ReportSnapshot',...
                     'Symbol','IBM')
xmlStr =
<?xml version="1.0" encoding="UTF-8"?>
<ReportSnapshot Major="1" Minor="0" Revision="1">
   <CoIDs>
      <CoID Type="RepNo">4741N</CoID>
      <CoID Type="CompanyName">International Business Machines Corp.</CoID>
      <CoID Type="IRSNo">130871985</CoID>
      <CoID Type="CIKNo">0000051143</CoID>
   </CoIDs>
   <Issues>
      <Issue ID="1" Type="C" Desc="Common Stock" Order="1">
         <IssueID Type="Name">Ordinary Shares</IssueID>
         <IssueID Type="Ticker">IBM</IssueID>
         <IssueID Type="CUSIP">459200101</IssueID>
         <IssueID Type="ISIN">US4592001014</IssueID>
         <IssueID Type="RIC">IBM</IssueID>
         <IssueID Type="SEDOL">2005973</IssueID>
         <IssueID Type="DisplayRIC">IBM.N</IssueID>
         <IssueID Type="InstrumentPI">261483</IssueID>
         <IssueID Type="QuotePI">1090370</IssueID>
         <Exchange Code="NYSE" Country="USA">New York Stock
Exchange</Exchange>
         <MostRecentSplit Date="1999-05-27">2.0</MostRecentSplit>
      </Issue>
      <Issue ID="2" Type="P" Desc="Preferred Stock" Order="1">
         <IssueID Type="Name">Preference Shares Series A</IssueID>
```

[43] https://interactivebrokers.github.io/tws-api/fundamentals.html

```
         <IssueID Type="Ticker">IBMPP</IssueID>
         <IssueID Type="CUSIP">459200200</IssueID>
         <IssueID Type="ISIN">US4592002004</IssueID>
         <IssueID Type="RIC">IBMPP.PK^C06</IssueID>
         <IssueID Type="InstrumentPI">1883112</IssueID>
         <IssueID Type="QuotePI">25545447</IssueID>
         <Exchange Code="OTC" Country="USA">Over The Counter</Exchange>
      </Issue>
   </Issues>
   <CoGeneralInfo>
      <CoStatus Code="1">Active</CoStatus>
      <CoType Code="EQU">Equity Issue</CoType>
      <LastModified>2016-06-29</LastModified>
      <LatestAvailableAnnual>2015-12-31</LatestAvailableAnnual>
      <LatestAvailableInterim>2016-03-31</LatestAvailableInterim>
      <Employees LastUpdated="2015-12-31">377757</Employees>
      <SharesOut Date="2016-03-31"
TotalFloat="959386999.0">959961852.0</SharesOut>
      <CommonShareholders Date="2015-12-31">444582</CommonShareholders>
      <ReportingCurrency Code="USD">U.S. Dollars</ReportingCurrency>
      <MostRecentExchange Date="2016-07-07">1.0</MostRecentExchange>
   </CoGeneralInfo>
   <TextInfo>
      <Text Type="Business Summary" lastModified="2016-04-
21T03:03:49">International Business Machines Corporation (IBM) is a
technology company...
...
```

The fundamental data is returned as an XML string by default, as shown above. This XML string can be quite long, depending on the requested report and security. We can store this string in a **.xml* file and open this file in an XML reader (for example, a browser). Alternatively, we can ask **IBC** to parse this XML and return a simple Matlab struct by setting **ParametersType**='struct'. Note: this XML parsing could take a long time (a full minute or even longer in some cases, such as long RESC reports):[44]

```
>> data = IBC('Action','fundamental','Type','ReportSnapshot',...
                   'Symbol','IBM', 'ParametersType','struct')
data =
             Name: 'ReportSnapshot'
       Attributes: [1×1 struct]
            CoIDs: [1×1 struct]
           Issues: [1×1 struct]
    CoGeneralInfo: [1×1 struct]
         TextInfo: [1×1 struct]
      contactInfo: [1×1 struct]
         webLinks: [1×1 struct]
         peerInfo: [1×1 struct]
         officers: [1×1 struct]
           Ratios: [1×1 struct]
     ForecastData: [1×1 struct]
```

[44] Note: the format of the returned struct was simplified and improved starting in *IBC* version 2.10 (2019-12-31)

```
>> data.Issues.Issue(2).IssueID(4)
ans = 
          Name: 'IssueID'
    Attributes: [1×1 struct]
          Data: 'US4592002004'

>> data.Issues.Issue(2).IssueID(4).Attributes
ans = 
    Type: 'ISIN'
```

In some cases, some of the fundamental reports may not be available for a certain security for some reason, while other reports for the same security may be available:

```
>> data = IBC('Action','fundamental', 'Symbol','IBM', ...
                   'Type','ReportRatios')
[API.msg2] We are sorry, but fundamentals data for the security
specified is not available. failed to fetch {636789740, 430}
data =
     []

>> data = IBC('Action','fundamental', 'Symbol','IBM', ...
                   'Type','CalendarReport')
[API.msg2] We are sorry, but fundamentals data for the security
specified is not available. Not allowed {636789744, 430}
data =
     []
```

The fundamental ratios report (**Type** = 'Ratios') differs from the other reports in several aspects: it does not require a Reuters subscription; it is always returned in struct (not XML) format regardless of the **ParametersType** value; and it is only available when the security trades. The reason for this is that this report uses IB's internal mechanism for reporting fundamental ratios of streaming quotes (see §7.1 below) with a **GenericTickList** of '258', [45] rather than Reuters data. For a description of the various data fields, refer to IB's documentation.[46]

```
>> data=IBC('Action','fundamental','Symbol','IBM','Type','Ratios')
data = 
       TTMNPMGN: 16.03988
           NLOW: 116.901
     REVTRENDGR: -3.92708
     TTMEPSXCLX: 13.27684
       QTANBVPS: -24.7614
      TTMPRCFPS: 17.78968
     TTMGROSMGN: 49.38146
       TTMCFSHR: 15.23175
      QCURRATIO: 1.37385
       PRICE2BK: 9.93472
         MKTCAP: 148275.7
            ...
     (and so on: dozens of different fields)
```

[45] http://interactivebrokers.github.io/tws-api/tick_types.html

[46] http://interactivebrokers.github.io/tws-api/fundamental_ratios_tags.html

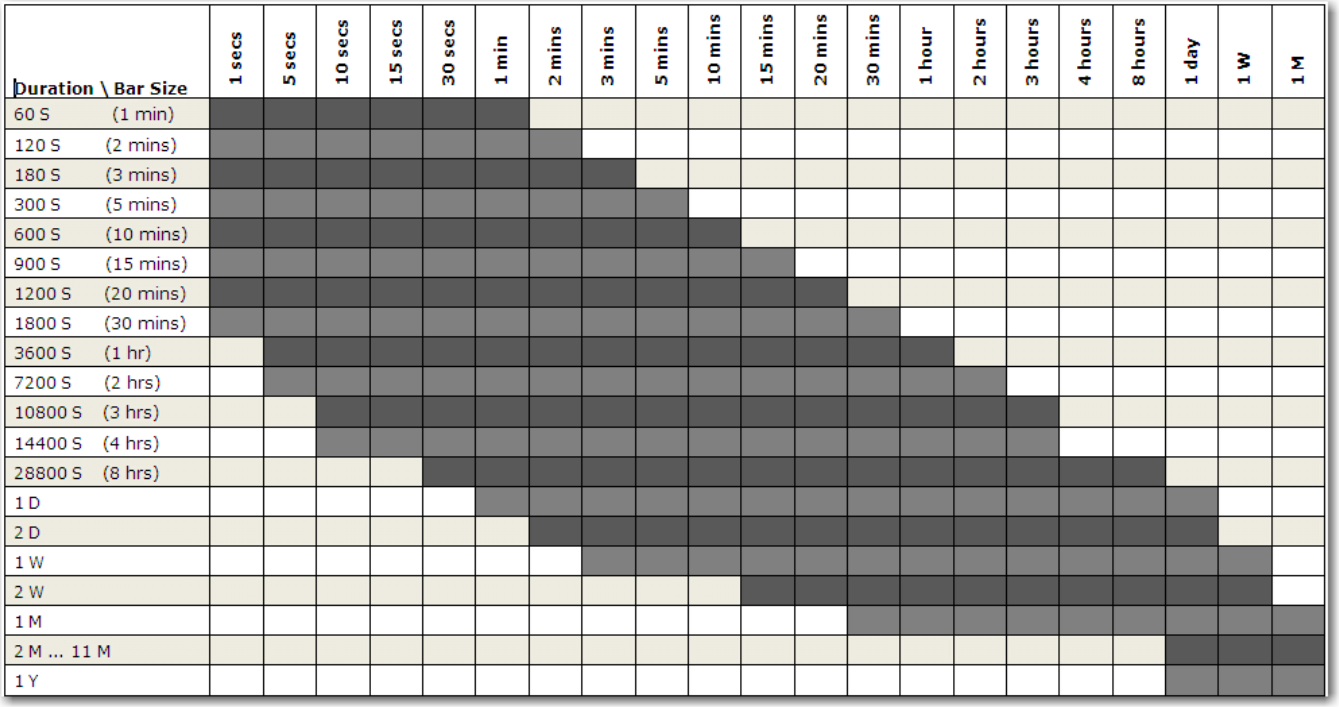
The parameters that affect fundamental data retrieval are as follows (for the contract properties, see §3.2 above):

Parameter	Data type	Default	Description
Type	string	'Ratios'	One of: • 'Ratios' (default) – fundamental ratios • 'ReportSnapshot' – company overview • 'ReportRatios' – financial ratios • 'ReportsOwnership' – company owners • 'ReportsFinSummary' – financial summary • 'ReportsFinStatements' – financial statements • 'RESC' – analyst estimates • 'CalendarReport' – company calendar
ParametersType	string	'XML'	One of: • 'XML' (default) • 'struct' – Matlab struct

6 Querying historical and intra-day data

Historical data can be retrieved from IB,[47] subject to your account subscription rights, and IB's lengthy list of pacing violation limitations. [48] Note that these are IB server limitations, not **IBC** limitations. As of Nov 2015, these limitations include:

1. Historical data is limited by default to 2000 results (data bars). You may have access to more results depending on your IB subscription level. If you request more results than your limit, the entire request is dropped.
2. Historical data is limited by default to the past year. If you purchase additional concurrent real-time market data-lines from IB you can access up to 5 years of history. If you request data older than your limit, the entire request is dropped.
3. Historical data requests that use a small (<1 min) bar size can only go back 6 months. If older data is requested, the entire request is dropped.
4. Requesting identical historical data requests within 15 seconds is prohibited. **IBC** will automatically return the previous results in such a case.
5. Requesting 6+ historical data requests having small bar-size and same contract, exchange, tick type within 2 seconds is prohibited; the request will be dropped.
6. Requesting 60+ historical data requests having small bar-size of any type within 10-minutes is prohibited; the entire request will be dropped.
7. Only certain combinations of **Duration** and **BarSize** are supported:[49]



Duration \ Bar Size	1 secs	5 secs	10 secs	15 secs	30 secs	1 min	2 mins	3 mins	5 mins	10 mins	15 mins	20 mins	30 mins	1 hour	2 hours	3 hours	4 hours	8 hours	1 day	1 W	1 M
60 S (1 min)	■	■	■	■	■	■															
120 S (2 mins)	■	■	■	■	■	■	■														
180 S (3 mins)	■	■	■	■	■	■	■	■													
300 S (5 mins)	■	■	■	■	■	■	■	■	■												
600 S (10 mins)	■	■	■	■	■	■	■	■	■	■											
900 S (15 mins)	■	■	■	■	■	■	■	■	■	■	■										
1200 S (20 mins)	■	■	■	■	■	■	■	■	■	■	■	■									
1800 S (30 mins)	■	■	■	■	■	■	■	■	■	■	■	■	■								
3600 S (1 hr)		■	■	■	■	■	■	■	■	■	■	■	■	■							
7200 S (2 hrs)		■	■	■	■	■	■	■	■	■	■	■	■	■	■						
10800 S (3 hrs)			■	■	■	■	■	■	■	■	■	■	■	■	■	■					
14400 S (4 hrs)			■	■	■	■	■	■	■	■	■	■	■	■	■	■					
28800 S (8 hrs)					■	■	■	■	■	■	■	■	■	■	■	■	■	■			
1 D						■	■	■	■	■	■	■	■	■	■	■	■	■	■		
2 D							■	■	■	■	■	■	■	■	■	■	■	■	■		
1 W								■	■	■	■	■	■	■	■	■	■	■	■	■	
2 W											■	■	■	■	■	■	■	■	■	■	
1 M													■	■	■	■	■	■	■	■	■
2 M ... 11 M																			■	■	■
1 Y																			■	■	■

[47] http://interactivebrokers.github.io/tws-api/historical_bars.html

[48] http://interactivebrokers.github.io/tws-api/historical_limitations.html

[49] Note: "1 secs", not "1 sec". Note the distinction from "1 min", "1 hour", and "1 day".

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).

IBC does not prevent you from entering invalid **Duration**s and **BarSize**s – 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 in the Matlab command window:

```
[API.msg2] Error validating request:-'qd' : cause - Historical data
bar size setting is invalid. Legal ones are: 1 secs, 5 secs, 10 secs,…
```

Another limitation is that retrieving historical data must be done from the same computer (IP) as the trading TWS:

```
[API.msg2] Historical Market Data Service error message: Trading TWS
session is connected from a different IP address {527921821, 162}
```

Of course, you must have an IB subscription for data from the requested exchange:

```
[API.msg2] Historical Market Data Service error message: No market
data permissions for NYSE STK {527921824, 162}
```

Also note that historical data retrieval is subject to the same pre-conditions as for retrieving the current live market data (see §5.1). If any of these limitations is not met, then an error message will be displayed and no data will be returned.

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

```
>> data = IBC('action','history', 'symbol','IBM', ...
                   'barSize','1 hour', 'useRTH',1)
data =
        dateNum: [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:

- `dateNum` – a numeric array of date/time values in Matlab's numeric format[50]
- `dateTime` – a cell-array of date strings, or a numeric array of date values in IB format (see the **FormatDate** parameter, explained below). Intra-day bars use local timezone; daily bars use exchange timezone.
- `open` – the bar's opening price
- `high` – the high price during the time covered by the bar

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

- `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

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
```

The following parameters affect historical data retrieval:[51]

Parameter	Data type	Default	Description
EndDateTime	string	'' (empty string), meaning now	'YYYYMMDD hh:mm:ss TMZ' format (the TMZ time zone is optional[52])
BarSize	string	'1 min'	Size of data bars to be returned (within IB/TWS limits). Valid values include: • 1 sec, 5/10/15/30 secs • 1 min (default) • 2/3/5/10/15/20/30 mins • 1 hour, 2/3/4/8 hours • 1 day, 1 w, 1 m
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)

[51] http://interactivebrokers.github.io/tws-api/historical_bars.html

[52] 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 type of data to return:[53] • 'Trades' (default; invalid for Forex) • 'Midpoint' • 'Bid' • 'Ask' • 'Bid_Ask' (see usage note below [54]) • 'Adjusted_Last' • 'Historical_Volatility' (STK/ETF/IND) • 'Option_Implied_Volatility' (as above) • 'Rebate_Rate' • 'Fee_Rate' • 'Yield_Bid' (corp. bonds only) • 'Yield_Ask' (corp. bonds only) • 'Yield_Bid_Ask' (corp. bonds only) • 'Yield_Last' (corp. bonds only)
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 was outside of its regular trading hours • 1 or true: only data within 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 data bars. Valid values include: 1) strings: '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). Only supported for **BarSize** < 1 day.
Timeout	number	Inf = unlimited	Max # of secs to wait for an IB response to a request. The timeout is ignored after partial data has been received.
IncludeExpired	integer or logical flag	0=false	If true, expired contracts are considered, otherwise they are not.

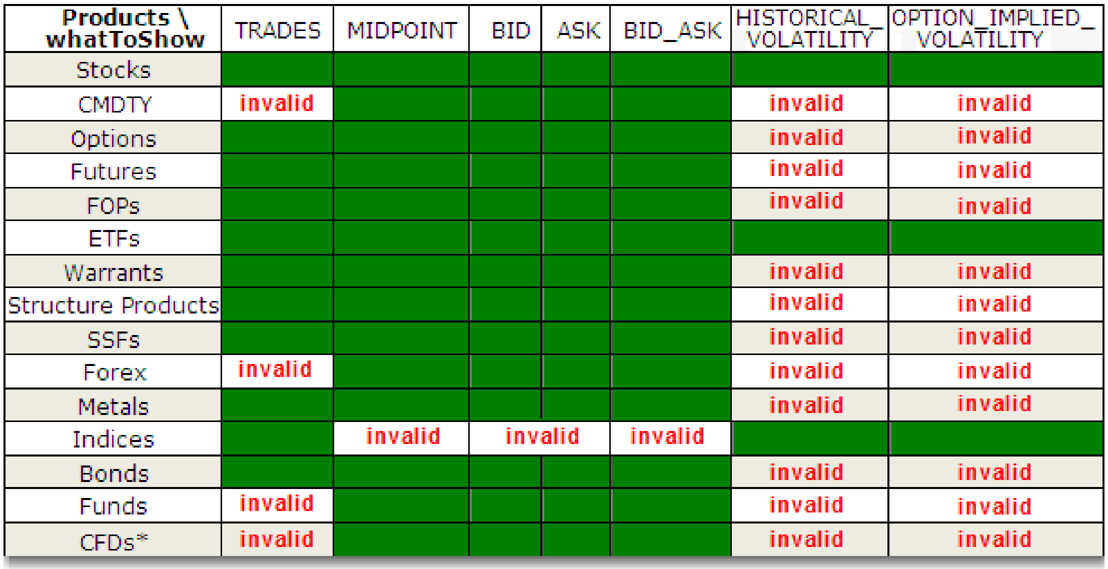
[53] http://interactivebrokers.github.io/tws-api/historical_bars.html#hd_what_to_show

[54] For Bid_Ask, the time-weighted average bid prices are returned in the `open` field, and the ask prices in the `close` field.

Note that if **IncludeExpired** is set to 1 (or true), the historic data on expired contracts is limited to the last year of the contract's life, and is initially only supported by IB for expired futures contracts (*IBC* imposes no limitation, but IB may indeed).

Also 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', 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.

Refer to IB's documentation[55] for the latest information on the allowed parameter values for historical data requests. Here is a table listing the allowed **WhatToShow** values for various **SecTypes**, valid as of 7/2018:[56]



Products \ whatToShow	TRADES	MIDPOINT	BID	ASK	BID_ASK	HISTORICAL_VOLATILITY	OPTION_IMPLIED_VOLATILITY
Stocks	🟩	🟩	🟩	🟩	🟩	🟩	🟩
CMDTY	invalid	🟩	🟩	🟩	🟩	invalid	invalid
Options	🟩	🟩	🟩	🟩	🟩	invalid	invalid
Futures	🟩	🟩	🟩	🟩	🟩	invalid	invalid
FOPs	🟩	🟩	🟩	🟩	🟩	invalid	invalid
ETFs	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Warrants	🟩	🟩	🟩	🟩	🟩	invalid	invalid
Structure Products	🟩	🟩	🟩	🟩	🟩	invalid	invalid
SSFs	🟩	🟩	🟩	🟩	🟩	invalid	invalid
Forex	invalid	🟩	🟩	🟩	🟩	invalid	invalid
Metals	🟩	🟩	🟩	🟩	🟩	invalid	invalid
Indices	🟩	invalid	invalid		invalid	🟩	🟩
Bonds	🟩	🟩	🟩	🟩	🟩	invalid	invalid
Funds	invalid	🟩	🟩	🟩	🟩	invalid	invalid
CFDs*	invalid	🟩	🟩	🟩	🟩	invalid	invalid

When no data is returned by IB, **IBC** will automatically try to resend the historical data request using **WhatToShow**='Trades' (for **SecType**='IND'), or **WhatToShow**='Midpoint' (for any other **SecType**), if these are different from the **WhatToShow** in the original request:

```
>> data = IBC('action','history', 'symbol','EUR', ...
              'localSymbol','EUR.USD', 'secType','cash', ...
              'exchange','idealpro', 'barSize','1 hour')
[API.msg2] Historical Market Data Service error message: No historical
market data for EUR/CASH@FXSUBPIP Last 3600 {786819168, 162}
No data returned from IB - retrying with WhatToShow='Midpoint'...
data =
     dateNum: [1×18 double]
    dateTime: {1×18 cell}
          ...
```

[55] http://interactivebrokers.github.io/tws-api/historical_limitations.html

[56] https://interactivebrokers.github.io/tws-api/historical_bars.html#available_products_hd

Also note that some exchanges return the requested historical data, but do not provide all of the historical data fields listed above. For example, with FOREX on IDEALPRO, 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 = IBC('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]
```

In this example, historical daily (**BarSize** = '1 day') data from the past 3 days was requested on 2012-02-13 (Monday). Data was received 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 within the same day, not at midnight.

It is possible to specify **BarSize** larger than duration. For example, on July 14, if we specify a duration of 3 weeks, and **BarSize**='1w', we'd get the results for all Fridays (=end of trading week) in the past 21 days. If we set **BarSize**='1m', we'd get two results: for June 30 (=end of last trading month) and July 14.

IB's historical data mechanism enables retrieving data as recent as a minute ago, or as old as a year (or more, 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.

One user has reported that in some cases IB returns empty data for historical index (**SecType**='Ind') queries. Restarting TWS/Gateway and re-querying when the exchange is active appears to solve this problem.

7 Streaming data

Streaming data is a near-real-time mechanism, where IB sends ongoing information to **IBC** 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. The request's ID (a scalar integer) is returned.
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 (a data struct), without stopping the background streaming.

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

```
>> reqId = IBC('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 **IBC** 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 **IBC**. A different buffer is maintained for each contract (or rather, combination of **LocalSymbol**, **SecType** and **Expiry**). 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**, [57] then up to the specified number of latest bid quotes will be stored (note: only bid quotes are counted here):

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

Note that using a large **QuotesBufferSize** increases memory usage, which could have an adverse effect if you use a very large buffer size (many thousands) and/or streaming for a large number of different securities.[58]

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

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 Matlab data struct similar to this:

```
>> dataStruct = IBC('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.[59] Specifying all the other parameters is normally unnecessary, since **IBC** 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 (scalar integer) for the original streaming request, the same ID that was returned by *IBC* in our initial request.
- `symbol`, `localSymbol` – the security whose data is being streamed.
- `isActive` – indicates whether quotes are currently streamed for this 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` – maximal number of streaming bid quotes requested for the security (=**QuotesNumber**). When `quotesReceived` >= `quotesToReceive`, streaming is stopped and `isActive` is set to false (0).[60]

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

[59] **IBC** versions since 2012-01-15 only need to use **LocalSymbol**; earlier versions of **IBC** 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.

[60] 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 a few additional real-time quotes may have arrived.

- `quotesBufferSize` – size of the data buffer (=**QuotesBufferSize**).
- `genericTickList` – **GenericTickList** as requested in the initial request.
- `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 data as `bidPriceTimestamp`. Future versions may possibly indicate the latest timestamp of any quote, not necessarily a bid.

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 = IBC('action','query', 'symbol','EUR', ...
                    'localSymbol','EUR.USD', 'currency','USD', ...
                    'secType','cash', 'exchange','idealpro', ...
                    'QuotesNumber',10, 'QuotesBufferSize',3);
% now run the following command at any time to get the latest 3 quotes:
>> dataStruct = IBC('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, resend the request with **QuotesNumber**=0. The request will return the `dataStruct` with the latest data accumulated up to that time.

The **ReconnectEvery** parameter can be used to bypass occasional problems 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 **IBC** 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 simulated timeline that illustrates the use of streaming data in *IBC*:

Time	Events so far	User command	Description

9:50:00	0	`IBC('action','query',`
		`'symbol','IBM',`
		`'QuotesNumber',100,`
		`'QuotesBufferSize',100);`
			Streaming data for IBM starts.
			Up to 100 events to accumulate.

9:50:10	23	`data = IBC('action','query',`
		`'symbol','IBM','QuotesNumber',-1)`
			Return the 23 accumulated quotes; background streaming continues.

9:50:20	42	`data = IBC('action','query',`
		`'symbol','IBM','QuotesNumber',-1)`
			Return the 42 accumulated quotes; background streaming continues.

9:50:30	57	`IBC('action','query',`
		`'symbol','IBM','QuotesNumber',80,`
		`'QuotesBufferSize',80);`
			Reduce max # of events 100→80.
			Only 57 events accumulated until now, so streaming continues.

9:50:40	65	`data = IBC('action','query',`
		`'symbol','IBM','QuotesNumber',-1)`
			Return the 65 accumulated quotes; background streaming continues.

9:50:50	72	`IBC('action','query',`
		`'symbol','IBM','QuotesNumber',0)`
			Reduce max # of events 80→0.
			72 events accumulated until now, so streaming stops immediately.

Here is a summary of the *IBC* 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	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.[61]
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 ("*Quote Booster*") from IB.
- IB's messages rate limitation (50/sec, see §3.1) does not directly affect streaming quotes, only messages sent to the IB server. There is no known IB limitation on streamed messages rate. However, a practical limitation is ~50-100 quotes/sec due to your client computer processing time.
- Streaming data retrieval is subject to the same pre-conditions as for retrieving the current live market data (see §5.1).

[61] https://interactivebrokers.github.io/tws-api/tick_types.html

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[62]. 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 *IBC*).

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 **CallbackRealtimeBar** parameter to set a dedicated Matlab callback function that will be invoked whenever a new bar is received (every 5 secs).[63]

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

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

[62] http://interactivebrokers.github.io/tws-api/realtime_bars.html

[63] 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 = IBC('action','realtime', 'symbol','AMZN', ...
                    'QuotesNumber',3, 'QuotesBufferSize',10)
reqId =
   345327051

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

>> dataStruct = IBC('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 =
     dateNum: [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 *IBC* 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 *IBC* 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.[64]
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.

[64] https://interactivebrokers.github.io/tws-api/tick_types.html

7.3 Streaming market depth (Level II) data

The streaming market depth mechanism [65] is also similar to streaming quotes in the sense that it enables the user to receive Level II information about a security every several seconds, until the specified **QuotesNumber** have been received. In fact, the only difference between streaming market depth data and streaming quotes data is that for market depth, the **NumberOfRows** parameter is set to an integer value between 2-5 (i.e., 2, 3, 4, or 5), and a slightly-different returned dataStruct:

```
>> reqId = IBC('action','query', 'symbol','EUR', ...
                    'localSymbol','EUR.USD', 'currency','USD', ...
                    'secType','cash', 'exchange','idealpro', ...
                    'NumberOfRows',3, 'QuotesNumber',1000)
reqId =
   464879608

>> dataStruct = IBC('action','query', 'localSymbol','EUR.USD', ...
                         'NumberOfRows',3, 'QuotesNumber',-1)
dataStruct =
                reqId: 464879608
              reqTime: '16-Dec-2014 14:46:47'
     lastEventDateNum: 735949.615954282
    lastEventDateTime: '16-Dec-2014 14:46:57'
               symbol: 'EUR'
          localSymbol: 'EUR.USD'
             isActive: 1
       quotesReceived: 362
      quotesToReceive: 1000
                  bid: [1x3 struct]
                  ask: [1x3 struct]
             contract: [1x1 com.ib.client.Contract]

>> dataStruct.bid(1)
ans =
          price: 1.2546
           size: 6560000
    marketMaker: ''
        dateNum: 735949.615954271
       dateTime: '16-Dec-2014 14:46:57'
```

Note that market-depth quotes are sent from the IB server at a much higher rate than streaming quotes. For EUR.USD at a specific date-time, there were 2-3 streaming quotes per second, compared to 30-50 market-depth updates per second.

As with streaming quotes, the streamed market-depth update events can be trapped in **IBC** using the **CallbackUpdateMktDepth** and **CallbackUpdateMktDepthL2** parameters. See §11.5 for a usage example of a continuously-updating order-book GUI, which uses these callbacks.

[65] See §5.2 above for a description of the market depth mechanism and its reported data fields.

8 Sending trade orders

8.1 General usage

Five order types are supported in **IBC**, which use the following values for *IBC*'s **Action** parameter: 'Buy', 'Sell', 'SShort', 'SLong', 'Close'. [66]

Several additional *IBC* 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 = IBC('action','BUY','symbol','GOOG','quantity',100,...
                   'type','LMT', 'limitPrice',600);
orderId = IBC('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 of Limit and limit price of US$600. *IBC* returns the corresponding orderId assigned by IB – we can use this orderId later to modify open orders, cancel open orders, or follow-up in TWS or in the trade logs.

Important: The IB server accepts up to 50 messages per second. If you exceed this rate, you will receive an error message from IB. This is important when submitting multiple orders to IB in a loop (baskets are not currently supported by the IB API).

IBC always returns an orderId (positive integer number) if the order is successfully created. This does **not** mean that the order is accepted: it may be rejected or held by the IB server or exchange. In such cases, a followup error message is sent from the IB server and appears as a red message in the Matlab console. For example:

```
[API.msg2] The following order "ID:662631663" size exceeds the Size
Limit of 500. Restriction is specified in Precautionary Settings of
Global Configuration/Presets. {662631703, 451}

[API.msg2] Order Message:
SELL 12 GOOG NASDAQ.NMS
Warning: your order will not be placed at the exchange until
2016-10-06 09:30:00 US/Eastern {662631843, 399}
```

The order's **Type** parameter is described in detail below (§8.3). In addition to specifying the **Symbol**, **Type**, **Quantity** and **LimitPrice**, several other parameters may need to be specified to fully describe the order.

All the order parameters listed below are optional, except for **Action** and **Quantity**. [67] Depending on the order **Type**, additional parameters may also be mandatory (e.g., **LimitPrice** and **AuxPrice**). Here is a summary of order parameters in *IBC*: [68]

[66] SShort is only relevant only for institutional accounts configured with Long/Short account segments or clear orders outside of IB, which need to distinguish between shorting/selling a position. In most acounts, orders are cleared by IB so you would only specify SELL for both shorting and selling (IB automatically infers the action type based on the currently-held position). SLong is available in specially-configured institutional accounts to indicate that long position not yet delivered is being sold.

[67] **Quantity** should only be omitted if **Action**='close' or **FAMethod**='PctChange' – see §8.2 and §8.5 below.

[68] Also see the corresponding API documentation: https://interactivebrokers.github.io/tws-api/classIBApi_1_1Order.html

Parameter	Type	Default	Description
Action	string	(none)	One of: 'Buy', 'Sell', 'SShort', 'SLong' [69] or 'Close' (see §8.2 below)
Quantity	number	0	Number of requested shares. Must be > 0.
Type	string	'LMT'	Refer to the order-types table in §8.3 below
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 (§8.4)
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) [70] • 'GAT' – Good After Time/date (uses **GoodAfterTime** parameter below) [71] • 'IOC' – Immediate or Cancel • 'FOK' – Fill or Kill[72] • 'OPG' – Open Price Guarantee[73] • 'AUC' – Auction, submitted at the Calculated Opening Price[74]
GoodTillDate	string	''	Format: 'YYYYMMDD hh:mm:ss TMZ' (TMZ is optional[75])
GoodAfterTime	string	''	Format: 'YYYYMMDD hh:mm:ss TMZ' (TMZ is optional[76])

[69] SShort is only relevant only for institutional accounts configured with Long/Short account segments or clear orders outside of IB, which need to distinguish between shorting/selling a position. In most acounts, orders are cleared by IB so you would only specify SELL for both shorting and selling (IB automatically infers the action type based on the currently-held position). SLong is available in specially-configured institutional accounts to indicate that long position not yet delivered is being sold.

[70] GTD requires enabling Advanced Time In Force Attributes in TWS / IB Gateway’s preferences: (http://interactivebrokers.com/en/software/webtrader/webtrader/orders/advanced%20time%20in%20force%20attributes.htm)

[71] GAT requires enabling Advanced Time In Force Attributes in the Preferences page of TWS / IB Gateway (http://interactivebrokers.com/en/software/webtrader/webtrader/orders/advanced%20time%20in%20force%20attributes.htm). For additional information see http://interactivebrokers.com/en/software/tws/usersguidebook/ordertypes/good_after_time.htm

[72] FOK requires the entire order to be filled, as opposed to IOC that allows a partial fill. For additional information on FOK see http://interactivebrokers.com/en/software/tws/usersguidebook/ordertypes/fill_or_kill.htm

[73] An OPG is used with a Limit order to indicate a Limit-on-Open order (http://interactivebrokers.com/en/software/tws/usersguidebook/ordertypes/limit-on-open.htm), or with a Market order to indicate a Market-on-Open order (http://interactivebrokers.com/en/software/tws/usersguidebook/ordertypes/market-on-open.htm)

[74] For additional information on AUC see http://interactivebrokers.com/en/software/tws/usersguidebook/ordertypes/auction.htm

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

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

Parameter	Type	Default	Description
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
Hold	integer or logical flag	0=false	• 0 or false: order is sent to IB immediately • 1 or true: order is prepared in *IBC* but not sent to IB. The user can them modify the order properties before sending it to IB. See §9.6 below for additional details.
Transmit	integer or logical flag	1=true	• 0 or false: order is sent to IB but waits in TWS until user clicks <Transmit> (§9.6) • 1 or true: order should immediately be sent by IB to the exchange for execution
WhatIf	integer or logical flag	0=false	• 0 or false: regular order sent to exchange • 1 or true: dummy order to compute margin impact (see §8.6 below for details)
AccountName	string	''	IB account used for this trade. Useful if you manage multiple accounts, else leave empty.
FAProfile	string	''	Financial Advisor profile for trades allocation (§8.5). Only relevant for Financial Advisor accounts, otherwise leave empty.
FAGroup	string	''	Financial Advisor account group for trades allocation (§8.5). Only relevant for Financial Advisor accounts, otherwise leave empty.
FAMethod	string	''	Method by which trades allocate within the stated **FAGroup** (§8.5). Only relevant for Financial Advisor accounts, else leave empty.
FAPercentage	number	0	Percentage of position change used when **FAMethod**='PctChange' (§8.5). Relevant for Financial Advisor accounts, else leave as-is.
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. Use this to modify open orders (see §10 below).
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 used in hedged- and bracket orders (see §9.3 below), but can also be used otherwise.

Parameter	Type	Default	Description

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.

BracketDelta	number	[]=empty	Price offset for stop-loss and take-profit bracket child orders (see §9.3 below). **BracketDelta** may be a single value or a [lowerDelta,upperDelta] pair of values > 0 Note: value(s) must be positive: - low bracket will use limitPrice – lowerDelta - high bracket will use limitPrice + upperDelta
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.[77]
OCAType	integer	2	One of (where allowed/applicable): • 1 = Cancel on fill with block • 2 = Reduce on fill with block • 3 = Reduce on fill without block
HedgeType	string	''	One of:[78] • 'D' – Delta (parent option, child stock) • 'B' – Beta • 'F' – FX • 'P' – Pair Relevant only for hedge orders, otherwise leave empty (or do not specify at all). Note: hedge orders must be child orders (**ParentId**>0) with **Quantity**=0.

HedgeParam	number	[]=empty	Beta = x for Beta hedge orders (0=unused);
			Ratio = y for Pair hedge orders.
			Relevant only for Beta/Pair hedge orders, otherwise leave empty (or do not specify).

[77] http://interactivebrokers.com/en/index.php?f=617

[78] http://interactivebrokers.github.io/tws-api/hedging.html, http://interactivebrokers.com/en/software/tws/attachedordertop.htm

Parameter	Type	Default	Description
TriggerMethod	integer	0	One of:[79] • 0=Default • 1=Double-Bid-Ask • 2=Last • 3=Double-Last • 4=Bid-Ask • 7=Last-or-Bid-Ask • 8=Mid-point
OrderRef	string	''	A comment that is attached to the order, displayable in TWS as an Order Attribute

8.2 Close orders

When setting the **Action** parameter to 'Close', *IBC* retrieves the current portfolio position for the specified **Symbol** or **LocalSymbol**, and issues a trade order that would liquidate this position.

For example, if we have 130 shares of GOOG in our portfolio, then the following two commands are equivalent (internally, the first command is automatically converted into the second command before being sent to IB):

```
orderId = IBC('action','CLOSE','symbol','GOOG',...
              'type','LMT', 'limitPrice',600);

orderId = IBC('action','SELL','symbol','GOOG','quantity',130,...
              'type','LMT', 'limitPrice',600);
```

The main benefit of using **Action**='Close' is that you do not need to know the exact number of shares in the portfolio. If *IBC* does not find the specified contract in the portfolio, then the command simply returns with an `orderId` of -1.

Naturally, when **Action**='Close', any user-specified **Quantity** value is ignored – the order quantity is determined based on the actual portfolio position.

Financial advisors should note that **Action**='Close' commands are not supported for multiple accounts at once, only for a single account at a time. If you try to issue the command for multiple accounts (as shown in §4.1, §4.2), then an error will be thrown asking you to specify the **AccountName** parameter to a single account. Alternatively, use the *PctChange* **FAMethod** to close the open positions (as shown in §8.5). If you only manage a single IB account, then the **AccountName** parameter is ignored and you do not need to worry about this limitation.

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

8.3 Order types

IB supports many order types. Some of these may not be available on your TWS and/or the requested exchange and security type.[80] Also, some order types are not supported by IB on paper-trading accounts, only live accounts.[81] You need to carefully ensure that the order type is accepted by IB before using it in *IBC*.

Here is the list of order types supported by *IBC*, which is a subset of the list in IB's documentation (if you encounter an order type that you need and is not on this list, try using it in *IBC* – perhaps it will indeed be accepted by the IB server):

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
	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).

[80] http://interactivebrokers.com/en/index.php?f=4985, http://interactivebrokers.github.io/tws-api/available_orders.html, http://interactivebrokers.com/en/software/tws/twsguide.htm#ordertypestop.htm

[81] http://interactivebrokers.com/en/software/am/am/manageaccount/paper_trading_limitations.htm

Class	Order type full name	Order type abbreviation	Description
Limit risk	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*) (see details and usage example in §8.4 below)
	Trailing Stop Limit	TRAIL LIMIT	A Trailing Stop Limit sell order sets the stop trigger price 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 (see usage example below). (*and vice versa for buy orders*) (see details and usage example in §8.4 below)

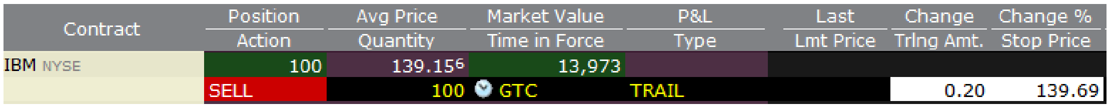
Class	Order type full name	Order type abbreviation	Description
Execution speed	Market	MKT	An order to buy (sell) a security at the offer (bid) price currently available in the marketplace. There is no guarantee that the order will fully or even partially execute at any specific price.
	Market-if-Touched	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. IB allows you to modify the cut-off, expiration times using Time in Force (**TIF**) and Expiration Date fields respectively.

8.4 Trail orders

Here's a usage example for sending a TRAIL order: [82] In this example, we previously purchased 100 shares of IBM at an average price of $139.156 and wish to lock-in a profit and limit our loss. We set a trailing stop order with the trailing amount $0.20 below the current market price of $139.71. To do this, create a sell order, with **Type**='TRAIL' and **AuxPrice**=0.20 (the trailing amount):

```
orderId = IBC('action','SELL', 'symbol','IBM', 'quantity',100,...
              'type','TRAIL', 'auxPrice',0.20);
```

The trigger (stop) price will follow (trail) market movements upwards, and remains stable when the market falls. The trigger (stop) price is initially set to $139.71 - $0.20 = $139.51, and rises with the market. When the market price reaches $139.89, the corresponding stop price is updated to $139.89 - $0.20 = $139.69. When the market price then falls to $139.73, the stop price remains stable at $139.69:



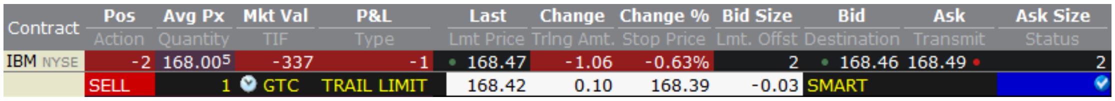
Contract	Position	Avg Price	Market Value	P&L	Last	Change	Change %
	Action	Quantity	Time in Force	Type	Lmt Price	Trlng Amt.	Stop Price
IBM NYSE	100	139.156	13,973				
	SELL	100	GTC	TRAIL		0.20	139.69

When the market price drops all the way to the stop price, the order is submitted as a Market order, which immediately fills (depending on market fluidity).

We can specify the trailing offset as either a fixed amount (**AuxPrice** parameter) or a percentage (**TrailingPercent** parameter), but not both.

As a related example, we can use a TRAIL LIMIT order: [83] Here we want a Limit (not Market) order when the market price drops to the trigger (stop) price. So, we provide a limit offset price in addition to trailing amount/%, using the **TrailStopPrice** parameter. In our example, when market (last) price was $168.38 we set **AuxPrice**= 0.10, **TrailStopPrice**=168.32, and **LimitPrice**=168.35, i.e. a limit offset of $-0.03:

```
orderId = IBC('action','SELL', 'symbol','IBM', 'quantity',1,...
              'type','TRAIL LIMIT', 'auxPrice',0.10, ...
              'TrailStopPrice',168.32, 'LimitPrice',168.50);
```



Contract	Pos	Avg Px	Mkt Val	P&L	Last	Change	Change %	Bid Size	Bid	Ask	Ask Size
	Action	Quantity	TIF	Type	Lmt Price	Trlng Amt.	Stop Price	Lmt. Offst	Destination	Transmit	Status
IBM NYSE	-2	168.005	-337	-1	● 168.47	-1.06	-0.63%	2	● 168.46	168.49 ●	2
	SELL	1	GTC	TRAIL LIMIT	168.42	0.10	168.39	-0.03	SMART		✓

As long as the market rises and last price >= **TrailStopPrice** + **AuxPrice**, both the trigger (stop) price and limit price will rise. Once the price drops to the latest stop price ($168.39, which is $0.10 below the highest market price up to now: $168.49), the order will change into a LMT order with a limit price of $168.39+$0.03=$168.42 and marked as triggered (Status field in TWS will change from ✓ to ★ ✓).

Note 1: IB's API changed the meaning of **TrailStopPrice** in 2016, so test carefully!
Note 2: specify either a fixed offset amount (in TWS config) or **LimitPrice**, not both.

[82] http://interactivebrokers.com/en/index.php?f=605

[83] http://interactivebrokers.com/en/index.php?f=606

8.5 Financial Advisor (multi-client) orders

Financial Advisor (FA, or “multi-client”) accounts in IB have the ability to manage multiple individual accounts under a single parent account. IB does not expose FA functionalities to individual-account holders – such users should skip this section.

When sending a trade order to an FA account, we need to tell IB which sub-account(s) should be affected by the order. The possible alternatives are:

- Execute the trade order in a specific sub-account.
- Execute the trade order in multiple sub-accounts, using an *Allocation Profile* (that was previously-defined in TWS[84]).
- Execute the trade order in multiple sub-accounts, using an *Account Group* (that was previously-defined in TWS[85]) and a specified *allocation method*.

We can also set-up a default allocation in TWS, [86] avoiding the need to specify the allocation separately for each trade order. The following discussion assumes that such a default allocation is not used, or that it is overridden on a per-trade basis.

For example, assume that our parent account is called DF1230 and it has three sub-accounts (DU1231, DU1232, and DU1233).

To send the trade to a specific account, simply state the **AccountName** parameter:

```
orderId = IBC('action','BUY', 'symbol','IBM', 'quantity',100,...
                   'type','MKT', 'AccountName','DU1232');
```

To send the trade to multiple specific sub-accounts using a predefined allocation profile, state the **FAProfile** parameter with the requested profile name:

```
orderId = IBC('action','BUY', 'symbol','IBM', 'quantity',100,...
                   'type','MKT', 'FAProfile','myProfile1');
```

To send the trade to multiple specific sub-accounts using a predefined accounts group, state the **FAGroup** and **FAMethod** parameters, and possibly also the **FAPercentage** parameter (only if **FAMethod** = 'PctChange'):

```
% Enter into position: allocate to sub-accounts based on their netliq
orderId = IBC('action','BUY', 'symbol','IBM', 'quantity',100,...
                   'type','MKT', 'FAGroup','EntryGroup', ...
                   'FAMethod','NetLiq');

% Exit position: each sub-account's position reduced by 100%
orderId = IBC('action','SELL', 'symbol','IBM', ... % no Quantity
                   'type','MKT', 'FAGroup','ExitGroup', ...
                   'FAMethod','PctChange', 'FAPercentage',-100);
```

[84] http://interactivebrokers.com/en/software/tws/usersguidebook/financialadvisors/create_a_share_allocation_profile.htm

[85] http://interactivebrokers.com/en/software/tws/usersguidebook/financialadvisors/create_an_account_group_for_share_allocation.htm

[86] http://interactivebrokers.com/en/software/tws/usersguidebook/financialadvisors/set_default_allocations.htm

Note how in this latest example, we've used the *NetLiq* method to enter into a position (split up amongst the sub-accounts defined in our EntryGroup, based on the accounts relative net liquidation values), but we exit the position using the *PctChange* method (i.e., sell securities such that the new position in each sub-account is -100% of its current position). This is a very typical entry/exit usage scenario.

The benefit of using *PctChange* to exit a position is that we do not need to calculate or even know the total **Quantity** nor the actual current position in each of the sub-accounts. We cannot use *NetLiq* to exit the position (as we have to enter it), since the different sub-accounts may possibly have a different NetLiq relative ratio between themselves, so the liquidation order would leave a few extra shares in some sub-acounts and a few missing (shorted) shares in the other sub-accounts.

Note that **FAMethod** only works with **FAGroup** and is a mandatory parameter when **FAGroup** is specified. In other words, we cannot specify **FAMethod** with **FAProfile**, nor specify **FAGroup** without a corresponding **FAMethod**.

When specifying **FAMethod**=*PctChange*, it is an error to specify the **Quantity**, since the quantity is automatically calculated by IB. Also, the trade order will only have an effect if the trade **Action** and the current total position would result in a trade having the same direction as the requested **FAPercentage**.

For example, if we currently have 20 shares of IBM in DU1231, 30 shares in DU1232 and 50 shares in DU1233 (i.e., a total of 100 shares), then the exit order above would result in a valid trade order that would sell 100 shares of IBM at MKT and reduce each of the sub-accounts' holdings to 0 shares (-100% of their current holdings).

On the other hand, if we used **Action**=BUY (rather than SELL), then the direction of the action and position (i.e., increase the position) would not match the direction of the requested **FAPercentage** (-100%, i.e. decrease position). The IB-calculated order size would be 0, the order would not execute, and IB will send us an error message:

```
orderId = IBC('action','BUY', 'symbol','IBM', ... % no Quantity
                  'type','MKT', 'FAGroup','ExitGroup', ...
                  'FAMethod','PctChange', 'FAPercentage',-100);
```

```
[API.msg2] The order size cannot be zero. {640996954, 434}
```

Action	**FAPercentage**	Long position	Short position
Buy	Positive (>0)	Trade **in**creases position	Error (no trade)
	Negative (<0)	Error (no trade)	Trade **de**creases position
Sell	Positive (>0)	Error (no trade)	Trade **in**creases position
	Negative (<0)	Trade **de**creases position	Error (no trade)

If you state a **FAMethod** that is not officially supported by IB, **IBC** issues a warning but sends the request to IB anyway, in the hope that the method is supported after all. If IB does not support it, the request is ignored and IB sends an error message:

```
IBC('action','BUY', 'symbol','IBM', 'quantity',100,...
          'type','MKT', 'FAGroup','EntryGroup', 'FAMethod','XYZ');

Warning: FAMethod 'XYZ' may possibly not be supported by IB
(Type "warning off YMA:IBC:FAMethod" to suppress this warning.)

[API.msg2] Order rejected - reason: Invalid value in field # 6159
{640973648, 201}
```

Note that **Action**='Close' commands (§8.2) are not supported for multiple accounts at once, only for a single account at a time. If you try to issue the command for multiple accounts (as shown in §4.1, §4.2), then an error will be thrown, asking you to specify the **AccountName** parameter to a single account. Alternatively, use the *PctChange* method to close the open positions as shown above.

The following parameters affect Financial Advisor (FA) trade orders:

Parameter	Type	Default	Description
AccountName	String	''	The specific IB account used for this trade. Useful when you manage multiple IB accounts, otherwise leave empty.
FAProfile	String	''	Financial Advisor profile for trade allocation. Only relevant for Financial Advisor (multi-client) accounts, otherwise leave empty.
FAGroup	String	''	Financial Advisor account group for trade allocation. Only relevant for Financial Advisor (multi-client) accounts, otherwise leave empty.
FAMethod	String	''	Method by which trades will be allocated within the stated **FAGroup**. Only relevant for Financial Advisor accounts, otherwise leave empty. IB officially supports the following methods: • NetLiq • EqualQuantity • AvailableEquity • PctChange (requires **FAPercentage**)
FAPercentage	Number	0	Percentage of position change used when **FAMethod** = 'PctChange'. Only relevant for Financial Advisor accounts, otherwise leave as-is. When this **FAPercentage** parameter is specified, the **Quantity** parameter may NOT be specified.

8.6 Potential impact of an order (“what-if”)

It is possible to investigate the margin and commission impact of a potential order using the **WhatIf** parameter. By default this parameter has a value of false, meaning that a regular trade order will be sent to the markets. But if you set **WhatIf** to a value of `true` (or 1) then IB will not really sent the trade order to the market – IB will just calculate and return the potential impact of the specified order on your account, including the updated account margin and the trade’s estimated commission:

```
>> data = IBC('action','buy', 'symbol','IBM', 'type','MKT', ...
                   'quantity',1, 'whatif',true)
data =
                m_status: 'PreSubmitted'
            m_initMargin: 38.59
            m_maintMargin: 38.59
        m_equityWithLoan: 1004892.14
            m_commission: 1.79769313486232e+308
         m_minCommission: 0.34685725
         m_maxCommission: 0.35345725
    m_commissionCurrency: 'USD'
           m_warningText: []
```

Note that the returned data is a Matlab struct, not a scalar orderId as for regular trade orders.

Also note that in the returned data struct, the value `1.79769313486232e+308` is simply IB’s way of indicating an undefined/uninitialized value.

9 Specialized trade orders

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

9.1 VWAP (best-effort) orders

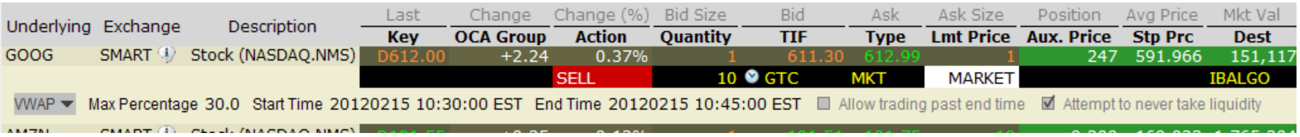
When the order **Type** is 'VWAP' (the best-effort type, since the guaranteed type has Type='GuaranteedVWAP'), IB treats the order as a Market order with a VWAP algo strategy.[87] *IBC* enables specifying the algo strategy's properties, as follows:

Parameter	Data type	Default	Description
Type	String	'LMT'	Set to 'VWAP' for this IBAlgo type
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 **EndTime** 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.
SpeedUp	integer or logical flag	0=false	If true, compensates for the decreased fill rate due to presence of limit price.
MonetaryValue	number	0	Cash quantity

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

```
orderId = IBC('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);
```

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



Underlying	Exchange	Description	Last	Change	Change (%)	Bid Size	Bid	Ask	Ask Size	Position	Avg Price	Mkt Val
			Key	OCA Group	Action	Quantity	TIF	Type	Lmt Price	Aux. Price	Stp Prc	Dest
GOOG	SMART	Stock (NASDAQ.NMS)	D612.00	+2.24	0.37%	1	611.30	612.99	1	247	591.966	151,117
					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

[87] http://interactivebrokers.github.io/tws-api/ibalgos.html#vwap, http://interactivebrokers.com/en/software/tws/usersguidebook/algos/vwap.htm

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. **EndTime** 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 **AllowPastEndTime**=1. If an EndTime is specified, then set **AllowPastEndTime**=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 the validity 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 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:

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

Setting the **NoTakeLiq** parameter value to true (or 1) 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.

VWAP orders are treated as LMT orders so the **LimitPrice** parameter is mandatory.[88]

Note: IB only enables VWAP algo orders for US equities on live accounts (i.e., not on paper-trading accounts[89]).

[88] IBC versions prior to 1.92 (July 14, 2017) used MKT orders for the VWAP algo

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.[90] *IBC* enables specifying the algo strategy’s properties, as follows:

Parameter	Data type	Default	Description
Type	string	'LMT'	Set to 'TWAP' for this IBAlgo type
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 EndTime 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 = IBC('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.

Note: IB only enables TWAP algo orders for US equities.

[89] http://interactivebrokers.com/en/software/am/am/manageaccount/paper_trading_limitations.htm

[90] http://interactivebrokers.github.io/tws-api/ibalgos.html#twap, http://interactivebrokers.com/en/software/tws/usersguidebook/algos/twap.htm

9.3 Bracket (child) 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.[91] 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 **IBC**, 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 a 2-value array [lowerDelta,upperDelta], then lowerDelta is used as the offset for the lower child, and upperDelta is used for the upper child. The corresponding child order limits will be set to **LimitPrice**-lowerDelta and **LimitPrice**+upperDelta, respectively.

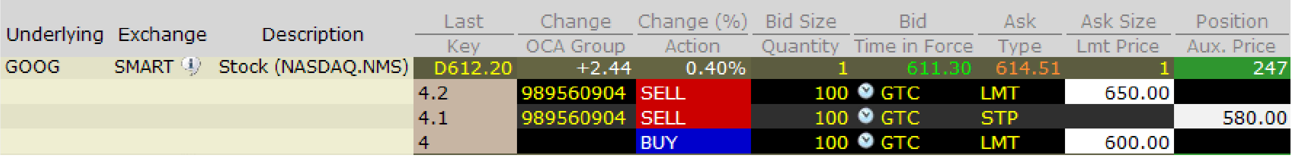
- If **BracketDelta** is a single (scalar) value, then this value is used as offset for both child orders: **LimitPrice**-offset and **LimitPrice**+offset, respectively.

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

Will result in the following situation in IB:



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

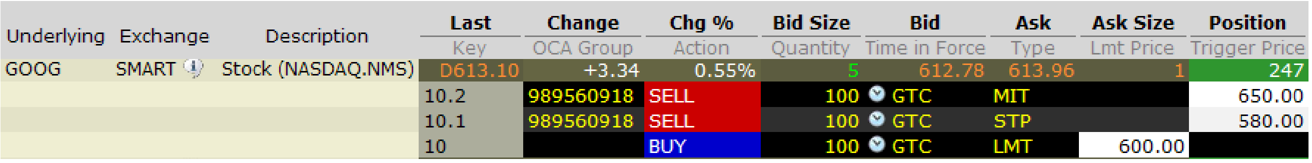
In this screenshot, notice that the parent order is shown as active (blue; IB status: "*Order is being held and monitored*") at the bottom. This order has a Last-Key value of "4" and is a simple Buy at Limit 600 order.

[91] http://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 have 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 (4.2, 4.1 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 = IBC('action','BUY', 'symbol','GOOG', ...
                    'quantity',100, 'type','LMT', ...
                    'limitPrice',600, 'BracketDelta',[20,50], ...
                    'BracketTypes',{'STP','MIT'});
```



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

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

The following parameters specifically affect bracket orders:

Parameter	Data type	Default	Description
BracketDelta	number	[]=empty	Price offset for stop-loss and take-profit bracket child orders. 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.

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).

IBC 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, *IBC*'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 *IBC*'s automated orders could be useful.

The following parameters affect automated orders in *IBC*:

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** only increases; if market price decreases, **LimitPrice** remains as-is • -1: **LimitPrice** only decreases; 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).

IBC 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 *IBC*'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=IBC('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=IBC('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=33, then again between t=36 and t=51, 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 *IBC*, specify the leg parameters (**Symbol**, **LocalSymbol**, **SecType**, **Exchange**, **Currency**, **Multiplier**, **Expiry**, **Strike**, and **Right**) in a cell array wherever the different legs have different values. In addition, you must specify the **ComboActions** parameter:

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

Alternatively, 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 = IBC('action','buy', 'exchange',{'CFE','CFE'}, ...
                   'quantity',1, 'SecType',{'FUT','FUT'}, ...
                   'LocalSymbol',{'VXZ2','VXX2'}, ...
                   'ComboActions',{'Buy','Sell'})
```

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

```
data = IBC('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
               high: -1
          lastPrice: -1
             volume: -1
               tick: 0.2500
           contract: [1x1 struct]
    contractDetails: [1x2 struct]
```

Only instantaneous market bid/ask data is reliably returned for combo queries – the open, close, low, high, lastPrice and volume fields are often returned empty (-1).

IB may rejects combo requests (query/trade), due to a variety of possible reasons:

1. IB only supports combos for a small subset of securities – generally speaking, US options and futures. For example, Forex is NOT supported as of 2016.
2. IB will reject a combo that has been incorrectly configured (see details below)
3. IB will reject a combo if you are not subscribed for real-time quotes for any of its legs.
4. IB does not support combos on the demo account, only on live and paper-trade accounts.[92] The availability of combo functionality may depend on your IB account's subscription plan and the specific combo that you try to use.
5. Some combo queries can only be received using streaming quotes (§7.1), but not snapshot quotes (§5.1), due to an IB server bug/limitation. A workaround for this limitation is included in **IBC** since version 1.97.

In all such cases, the query will return empty (-1) data in all data fields, including bidPrice/askPrice (for queries); or the command will simply be ignored by IB (for trade orders).

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 one or more legs is incorrectly configured or not supported or not subscribed for real-time data; perhaps the market is closed; etc. Contact IB to check your specific case.

When specifying combo legs, you can specify the optional **ComboRatios** parameter, as an array of positive values that shall be used to determine the relative weights of the legs. *IBC* uses default ratios of [1,1], i.e. the same ratio for all legs.

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 example, the **ComboRatios** for the ZN/ZT spread (10-vs-2-year US Treasury-Notes) must be set to 1:2 since the ECBOT exchange does not currently (1/1/2016) support any other ratio. This ratio changes over time: the ratio was 1:2 in early 2013, then changed to 3:5, then 1:2 again.[93] 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}
```

In cases where you cannot figure out the exact set of parameters for a combo, it might help to try to create the combo directly in TWS: If the combo is supported by TWS then it might also be available to the API (and *IBC*). But if the combo is not supported by TWS then it will also certainly not work in *IBC*.

[92] https://quant.stackexchange.com/questions/8744/what-is-the-difference-between-the-interactive-brokers-demo-account-and-a-personal-paper-trader-account; http://interactivebrokers.com/en/software/am/am/manageaccount/paper_trading_limitations.htm

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

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, the ZN/ZT spread has the internal symbol ‘TUT’:[94]

```
IBC('action','query', 'SecType','FUT', 'exchange','ECBOT', ...
         'ComboBagSymbol','TUT', ...  % the spread's symbol is TUT
         'LocalSymbol',{'ZN    MAR 16','ZT    MAR 16'}, ...
         'ComboActions',{'Sell','Buy'}, 'ComboRatios',[1,2])
ans =
              reqId: 576662704
            reqTime: '24-Dec-2015 05:18:49'
           dataTime: '24-Dec-2015 05:18:53'
      dataTimestamp: 7.3632e+05
      lastEventTime: 7.3632e+05
             ticker: ''
           bidPrice: -0.0078
           askPrice: 0.0078
               open: -1
              close: 0
                low: 0.0078
               high: 0.0078
          lastPrice: 0.0078
             volume: -1
             halted: 0
               tick: 0.0078
           contract: [1x1 struct]  <= Note: leg info in .m_comboLegs
    contractDetails: [1x2 struct]  <= Note: contractDetails for 2 legs
            bidSize: 592
            askSize: 25
           lastSize: 2
      lastTimestamp: '1450947638'
```

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:

```
IBC('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',[1,2], ...
         'QuotesNumber',2);
pause(1.5); % wait a bit for data to be received from IB server
data = IBC('action','query', 'QuotesNumber',-1, ...
                'LocalSymbol',{'ZN    MAR 13','ZT    MAR 13'});
```

When specifying the spread’s **LocalSymbol**, be careful to enter all the spaces. For example, the ZN **LocalSymbol** has 4 spaces between “ZN” and “MAR”. IB is very

[94] 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

sensitive about this: if you specify a **LocalSymbol** that is even slightly incorrect, IB will complain that it cannot find the specified contract. See §14.2 for additional details.

To complete the picture, here's an example order to purchase a bear spread for 9/2018 E-mini S&P 500 Future Options (**SecType**='FOP'; note the negative **LimitPrice**):

```
    orderId = IBC('action','buy', 'exchange','GLOBEX', 'quantity',1,...
                       'SecType','FOP', 'type','LMT', 'limitPrice',-4, ...
                       'symbol','ES', 'expiry',201809, 'right','Call', ...
                       'strike',[2720,2730], 'ComboActions',{'Sell','Buy'})
```

or alternatively:

```
    orderId = IBC('action','buy', 'exchange','GLOBEX', 'quantity',1,...
                       'SecType','FOP', 'type','LMT', 'limitPrice',-4, ...
                       'localSymbol',  {'ESU8 C2720', 'ESU8 C2730'}, ...
                       'ComboActions',{'Sell','Buy'})
```

The following parameters affect combo orders in *IBC*:

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. If left empty, IB tries to infer it from the other parameters.
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.
Multiplier	number	[]	The contract multiplier (for options)
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 – only 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, *IBC* will use the last leg's **LocalSymbol** and **Symbol** for the parent bag contract.

9.6 Setting special order attributes

Most of the important order parameters that are supported by IB are also supported as *IBC* 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 [95]), or to specify the All-or-None flag (via the order object's m_allOrNone property[96]).

These properties are not available as *IBC* parameters, but they can still be specified in **IBC** using the `ibConnectionObject` Java object returned by *IBC* 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 *IBC*'s **Hold** parameter (see §8.1) 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 – *IBC* 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 usage example:[97]

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

% Modify some contract properties
contract.m_secIdType = 'ISIN';
contract.m_secId = 'US0378331005';  % =Apple Inc.
contract.m_multiplier = '100';      % only relevant for option/future

% Modify some order properties
order.m_clearingIntent = 'Away';    % Possible values: IB, Away, PTA
order.m_settlingFirm = 'CSBLO';     % =Credit Suisse Securities Europe
order.m_allOrNone = true;           % set the order to be All-or-None
order.m_sweepToFill = true;         % set the order to be Sweep-to-Fill
order.m_orderRef = 'abra kadabra';  % internal trading note

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

Note that the contract and order objects are only returned from *IBC* for trading orders (i.e., **Action** = 'Buy', 'Sell', 'SShort' or 'SLong'), but not other *IBC* actions.

[95] https://interactivebrokers.com/php/whiteLabel/Interoperability/Socket_Client_Java/java_properties.htm

[96] https://interactivebrokers.com/php/whiteLabel/Interoperability/Socket_Client_Java/java_properties.htm

[97] Note that many properties can also be set via direct *IBC* parameters e.g. **SecId, SecIdType**, **Multiplier, OrderRef**

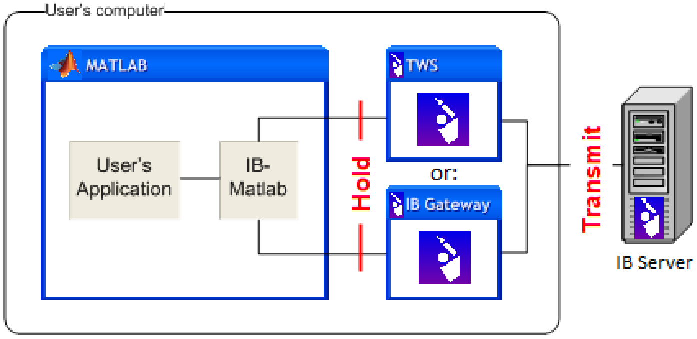
Some additional order fields that can be set in this manner include:

- `m_hidden` – true for a hidden order routed via the INet (Island) exchange[98]
- `m_displaySize` –integer >0 for an Iceberg order[99]
- `m_volatility` – value >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 listed in IB's API documentation[100]

When changing an order immediately following creation, IB might reject the request. In such cases, adding a short `pause(0.5)` normally solves the problem:

```
[API.msg2] Unable to modify this order as it is still being processed
```

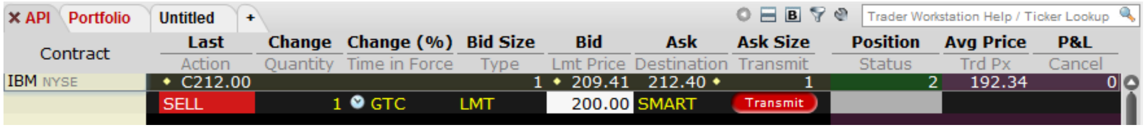
The **Hold** and **Transmit** parameters should not be confused: **Hold** delays the order in *IBC* (you *will not* see it in TWS); **Transmit** delays the order within *TWS*:



Using **Transmit**, orders can be sent to TWS and delayed (not sent for execution), until the user clicks the <Transmit> (or <Cancel>) button. Prepare such orders as follows:

```
IBC('action','BUY', 'Transmit',false, ...);
```

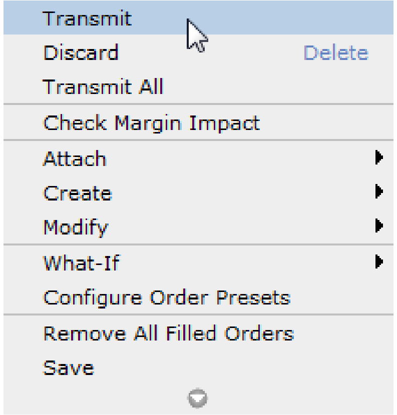
This will create the order in TWS without transmitting it. You will see the order in TWS's API tab with a button to transmit:



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, as seen on the right here:

While the order is waiting in TWS for transmission, its attributes can be modified, either directly in TWS, or programmatically (see §10).



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

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

[100] http://interactivebrokers.com/php/whiteLabel/Interoperability/Socket_Client_Java/java_properties.htm

9.7 Exercising and lapsing options

To exercise or lapse an option, use **Action**='exercise' or 'lapse' (respectively). You must specify the quantity of contracts and the exchange (IB's SMART exchange cannot be used). [101] You can also indicate whether to override IB's default handling action (exercise in-the-money options only). For example:

```
orderId = IBC('action','exercise', 'symbol','GOOG', ...
              'secType','OPT', 'expiry','201509', ...
              'multiplier',100, 'strike',600, 'right','C', ...
              'quantity',5, 'exchange','AMEX', 'override',true)
```

Assuming that the information is correct and that I have 5 unlapsed GOOG 9/2015 Call-600 options in my portfolio, then these 5 options will be exercised and turn into 500 shares (5 options * 100 multiplier) of the underlying GOOG, at USD 600 each.

At the time of this writing, GOOG trades at USD 542.34, so the exercise is not in the money and would be rejected if I had not stated **Override**=true. Because of the override the exercise order is executed at a nominal loss of USD 57.66 (=600-542.34) per share (excluding commissions).

If the option is not in-the-money and you try to exercise without specifying the **Override** parameter (or if you set the **Override** value to the default=false), you will receive an error from IB:

```
[API.msg2] Error processing request: Exercise ignored because option
is not in-the-money. {498825899, 322}
```

If the options do not exist in your portfolio you will receive a different error message:

```
[API.msg2] Error processing request: No unlapsed position exists in
this option in account DU123456. {498752361, 322}
```

If you have several IB accounts, then the **AccountName** parameter must be specified, otherwise you will receive yet a different error message:

```
[API.msg2] Error validating request:-'kd': cause - The account code is
required for this operation. {498752362, 321}
```

You can only lapse an option on its last trading day. If you try to lapse it on a different date, you will receive two separate error messages from IB:

```
[API.msg2] Order rejected - reason: trade date must match last trade
date of the contract {498825901, 201}

[API.msg2] Error processing request: Exercise/Lapse failed due to
server rejection {498825901, 322}
```

Finally, as of the time of this writing, IB only supports exercising/lapsing options, not FOP (future-on-option) or warrants. Customers wishing to exercise or lapse such contracts must submit a manual request ticket to IB.

[101] http://interactivebrokers.github.io/tws-api/classIBApi_1_1EClient.html#aad70a7b82ad3b5e7ae3e9f0b98dc2a5b

The following parameters affect exercising/lapsing options in *IBC*:

Parameter	Data type	Default	Description
Action	string	(none)	Either 'exercise' or 'lapse'.
Symbol	string	(none)	The symbol of the underlying asset.
LocalSymbol	string	''	The local exchange symbol of the option contract. When left empty, IB infers it from **Symbol** and the other properties.
SecType	string	(none)	Needs to be 'OPT'. IB does not currently allow exercising any other **SecType**.
Exchange	string	(none)	The exchange that should process the request – **cannot** be set to 'SMART'.
Currency	string	'USD'	The currency for the option contract.
Multiplier	number	[]	The option contract multiplier.
Expiry	string	''	'YYYYMM' or 'YYYYMMDD' format.
Strike	number	0.0	The strike price of the option contract.
Right	string	''	One of: ‘P’, ‘PUT’, ‘C’, ‘CALL’.
Quantity	integer	0	Number of contracts to exercise or lapse.
Override	integer or logical flag	0=false	• 0 or false: use default action (exercise in-the-money options only) • 1 or true: override the default action
AccountName	string	''	The specific IB account ID to use. Useful when you handle multiple IB accounts, otherwise leave empty.

9.8 Algorithmic trading orders

In addition to VWAP (§9.1) and TWAP (§9.2), **IBC** supports multiple algo-trading strategies, provided by both IB ("IBAlgo") and 3rd parties. Some important IBAlgos (Arrival Price,[102] Close Price,[103] Dark Ice,[104] Percentage of Volume,[105] Balance Impact/Risk,[106] Minimize Impact[107]) have dedicated *IBC* convenience parameters; numerous other algos can be specified using a pair of generic parameters.

Algo properties are specified in *IBC* as follows (Kind: D=dedicated; G=generic):

Parameter	Kind	Data type	Default	Description
Type	D	string	'LMT'	Set to one of the following: • 'VWAP' (see §9.1) • 'TWAP' (see §9.2) • 'ArrivalPx' • 'ClosePx' • 'DarkIce' • 'PctVol' • 'BalanceImpactRisk' • 'MinImpact'
MaxPctVol	D	number	0.1=10%	Max % participation of average daily volume up to 0.5 (=50%).
PctVol	D	number	0.1=10%	Target % participation of avg daily volume up to 0.5 (=50%).
StartTime	D	string	'9:00:00 EST'	Format: 'YYYYMMDD hh:mm:ss TMZ' (TMZ optional)
EndTime	D	string	'16:00:00 EST'	(*same as StartTime above*)
AllowPastEndTime	D	integer or logical flag	1=true	If true, allow the algo to continue to work past the specified **EndTime** if the full quantity has not been filled.
NoTakeLiq	D	integer or logical flag	0=false	If true, avoid hitting the bid or lifting the offer, if possible.

[102] http://interactivebrokers.github.io/tws-api/ibalgos.html#arrivalprice, http://interactivebrokers.com/en/index.php?f=1122, http://interactivebrokers.com/en/software/tws/usersguidebook/algos/arrival_price.htm

[103] http://interactivebrokers.github.io/tws-api/ibalgos.html#closepx, http://interactivebrokers.com/en/software/tws/usersguidebook/algos/closeprice.htm

[104] http://interactivebrokers.github.io/tws-api/ibalgos.html#darkice, http://interactivebrokers.com/en/software/tws/usersguidebook/algos/dark_ice.htm

[105] http://interactivebrokers.github.io/tws-api/ibalgos.html#pctvol, http://interactivebrokers.com/en/software/tws/usersguidebook/algos/percentage_of_volume_strategy.htm

[106] http://interactivebrokers.github.io/tws-api/ibalgos.html#balanceimpact, http://interactivebrokers.com/en/software/tws/usersguidebook/algos/balance_impact_and_risk.htm

[107] http://interactivebrokers.github.io/tws-api/ibalgos.html#minimpact, http://interactivebrokers.com/en/software/tws/usersguidebook/algos/minimize_impact.htm

Parameter	Kind	Data type	Default	Description
SpeedUp	D	integer or logical flag	0=false	If true, compensate for decreased fill rate due to a limit price. Only relevant for VWAP algo.
RiskAversion	D	string	‘Neutral’	One of: • 'Neutral' (default) • 'Get Done' • 'Aggressive' • 'Passive'
ForceCompletion	D	integer or logical flag	0=false	If true, attempt completion by end of day.
DisplaySize	D	integer	1	The order quantity (size) you want to display to the market. The algo will randomize the size by 50% on either side. Only relevant for DarkIce algo.
MonetaryValue	D	number	0	Cash quantity
AlgoStrategy	G	string	''	Any algo name listed in https://interactivebrokers.github.io/tws-api/algos.html
AlgoParams	G	cell-array	{}	Cell array of name,value pairs. Example: `{'MaxPctVol',0.25, 'RiskAversion','Aggressive'}`

Note: **StartTime**, **EndTime**, **AllowPastEndTime**, **NoTakeLiq**, **SpeedUp** and **MonetaryValue** were described in §9.1.

Only a few important IBAlgos have dedicated IBC parameters, listed above. All other strategies/algos are supported by using the **AlgoStrategy** and **AlgoParams** parameters, which cover the dedicated convenience parameters as well as many others.

Here is an example for an Arrival Price order using dedicated convenience parameters:

```
orderId = IBC('action','BUY', 'symbol','GOOG', 'quantity',10, ...
                   'TIF','Day', 'limitPrice',600, 'type','ArrivalPx',...
                   'MaxPctVol',0.01, 'RiskAversion','Passive', ...
                   'StartTime','20120215 10:30:00 EST', ...
                   'EndTime',  '10:45:00 EST', ...
                   'ForceCompletion',true, 'AllowPastEndTime',false);
```

And the same Arrival Price order using the generic **AlgoStrategy** and **AlgoParams**:

```
algoParams = {'maxPctVol',0.01, 'riskAversion','Passive', ...
              'startTime','20120215 10:30:00 EST', ...
              'endTime',  '10:45:00 EST', ...
              'forceCompletion',true, 'allowPastEndTime',false};
orderId = IBC('action','BUY', 'symbol','GOOG', 'quantity',10,...
                   'TIF','Day', 'limitPrice',600, ...
                   'algoStrategy','ArrivalPx', 'algoParams',algoParams);
```

IB regularly adds/modifies algo strategies and their corresponding parameters. Some algo properties (parameters) are only relevant to some algos but not others, and this list is also dynamic. For an up to date listing of the available algos and parameters, visit https://interactivebrokers.github.io/tws-api/algos.html.

IB only enables algo strategy orders for a subset of security types and exchanges. For example, as of December 2019 IBAlgos are limited to US stocks, while QBAlgos are limited to futures. Refer to the specific algo's documentation for details.

IB supports some algos only in TWS, not via the API. For example, as of December 2019, IB officially supports Fox River algos only in TWS, not the API.[108] If you specify such algos in *IBC*, IB may possibly reject the requested order. When IB adds any new algo provider, algo strategy and/or algo parameter, you can immediately use them in *IBC* via the **AlgoStrategy**, **AlgoParams** parameters.

As with VWAP and TWAP, IB automatically routes all IBAlgo trades to its internal servers (IBALGO), ignoring the specified **Exchange**. In contrast, all 3rd-party (non-IB) algos require routing the order through the corresponding 3rd-party algo servers: CSFB (Credit-Suisse First Boston) algos[109] require setting **Exchange**='CSFBALGO'; Jefferies algos[110] require **Exchange**='JEFFALGO'; and QB (Quantitative Brokers)[111] algos require **Exchange**='QBALGO'. Here is an example of a CSFB 'Inline' algo order:

```
algoParams = {'StartTime','20120215 10:30:00 EST', ...
              'EndTime',  '10:45:00 EST', ...
              'ExecStyle','Patient', 'Auction','Default',...
              'MinPercent',10, 'MaxPercent',20, 'DisplaySize',100,...
              'BlockFinder',false, 'BlockPrice',40,
              'MinBlockSize',100, 'MaxBlockSize',100, 'IWouldPrice',35};
orderId = IBC('action','BUY', 'symbol','GOOG', 'quantity',10,...
                   'Exchange','CSFBALGO', ...  % note the Exchange
                   'algoStrategy','Inline', 'algoParams',algoParams);
```

Additional notes:

- As with standard LMT orders, some algo orders are not guaranteed to execute.
- Many algos are only available in the live account, and cannot be tested in a paper-trading account. None of the algos are available in IB's Demo account.
- IBAlgo orders cannot use the default **TIF** value of 'GTC' – use 'Day' instead.
- IBAlgo orders are treated as LMT orders, so you must specify the **LimitPrice** parameter in all IBAlgo orders.[112] For better control over the order, avoid using the dedicated convenience algo parameters; use **AlgoStrategy** and **AlgoParams** instead. These do not override any order or contract parameter, so you can set (for example) **Type**='MKT' if you wish.

[108] https://interactivebrokers.com/en/software/tws/twsguide.htm#algostop.htm%3FTocPath%3DAlgos%7C 0; https://interactivebrokers.com/en/index.php?f=4985#thirdy-party-algos

[109] https://interactivebrokers.github.io/tws-api/csfb.html

[110] https://interactivebrokers.github.io/tws-api/jefferies.html

[111] https://interactivebrokers.github.io/tws-api/qbalgos.html

[112] IBC versions prior to 1.92 (July 14, 2017) used MKT orders for the VWAP algo

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 = IBC('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 parent order and two open bracket child-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:[113]

- `orderId` – this is the ID returned by *IBC* 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.[114]
- `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.

[113] http://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html#a17f2a02d6449710b6394d0266a353313

[114] http://interactivebrokers.github.io/tws-api/classIBApi_1_1OrderState.html

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: 1.79769313486232e+308
           m_maintMargin: 1.79769313486232e+308
        m_equityWithLoan: 1.79769313486232e+308
            m_commission: 1.79769313486232e+308
         m_minCommission: 1.79769313486232e+308
         m_maxCommission: 1.79769313486232e+308
    m_commissionCurrency: 'USD'
           m_warningText: []
```

Note: `1.79769313486232e+308` is IB's value for uninitialized data; some fields had values as strings (e.g. '1.797E308') in *IBC* versions older than 1.93 (10/2017).

Note: IB warns[115] 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 = IBC('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 = IBC('action','query', 'type','open', 'symbol','goog')
```

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

```
>> data = IBC('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 = IBC('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 **IBC ClientID**. Trades placed directly in TWS, or via another API client that connects to TWS, or by another **IBC** connection session with a different **ClientID**, are not normally accessible. If this limitation affects your work, use a static **ClientID** of 0, thereby enabling access to all open orders placed by any **IBC** session (since they will all have the same **ClientID**=0) as well as directly on TWS (which uses the same **ClientID**=0).[116] See §13 for additional details on **ClientID**.

[115] http://interactivebrokers.github.io/tws-api/order_submission.html#order_status

[116] http://interactivebrokers.github.io/tws-api/open_orders.html

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 *IBC* (and IB) to modify that specific order, rather than to create a new order:

```
[orderId,ibConnectionObject]=IBC('action','BUY','symbol','GOOG',...
                         'quantity',100,'type','LMT','limitPrice',600);

% Let some time pass...

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

      % Send the trade with modified parameters
      IBC('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(IBC('action','query','type','open','OrderId',orderId))

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

end
```

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

```
data = IBC('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.

Alternatively, 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] = IBC('action','BUY', ...);

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

11 Processing IB events

11.1 Processing events in IBC

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.[117]

Matlab has built-in support for asynchronous events, called *callbacks* in Matlab jargon.[118] Whereas Matlab callbacks are normally used in conjunction with Graphical User Interfaces (GUI), they can also be used with **IBC**, which automatically converts all the Java events received from IB into Matlab callbacks.

There are two types of callbacks that you can use in **IBC**:

- Generic callback – this is a catch-all callback function that is triggered upon any IB event. Within this callback, you would need to write some code to distinguish between the different event types in order to process the events' data. A skeleton for this is given below. The parameter controlling this callback in *IBC* 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 *IBC* 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,[119] so IB's event name is misleading in this specific case).

[117] https://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html

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

[119] http://interactivebrokers.github.io/tws-api/error_handling.html

When you specify any callback function to *IBC*, 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 = IBC('action','query', ..., ...
               'CallbackExecDetails',@IBC_CallbackExecDetails);
```

where `IBC_CallbackExecDetails()` is a Matlab function created by you that accepts two input arguments (which are automatically populated in run-time):

- `hObject` – the Java connector object that is described in §15 below
- `eventData` – a Matlab struct that contains the event's data in separate fields[120]

An example for specifying a Matlab callback function is:

```
function IBC_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:[121]

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

function IBC_CallbackExecDetails(ibConn,eventData,extra1,extra2)

    % do the callback processing here

end
```

When you specify any callback function to *IBC*, you only need to set it once, in any *IBC* command. Unlike most *IBC* parameters, which are not remembered across *IBC* 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.[122]

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

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

[120] Until *IBC* v2.00, `eventData` contained Java objects whose fields could be inspected via the `struct` function (e.g. `struct(eventData.contract)`). Starting in v2.01, such Java objects are regular Matlab structs. Accessing their fields in *IBC* callbacks remains unchanged (e.g. `eventData.contract.m_symbol`); the objects simply became easier to inspect.

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

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

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.

Using Matlab callbacks with the Java connector object can be used, for example, to implement combo trades, [123] 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[124] 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 *IBC* (for additional information about any of the callbacks, follow the link in the "IB Event" column):

***IBC* parameter**	**IB Event**	**Triggered by *IBC*?**	**Called when?**
CallbackAccountDownloadEnd	accountDownloadEnd	Yes	in response to account queries, after all UpdateAccount events were sent, to indicate end of data
CallbackAccountSummary	accountSummary	Yes	for every single field in the summary data when the account data is requested (see §4.1)
CallbackAccountSummaryEnd	accountSummaryEnd	Yes	indicates end of data after all AccountSummary events are sent
CallbackBondContractDetails	bondContractDetails	Yes	in response to market queries; not really used in *IBC*
CallbackCommissionReport	commissionReport	Yes	immediately after a trade execution, or when requesting executions (see §12.1 below)
CallbackConnectionClosed	connectionClosed	Yes	when **IBC** loses its connection (or reconnects) to TWS/Gateway
CallbackContractDetails	contractDetails	Yes	in response to market queries; used in *IBC* only to get the tick value
CallbackContractDetailsEnd	contractDetailsEnd	Yes	indicates end of data after all ContractDetails events were sent

[123] http://interactivebrokers.github.io/tws-api/basic_orders.html#combolimit

[124] http://interactivebrokers.github.io/tws-api/classIBApi_1_1ComboLeg.html; http://interactivebrokers.com/php/whiteLabel/Interoperability/Socket_Client_Java/java_properties.htm

IBC parameter	IB Event	Triggered by *IBC*?	Called when?
CallbackCurrentTime	currentTime	Yes	numerous times during regular work; returns the current server system time
CallbackDeltaNeutralValidation	deltaNeutralValidation	No	in response to a Delta-Neutral (DN) RFQ
CallbackExecDetails	execDetails	Yes	whenever an order is partially or fully filled, or in response to the Java connector's *reqExecutions()*
CallbackExecDetailsEnd	execDetailsEnd	Yes	indicates end of data after all the ExecDetails events were sent
CallbackFundamentalData	fundamentalData	Yes	in response to requesting fundamental data for a security
CallbackHistoricalData	historicalData	Yes	in response to a historical data request, for each of the result bars separately
CallbackManagedAccounts	managedAccounts	Yes	when a successful connection is made to a Financial Advisor account, or in response to calling the Java connector's *reqManagedAccts()*
CallbackMarketDataType	marketDataType	No	when the market type is set to Frozen or RealTime, to announce the switch, or in response to calling the Java connector's *reqMarketDataType()*
CallbackMessage	error	Yes	whenever IB wishes to send the user an error or informational message. See §14.1 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	Yes	in response to a user query for portfolio positions (for each position in the portfolio)
CallbackPositionEnd	positionEnd	Yes	indicates end of data after all Position events have been sent
CallbackTickPrice	tickPrice	Yes	in response to a market query, for price fields (e.g., bid)

IBC parameter	IB Event	Triggered by *IBC*?	Called when?
CallbackTickSize	tickSize	Yes	in response to a market query, for size fields (e.g., bidSize)
CallbackTickString	tickString	Yes	in response to a market query, for string fields (e.g., lastTimestamp)
CallbackTickGeneric	tickGeneric	Yes	in response to a query with a **GenericTickList** param
CallbackTickEFP	tickEFP	No	when the market data changes. Values are updated immediately with no delay
CallbackTickOptionComputation	tickOptionComputation	No	when the market of an option or its underlier moves. TWS's option model volatilities, prices, and deltas, along with the present value of dividends expected on that underlier are received
CallbackTickSnapshotEnd	tickSnapshotEnd	Yes	when all events in response to a snapshot query request have been sent, to indicate end of data
CallbackRealtimeBar	realtimeBar	Yes	in response to a realtime bars request, for each bar separately
CallbackReceiveFA	receiveFA	No	in response to calling the Java connector's *requestFA()*
CallbackScannerData	scannerData	Yes	in response to a user query for scanner data, for each result row
CallbackScannerDataEnd	scannerDataEnd	Yes	indicates end of data after the last scannerData event was sent
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	Yes	when market depth has changed
CallbackUpdateMktDepthL2	updateMktDepthL2	Yes	when the Level II market depth has changed
CallbackUpdateNewsBulletin	updateNewsBulletin	No	for each new bulletin if the client has subscribed by calling the Java connector's *reqNewsBulletins()*
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 = IBC('action','BUY', 'symbol','GOOG', 'quantity',1, ...
                   'limitPrice',600, ...
                   'CallbackExecDetails',@IBC_CallbackExecDetails);
```

Where the function `IBC_CallbackExecDetails` is defined as follows (for example, in a file called IBC_*CallbackExecDetails.m*):[125]

```
%https://interactivebrokers.com/php/whiteLabel/Interoperability/Socket_Client_Java/java_properties.htm
function IBC_CallbackExecDetails(ibConnector, eventData, varargin)

   % Extract the basic event data components [126]
   contractData  = eventData.contract;
   executionData = eventData.execution;

   % Example of extracting data from the contract object:
   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:
   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  % IBC_CallbackExecDetails
```

[126] Until *IBC* v2.00, `eventData` contained Java objects whose fields could be inspected via the `struct` function (e.g. `struct(eventData.contract)`). Starting in v2.01, such Java objects are regular Matlab structs. Accessing their fields in *IBC* callbacks remains unchanged (e.g. `eventData.contract.m_symbol`); the objects simply became easier to inspect.

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[127] (also see §5.1 above).

Note: according to IB, [128] "*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 = IBC('action','Query', 'symbol','GOOG', ...
                   'GenericTicklist','236', 'QuotesNumber',2, ...
                   'CallbackTickGeneric',@IBC_CallbackTickGeneric);
```

where the function `IBC_CallbackTickGeneric` is defined as follows:[129]

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

    % Only check the shortable tick type =46, according to
    % https://interactivebrokers.github.io/tws-api/tick_types.html#shortable

    if eventData.field == 46  % 46=Shortable (see footnote below)

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

        % Get the corresponding shortable value
        shortableValue = eventData.generic;  % (see footnote below)

        % 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  % IBC_CallbackTickGeneric
```

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

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

[127] https://interactivebrokers.github.io/tws-api/tick_types.html#shortable. Additional details: https://ibkr.info/it/article/2024

[128] https://investors.interactivebrokers.com/php/apiguide/interoperability/generictick.htm

[129] tickGeneric's eventData contains the fields "*field*", "*generic*", which correspond to IB's documented "*tickType*", "*value*" fields.

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.[130]

As noted in §5.4, 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 **LocalSymbol**s in a single command. This relies on the fact that when the **Right** and **Strike** parameters of an option security 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 futures options for Light Sweet Crude Oil (CL) with empty **Right** and **Strike**. We can safely ignore the IB warning about ambiguous or missing security definition:

```
>> data=IBC('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)
```

[130] A synchronous alternative for retrieving the options chain is explained in §5.4 above

The returned `data` struct will naturally contain empty market data, but its contractDetails field will contain useful data about the requested security: [131]

```
>> 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'
                 ...
```

[131] Until *IBC* v2.00, `eventData` contained Java objects whose fields could be inspected via the `struct` function (e.g. `struct(eventData.contract)`). Starting in v2.01, such Java objects are regular Matlab structs. Accessing their fields in *IBC* callbacks remains unchanged (e.g. `eventData.contract.m_symbol`); the objects simply became easier to inspect.

11.5 Example – using CallbackUpdateMktDepth for realtime order-book GUI update

In this example, we wish to update a real-time GUI display of the order-book (at least the top few rows of the book), based on Level II data.

As noted in §7.3 above, market-depth events may be sent at a very high rate from the IB server, and so it is not feasible or useful to update the Matlab GUI for each update. Instead, we update the GUI with the latest data at a steady rate of 2 Hz (twice a second). This can be achieved in two different ways: one alternative is to set-up a periodic timer that will run our GUI-update callback every 0.5 secs, which will call IBC(…,'QuotesNumber',-1) to fetch the latest data and update the GUI.

Another alternative, shown here below, is to attach a user callback function to *updateMktDepth*[132] and *updateMktDepthL2*[133] events, updating an internal data struct, but only updating the GUI if 0.5 secs or more have passed since the last GUI update:

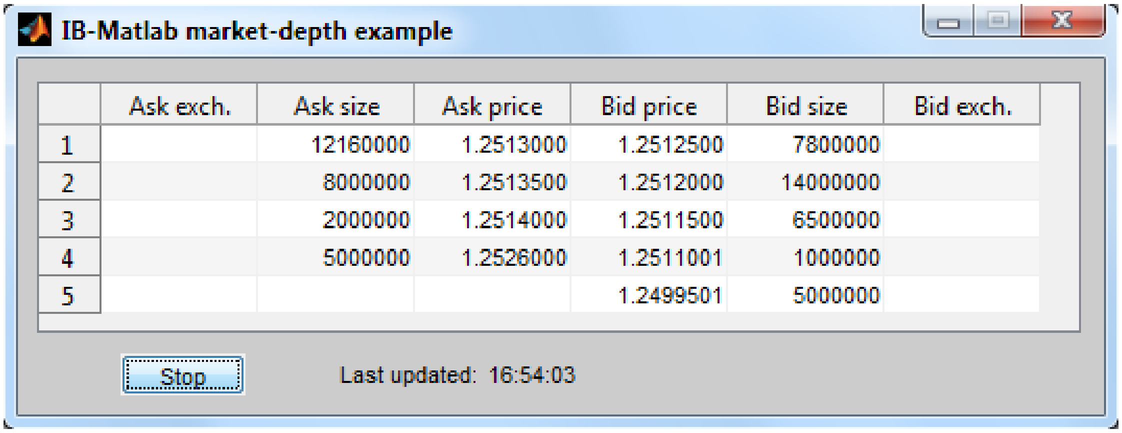
```
% IBC_MktDepth - sample Market-Depth usage function
function IBC_MktDepth(varargin)

    % Initialize data
    numRows = 5;
    depthData = cell(numRows,6);
    lastUpdateTime = -1;
    GUI_refresh_period = 0.5 * 1/24/60/60;  % =0.5 secs

    % Prepare the GUI
    hFig = figure('Name','IBC market-depth example', ...
                  'NumberTitle','off','CloseReq',@figClosedCallback,...
                  'Menubar','none', 'Toolbar','none', ...
                  'Resize','off', 'Pos',[100,200,520,170]);
    color = get(hFig,'Color');
    headers = {'Ask exch.','Ask size','Ask price', ...
               'Bid price','Bid size','Bid exch.'};
    formats = {'char','numeric','long', 'long','numeric','char'};
    hTable = uitable('Parent',hFig, 'Pos',[10,40,500,120], ...
                     'Data',depthData, ...
                     'ColumnName',headers, 'ColumnFormat',formats);
    hButton = uicontrol('Parent',hFig, 'Pos',[50,10,60,20], ...
                        'String','Start', 'Callback',@buttonCallback);
    hLabel1 = uicontrol('Parent',hFig, 'Pos',[120,10,100,17], ...
                        'Style','text', 'String','Last updated:', ...
                        'Horizontal','right', 'Background',color);
    hLabelTime = uicontrol('Parent',hFig, 'Pos',[225,10,100,17], ...
                        'Style','text', 'String','(not yet)', ...
                        'Horizontal','left', 'Background',color);

    % Send the market-depth request to IB using IBC
    contractParams = {'symbol','EUR', 'localSymbol','EUR.USD', ...
                      'secType','cash', 'exchange','idealpro', ...
                      'NumberOfRows',5, varargin{:}};
```

[132] http://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html#ab0d68c4cf7093f105095d72dd7e7a912

[133] http://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html#ad8afb71cd866b423a84555f500992968

```
reqId = IBC('action','query', 'QuotesNumber',inf, ...
                 contractParams{:}, ...
                 'CallbackUpdateMktDepth',  @mktDepthCallbackFcn,...
                 'CallbackUpdateMktDepthL2',@mktDepthCallbackFcn);

% Figure close callback function - stop the market-depth streaming
function figClosedCallback(hFig, eventData)
    % Delete the figure window and stop any pending data streaming
    delete(hFig);
    IBC('action','query', contractParams{:}, 'QuotesNumber',0);
end  % figClosedCallback

% Start/stop button callback function
function buttonCallback(hButton, eventData)
    currentString = get(hButton,'String');
    if strcmp(currentString,'Start')
        set(hButton,'String','Stop');
    else
        set(hButton,'String','Start');
    end
end  % buttonCallback

% Callback functions to handle IB Market Depth update events
function mktDepthCallbackFcn(ibConnObj, eventData)

    % Ensure that it's the correct MktDepth event
    if eventData.tickerId == reqId

        % Get the updated data row
        % Note: Java indices start at 0, Matlab starts at 1
        row = eventData.position + 1;

        % Get the size & price data fields from the event's data
        size  = eventData.size;
        price = eventData.price;

        % Prevent extra LS digits in uitable display
        price = single(price + 0.00000001);

        % Exchange (marketMaker) data is only available in L2:
        try
           exchange = char(eventData.marketMaker);
        catch
           exchange = '';
        end

        % Update the internal data table
        if eventData.side == 0  % ask
           if eventData.operation == 0  % insert
              depthData(row+1:end,1:3) = depthData(row:end-1,1:3);
              depthData(row,1:3) = {exchange, size, price};
           elseif eventData.operation == 1  % update
              depthData(row,1:3) = {exchange, size, price};
           elseif eventData.operation == 2  % delete
              depthData(row:end-1,1:3) = depthData(row+1:end,1:3);
```

```
                  depthData(end,1:3) = {[],[],[]};
               else
                  % should never happen!
               end
            else  % bid (same as ask but the data columns are reversed)
               if eventData.operation == 0  % insert
                  depthData(row+1:end,4:6) = depthData(row:end-1,4:6);
                  depthData(row,4:6) = {price, size, exchange};
               elseif eventData.operation == 1  % update
                  depthData(row,4:6) = {price, size, exchange};
               elseif eventData.operation == 2  % delete
                  depthData(row:end-1,4:6) = depthData(row+1:end,4:6);
                  depthData(end,4:6) = {[],[],[]};
               else
                  % should never happen!
               end
            end

            % Update the GUI if more than 0.5 secs have passed and
            % the <Stop> button was not pressed
            isStopped = strcmp(get(hButton,'String'),'Start');
            if now - lastUpdateTime > GUI_refresh_period && ~isStopped
                set(hTable,'Data',depthData);
                set(hLabelTime,'String',datestr(now,'HH:MM:SS'));
                lastUpdateTime = now;
            end
        end
    end  % mktDepthCallbackFcn

end  % IBC_MktDepth
```



	Ask exch.	Ask size	Ask price	Bid price	Bid size	Bid exch.
1		12160000	1.2513000	1.2512500	7800000	
2		8000000	1.2513500	1.2512000	14000000	
3		2000000	1.2514000	1.2511500	6500000	
4		5000000	1.2526000	1.2511001	1000000	
5				1.2499501	5000000	

Last updated: 16:54:03

12 Tracking trade executions

IBC 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,[134] use **Action**='query' and **Type**='executions' as follows (note the similarities to the request for open order, §10.1 above):

```
>> data = IBC('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]
```

[134] To view executions from previous days, open the Trades Log in TWS and request executions while the Trades Log is displayed

```
>> 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:[135]

- `orderId` – this is the ID returned by *IBC* 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

[135] http://interactivebrokers.github.io/tws-api/classIBApi_1_1Execution.html

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

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

```
>> data = IBC('action','query','type','executions','symbol','goog')
```

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

```
>> data = IBC('action','query','type','executions','symbol','xyz')
data =
     []
```

12.2 Automated log files

IBC automatically stores two log files of trade executions. Both files have the same name, and different extensions:

- 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`.

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*. The log file name will remain unchanged until you modify it or restart Matlab.

The log filename can be modified by setting the **LogFileName** parameter (default = './IB_tradeslog_YYYYMMDD.csv') when you specify a trade order:

```
newLogFileName =  ['./IB_tradeslog_' datestr(now,'yyyymmdd') '.csv'];
orderId = IBC('Action','Buy', 'LogFileName',newLogFileName, ...);
```

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. Also note that the new **LogFileName** should end with '.csv'.

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. Section §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 *IBC* command that sends the trade order). You do not need to re-specify this callback in subsequent *IBC* 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 TWS connection parameters

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

IBC connects to whichever TWS is currently active. If you login to TWS with a paper-trade login, *IBC* will work on the simulated account, and similarly for a live account. TWS's account type is transparent to *IBC*: the only way to control whether *IBC* will use simulated/live trading is to use the appropriate TWS login.

IBC automatically connects to IB when any request that requires IB is made and the connection is not live. This happens upon the first *IBC* request (when the initial connection needs to be established); after TWS/Gateway closed; after calling `IBC('disconnect')` or `ibConnectionObject.disconnectFromTWS` (see §15.1); after Matlab restarts; when specifying a new **ClientID**; and in a few other special cases.[136]

```
data = IBC('action',...);  % do whatever
IBC('disconnect');   % disconnect from IB
data = IBC('action','portfolio');  % will automatically reconnect
```

To programmatically check whether **IBC** is currently connected to IB, do this:

```
[~,ibConnectionObject] = IBC;
flag = ibConnectionObject.isConnected;   % true/false
```

IBC 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 ok for the majority of use-cases.

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

Parameter	Data type	Default	Description
ClientId	integer	(*random*)	A number that identifies **IBC** 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 (§8.5), 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 #5,6).

[136] See discussion of the **ReconnectEvery** parameter (§7.1 above).

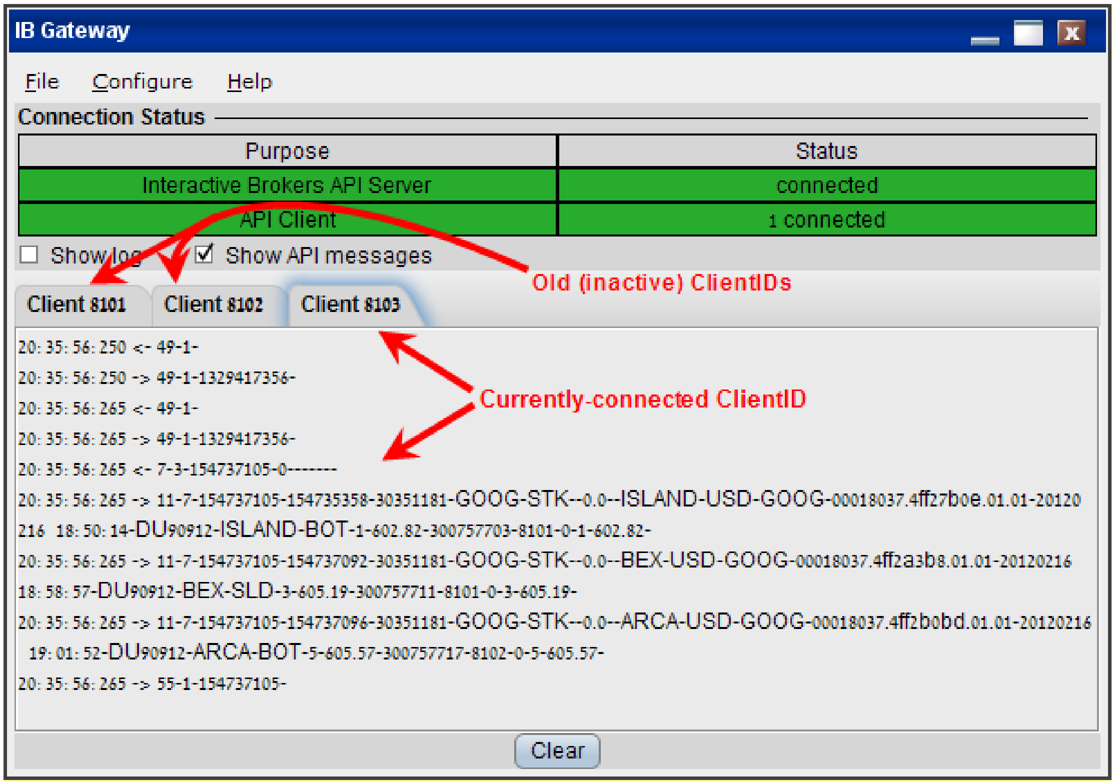
In reconnections of any kind, **IB-Matlab** automatically tries to reuse the same **ClientID** as in the previous connection, even if you do not explicitly specify the **ClientID**.

Setting a static **ClientID** can be used to modify open orders (and track order executions) placed in a different **IBC** session (i.e., after the original **IBC** client has disconnected from IB and a new **IBC** has connected). IB normally prevents clients from accessing orders placed by other clients, but this limitation will not affect you if all your **IBC** sessions use the same **ClientID**.

ClientID=0 is special: it simulates the TWS and enables **IBC** to receive/ modify/cancel open orders that were interactively entered in TWS (not via **IBC**). Instead of **ClientID** 0, you can use any other value that you pre-configured as the *Master API Client ID* in TWS/Gateway's API configuration screen (see §2 installation steps #5,6). Using a *Master API Client ID* enables **IBC** to track/modify open orders that were placed in the IB account using any client ID. 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.

When a new **ClientID** is specified for any *IBC* command, *IBC* automatically disconnects the previous client ID and reconnects as the new **ClientID**. 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 = IBC('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-
```

14 Handling errors, problems, and IB messages

14.1 Messages sent from IB

IB constantly sends messages of various severity levels to **IBC**. 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 {153745243, 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).

IBC automatically infers whether an IB message is an error, warning or informational message. Errors are sent to the standard error stream (*stderr*) and displayed in **red** in the Matlab console.[139] Other messages are sent to the standard output (*stdout*) and displayed in regular black text on the Matlab console.

Users can control the display of IB messages in the Matlab console 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)

The information contained in the message events varies depending on message type.[140] The events have 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, data2>3000
message, exception object	severe IB errors (exceptions)	[API.msg3]	**MsgDisplayLevel** < 2

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

A typical example of such messages is the following:

```
>> data = IBC('action','query', 'symbol','EUR');
[API.msg2] No security definition has been found for the request
{153745243,200}
```

[139] The Matlab console is the Desktop Command Window in the case of an interactive Matlab session, or the command-prompt window in the case of a compiled (deployed) program.

[140] http://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html#a7dfc221702ca65195609213c984729b8

Most IB messages are of type msg2 (as in this example) and contain two numeric fields: data1 contains message-specific ID, which typically corresponds to the order ID or request ID that caused the problem; data2 contains the message code (type). In the example above, type='msg2', id (data1) =153745243, and code (data2) =200. This tells us that this error (code=200) occurred for request ID 153745227, so we can correlate between the error and our request. Some messages do not have an associated message-specific ID; in such cases id (data1) = -1. For example, "*Market data farm connection is OK:cashfarm*" (code 2104) is a general message about IB connectivity that is not related to any specific user request or order, so its id (data1) will be -1.

The full list of message codes (data2) for API.msg2 (which is the most common message type) is listed online.[141] It is sub-divided into three groups:

- Error messages (data2 codes between 100-999 or >10000)
- System messages (data2 codes between 1000-1999)
- Warning messages (data2 codes between 2000-2999)

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 = IBC('action','query', ..., 'MsgDisplayLevel',-1, ...
               'CallbackMessage',@IBC_CallbackMessage);
```

In this example, all IB messages will be passed through `IBC_CallbackMessage`, which is a Matlab function that we should create (it is not part of *IBC*). Here is a sample implementation of such a message processing function that you can adapt:[142]

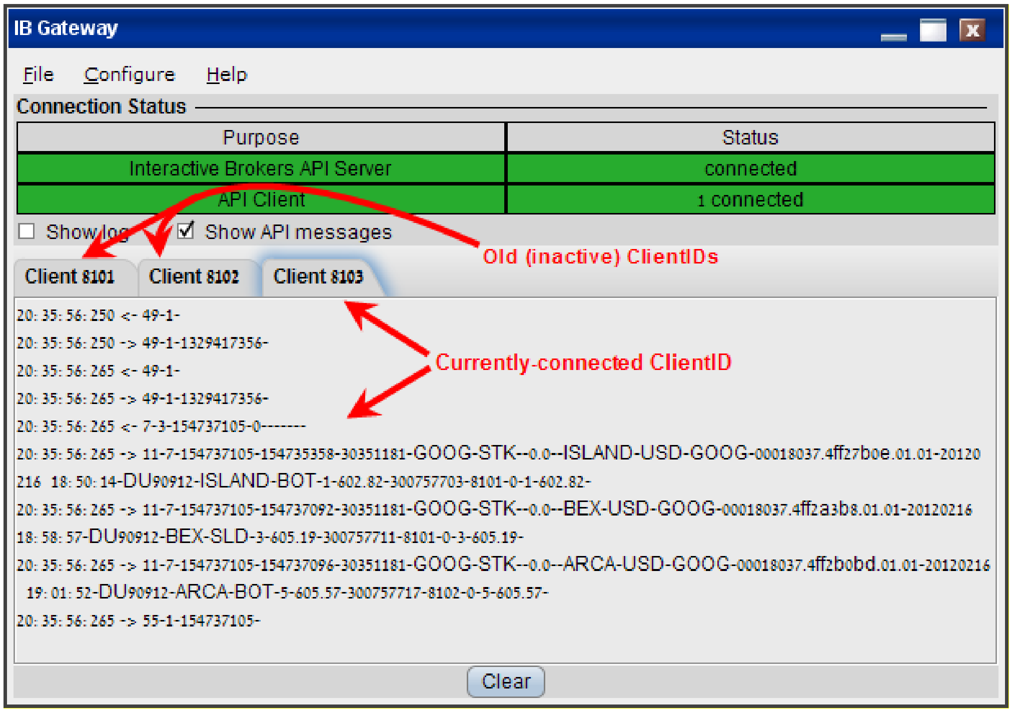
```
% Processing function for IB messages
function IBC_CallbackMessage(ibConnector, eventData)

    % Process messages based on their code:
    if strcmp(eventData.type, 'msg2')  % most common message: API.msg2
        switch eventData.data2  % data2 contains the message code
            case 100   % Max rate of message has been exceeded
                msgbox(eventData.message, 'IB error');
            case 200   % Ambiguous/non-existing contract specification
                disp('Ambiguous (or invalid) contract specs')
            ...
            otherwise  % Any other message type
        end
    else  % msg1 or msg3
        ...
    end

end
```

[141] https://interactivebrokers.github.io/tws-api/message_codes.html

[142] See §11 for additional explanations and usage examples of user callback functions for IB messages/events

IB's error messages are often cryptic. It is sometimes difficult to understand the problem's root cause.[143] Several mechanisms can help us with this detective work:

- We can set *IBC*'s **MsgDisplayLevel** parameter to -1 or -2 (see above).
- We can set *IBC*'s **Debug** parameter to 1 (default=0). This will display in the Matlab Command Window a long list of parameters used by *IBC* to prepare the request for IB. Check this list for any default values that should actually be set to some non-default values.
- We can set the API logging level to "Detailed" in the TWS/Gateway API configuration window.[144] By default it is set to "Error", and can be changed at any time. This affects the amount of information (verbosity) logged in IB's log files that are located in IB's installation folder (e.g., C:\Jts).[145] 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, 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 (see screenshot below). 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".



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-
```

Old (inactive) ClientIDs

Currently-connected ClientID

[143] See examples in §14.2 above

[144] See §2, installation step 5d. Also see http://interactivebrokers.github.io/tws-api/support.html#tws_logs

[145] http://interactivebrokers.github.io/tws-api/logs.html

14.2 Ambiguous/invalid security errors

Much of *IBC*’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:[146]

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

Unfortunately, IB is not very informative about what exactly was wrong with our request; we need to discover this ourselves. It turns out that in this specific case, we need to specify a few additional parameters, since their default values (**SecType**='STK', **Exchange**='SMART', **LocalSymbol**=**Symbol**) are incorrect for this security:

```
>> data = IBC('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
```

[146] The error messages can be suppressed using the **MsgDisplayLevel** parameter, and can also be trapped and processed using the CallbackMessage parameter – see §14.1 below for details

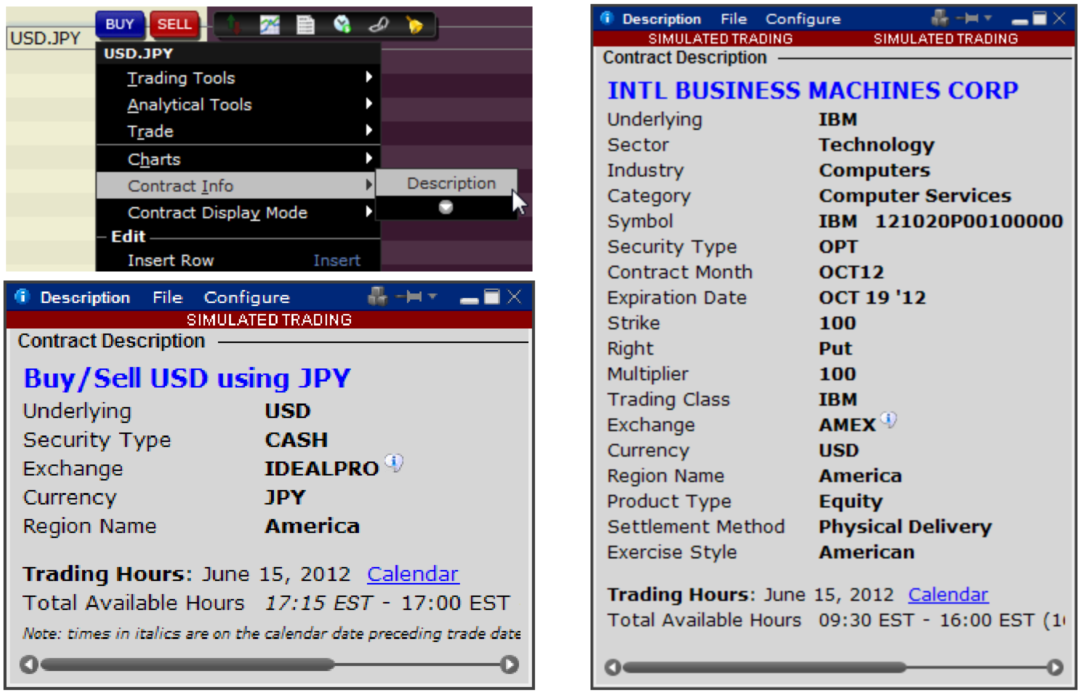
In other cases, we may also need to specify the **Currency** (default='USD'). For example, the Russell 2000 index (RUT) is listed on the Toronto Stock Exchange (TSE) and trades in CAD currency. Likewise, the USD.JPY currency pair trades in Yens (JPY currency), not USD.[147] Similarly, when **Exchange**='SMART' and **Symbol**='IBM', the **Currency** must be specified since IBM trades in either GBP or USD. Due to such potential ambiguities it is a good idea to always specify **Currency**.

For options/future we also need to specify the **Expiry**, **Strike** and **Right** parameters. In some cases, specifying the **Expiry** in YYYYMM format is ambiguous because the underlying contract has several separate futures/options expiring in the same month:

```
>> data = IBC('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 is to specify the **Expiry** date in YYYYMMDD format (i.e., specify the exact full date rather than just the month), or to specify the **Multiplier** parameter.

If you are unsure of a security's contract details, try using different parameter values. Alternatively, 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

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 121020P00100000' (**Symbol** is 'IBM'). This **LocalSymbol** has multiple spaces[148]. For this reason it is best to copy-paste the value directly from the window.

[147] http://interactivebrokers.com/en/index.php?f=2222&ns=T&exch=ibfxpro

[148] OSI specification: http://interactivebrokers.com/download/ociguide.pdf, http://en.wikipedia.org/wiki/Option_symbol

Alternatively, use your TWS paper-trade (simulated trading) account to buy a virtual unit of the security, then use **IBC** to read the portfolio (see §4 below) and check the reported contract data. For example:

```
>> data = IBC('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:[149]

LocalSymbol	Exchange	SecType	Currency	Description
CO	SMART	STK	USD	Cisco Corp., SMART (NASDAQ)
GE	SMART	STK	USD	General Electric, SMART (NYSE)
VOD	LSE	STK	GBP	Vodafone Group, London Stock Exch.
SPX	CBOE	IND	USD	S&P 500 Index
INDU	NYSE	IND	USD	Dow Jones Industrials Average Index
ESM1	GLOBEX	FUT	USD	Emini S&P 500 (ES) 6/2011 futures
ES	GLOBEX	CONTFUT	USD	Emini S&P continuous (rolling) future
YM JUN 11	ECBOT	FUT	USD	Emini Dow (YM) 6/2011 future Note: 3 spaces between symbol and month,1 space between month, year
QMN5	NYMEX	FUT	USD	Crude Oil (QM) 6/2005 future
FGBL DEC 11	DTB	FUT	EUR	German Bund 12/2011 future
P OZN DEC 15 13100	ECBOT	FOP	USD	10-year T-note (ZN) 11/2015, 131.0 Put future-option. Note 2 spaces, twice
XAUUSD	SMART	CMDTY	USD	London Gold Spot
EUR.USD	IDEALPRO	CASH	USD	Euro/Dollar currency pair

Traders who wish to get the full option chain list are referred to §5.4 and §11.4.

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

14.3 Programmatic errors

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

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

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

Try-catch blocks have negligible performance and memory overheads 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 *IBC* 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?).

Sometimes it is preferable to silently trap errors, rather than receive a Matlab exception error. This can be done by specifying a 5th output argument in the call to *IBC*:

```
[orderId, ibConnectionObject, contract, order, errMsg] = ...
    IBC('action','BUY',...);
if ~isempty(errMsg)
    % process the error here (errMsg is a char array)
end
```

In most cases we do not need the `ibConnectionObject`, `contract`, `order` output args of *IBC* (see §9.6 for their usage), so such cases can be simplified to this:

```
[orderId, ~, ~, ~, errMsg] = IBC('action','BUY',...);
if ~isempty(errMsg)
    % process the error here (errMsg is a char array)
end
```

Programmatic errors may occur in programs that rely on valid account, portfolio or market-data data returned from IB. Unfortunately, sometimes 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, is discussed in §14.2 and §4. User programs should implement sanity checks on the returned data, and resend the request until valid data is received. Failing to do so may result in applicative errors or bad trading decisions.

A common cause of program errors is due to specifying numeric values as strings or vice versa. For example, specifying 12 rather than "12", or "0" rather than 0 or false. Numbers should not be enclosed with quote marks when specifying parameter values. For example, specify `IBC(…, 'Strike',5.20)`, not `IBC(…, 'Strike','5.20')`, otherwise Matlab might get confused when trying to interpret the string `'5.20'` as a number. Each parameter has a specific data type, which is listed in the parameter tables in this guide. **IBC** is often smart enough to automatically convert to the correct data type, but you should not rely on this: it is better to always use the correct data type.

Another common cause of errors when using **IBC** is relying on default parameter values (for example, relying on the default **SecType** ('STK') for non-equity contracts (Forex, future, option etc.); see §3.2 and §14.2 for additional details.

The final type of error is 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. These can be fixed by running **IBC** on a computer having more memory, or by reducing the amount of stored data.[150]

Java memory errors are recognized by the message "`java.lang.OutOfMemoryError: Java heap space`". They can be solved by increasing Matlab's pre-allocated Java heap memory using Matlab's preferences, or via a *java.opts* file.[151]

A possible cause of confusion is Matlab's default use of the "short" format, which rounds numbers in the Matlab console (Command Window) to 4 digits after the decimal. **This is not an error or bug**: The data actually has higher precision, so when we use it in a calculation the full precision is used; this is simply not displayed in the console.

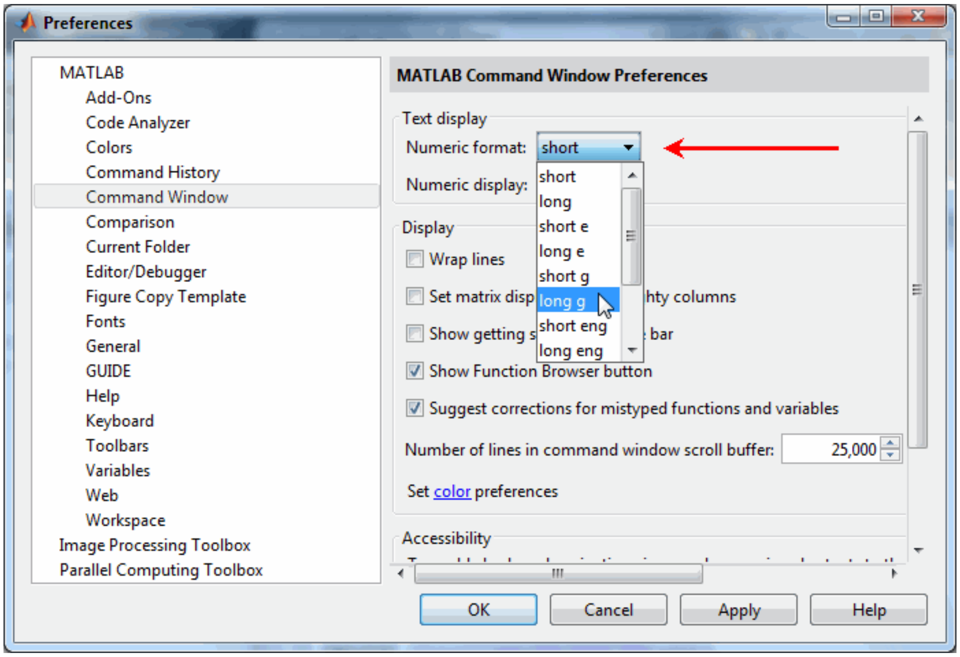
IBC does not truncate/round/modify the IB data in any manner!

To display the full numeric precision in the Matlab console, change your Matlab's Command Window's Numeric Format from "short" to "long" (or "long g") in Matlab's Preferences window, or use the "`format long`" Matlab command:

```
>> data = IBC('action','query','localsymbol','EUR.USD',...);
>> data.askPrice   % short format
ans =
    1.0727
>> format long g   % long format
>> data.askPrice
ans =
    1.07265
```

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

[151] https://www.mathworks.com/matlabcentral/answers/92813-how-do-i-increase-the-heap-space-for-the-java-vm-in-matlab



MATLAB Command Window Preferences

Text display

Numeric format: short

Numeric display:

Display

□ Wrap lines

□ Set matrix disp… hty columns

□ Show getting s… bar

☑ Show Function Browser button

☑ Suggest corrections for mistyped functions and variables

Number of lines in command window scroll buffer: 25,000

Set color preferences

Accessibility

14.4 Troubleshooting specific problems/errors

Error	Description / solution	Sections
`Couldn't connect to TWS. Confirm that "Enable ActiveX and Socket Clients" is enabled on the TWS "Configure->API" menu.`	Enable ActiveX etc. in TWS/Gateway API settings. Note that IBGateway has a separate set of settings than TWS.	2.0 steps #9-10
`failed to connect using Port=7496; retrying to connect using Port=4001...`	TWS and IBGateway use different default ports for API connections. You can either modify their settings to use the same value, or specify the **Port** parameter in your *IBC* command, or simply ignore this message.	2.0 steps #9-10
`IBC is not activated on this computer`	Some component of your activated computer fingerprint has changed. Revert this change or contact us to modify the activated fingerprint.	2.1
`Your IBC license has expired on 1-Jun-2014`	**IBC**'s license is limited in duration. When the license term expires, you can contact us to renew it.	2.1
`Cannot connect to Authorization server to validate your IBC license`	**IBC** validates its license on the authorization server. Your internet connection may be down, or this domain may be blocked by your firewall (your IT admin can unblock it).	2.1
`IBC.jar was not found in the static Java classpath`	**IBC** cannot work properly unless its Java file (IBC*.jar*) is added to Matlab's static Java classpath.	2.3
`NullPointerException com.mathworks.jmi.bean.-MatlabBeanInterface.-addCallback`	**IBC** cannot work properly unless its Java file (IBC*.jar*) is added to Matlab's static Java classpath.	2.3
(Commands can be sent to IB, but no data is received from IB)	**IBC** cannot receive IB data unless its Java file (IBC*.jar*) is added to Matlab's static Java classpath.	2.3
`Max rate of messages per second has been exceeded: max=50 rec=55`	IB server limits the rate of messages sent to the IB server to 50 msgs/sec. Reduce your requests rate using Matlab's ***pause(0.02)*** command.	3.1, 8

Error	Description / solution	Sections
`Requested market data is not subscribed`	You are not subscribed for IB real-time data for the asset. Sometimes happens when security parameters are incorrect.	3.2, 5.1

`The account code is required for this operation`
or:
`You must specify an allocation (either a single account, group, or profile)`
	You manage multiple IB accounts, and IB does not understand which of these accounts relates to your requested action. Specify the **AccountName** or **FAGroup** or **FAProfile** parameter.	3.2, 8.5, 9.7

`Historical Market Data Service error message:`
`No market data permissions for NYSE STK`
	You are not subscribed to the IB service of fetching historical data for the specified security type or exchange.	5.1

`Deep market data is not supported for this combination of security type/exchange`	The specified exchange does not provide market depth information for the requested security type.	5.2
`Historical data request pacing violation`	Historical and real-time streaming data is subject to IB's strict pacing limits. Either limit your requests rate, or ask your IB representative to remove/raise your account limits.	6, 7.1
`Historical data bar size setting is invalid`	IB only accepts some combinations of barSize/duration in historic data requests	6
`Historical Market Data Service error message:No historical market data for EUR/CASH@IDEALPRO Last 1d`	The default **WhatToShow** parameter value is 'Trades', which is not supported for Forex. Specify a 'Midpoint' parameter value here.	6
(IBC stops receiving streaming data from IB)	Streaming data from TWS is sometimes stopped, depending on data rate. Try to set **ReconnectEvery**, or restart Matlab.	7.1
`Symbol "IBM" is not currently streaming`	Start the streaming (**QuotesNumber**>0) before requesting any streamed data.	7.1, 7.2, 7.3
`Can't find EID with tickerId`	You can safely ignore this message. It represents a harmless request from *IBC* to IB, to cancel a streaming data request that was already cancelled.	7.2
`The order size cannot be zero`	You have either specified **Quantity**=0, or **FAPercentage** with invalid direction	8.1, 8.5
`Order rejected - reason: Invalid value in field # 6159`	One of the provided parameter values (e.g., **FAMethod**) is invalid.	8.5

Error	Description / solution	Sections
`Invalid ICS spread`	You specified incorrect **ComboRatios** values between the combo-order legs.	9.5
`Unable to modify this order as it is still being processed`	You tried to modify an order before it was fully registered by the IB server.	9.6
`Exercise ignored because option is not in-the-money`	You tried to exercise an out-of-money option without specifying **Override**.	9.7
`No unlapsed position exists in this option in account`	You tried to exercise an option that does not exist in your account.	9.7
`Order rejected - reason: The time-in-force is invalid for IB algorithmic orders`	You tried to send an IBAlgo order with an invalid (or default='GTC') value. Use **TIF**='Day' instead.	9.8
`No security definition has been found for the request`	The requested security's parameters are not properly specified and IB cannot identify it. Try specifying additional contract parameters.	14.2
(Missing digits displayed in Matlab Command Window)	Matlab's display format may be set to “short” instead of “long”	14.2
`The contract description specified for TNA is ambiguous; you must specify the multiplier`	The requested security's parameters are not properly specified and IB cannot identify it. Try specifying additional contract parameters.	14.2

`Out of memory`
or:
`Maximum variable size allowed by the program is exceeded`
or:
`Requested array exceeds maximum array size preference`
	This Matlab error might occur when receiving huge amounts of streaming/ historic data. Different Matlab releases display different messages with the same basic idea. Run **IBC** on a computer with more memory, or reduce the amount of stored/processed data.	14.3

`java.lang.OutOfMemoryError: Java heap space`	Run Matlab with more allocated Java heap memory than the default value of 64MB or 128MB (depending on Matlab release). This can be set in Matlab's preferences, or via a *java.opts* file.	14.3
`Unknown parameter 'xyz'`	The specified parameter provided in the *IBC* command is not a valid *IBC* parameter. Refer to the relevant User Guide section for a list of acceptable parameter names, or type “`help IBC` ” in the Matlab console.	

15 Using the Java connector object

15.1 Using the connector object

Each call to *IBC* 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 *IBC* with a single output:

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

However, flexible and feature-rich as *IBC* is, it does not contain the entire set of functionalities exposed by IB's Java API. We can use `ibConnectionObject` to access additional functionalities:

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

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

```
[data, ibConnectionObject] = IBC('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[152] and responses[153]). 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, and if you have any question on it you should ask IB about it.

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 *IBC* name-value pair interface. Placing trade orders via both interfaces in a single **IBC** session will result in request ID mixup and failed trades: the IB server will reject trades that 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.

[152] https://interactivebrokers.github.io/tws-api/classIBApi_1_1EClientSocket.html

[153] https://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html

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 boolean isConnected()

    // Disconnect from TWS
    public void disconnectFromTWS()

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

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

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

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

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

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

    // Cancel 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)

// Create a contract
public Contract createContract()  // default Contract: USD on SMART, empty fields
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()  // create default Order: Buy 0 LMT 0, RTH only
public Order createOrder(String action, int quantity, String type,
                         double lmtPrice, double auxPrice, String tif,
                         String ocaGroup, int parentId, String goodAfterTime,
                         String goodTillDate, double trailStopPrice,
                         int triggerMethod, boolean outsideRTH)
// Cancel a placed order (if still open)
public void cancelOrder(int tickerId)

// 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, boolean outsideRTH)

// 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 (boolean autoBindFlag)
```

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

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

// Request IB news bulletins
public void reqNewsBulletins (boolean 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)[154]
public void reqRealTimeBars(int tickerId, Contract contract, int barSize,
                            String whatToShow, boolean 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

// Calculate the implied volatility of a contract
public void calculateImpliedVolatility(int tickerId, Contract contract,
                                       double optionPrice, double underPrice)
// 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 underPrice)
// Cancel request to calculate an option price
public void cancelCalculateOptionPrice(int tickerId)

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

[154] http://interactivebrokers.github.io/tws-api/realtime_bars.html

```
// 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)

// 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(boolean flag)
// Get the Done flag
public boolean 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)
// Indicates end of execution report messages
public void execDetailsEnd(int reqId)
```

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

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

// 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)

// Indicates end of open orders messages
public void openOrderEnd()

// 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 timeStamp)

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

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

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

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

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

```
    // Receives real-time bars data
    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 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 Reuters global fundamental market data
    public void fundamentalData(int reqId, String data)

    public void accountDownloadEnd(String accountName)

    public void deltaNeutralValidation(int reqId, UnderComp underComp)

    // 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 (also see §9.8).[155]

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 **IBC**:

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);
```

[155] http://interactivebrokers.github.io/tws-api/ibalgos.html#arrivalprice, http://interactivebrokers.com/en/software/tws/usersguidebook/algos/arrival_price.htm, http://interactivebrokers.com/en/index.php?f=1122

And now for the corresponding Matlab code (notice how closely it resembles the original Java code):[156]

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

% Get the ibConnectionObject reference from IBC
[dummy, ibConnectionObject] = IBC('action','account');

% If IBC is already connected we can get this reference faster:
[dummy, ibConnectionObject] = IBC();  % faster alternative

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

% Alternatively, we could have done as follows:
m_contract = ibConnectionObject.createContract();
m_contract.m_symbol   = 'MSFT';
m_contract.m_secType  = 'STK';
m_contract.m_exchange = 'SMART'; %=default value so not really needed
m_contract.m_currency = 'USD';   %=default value so not really needed
% end of alternative code

% 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, using the **Hold** parameter, is explained in §9.6 above.

16 Sample strategies/models using IBC

16.1 Pairs trading

16.1.1 Once a day – decide whether two securities are co-integrated [157]

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 [158]).

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 **IBC** to get last year's daily history data for both securities: [159]

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

   GOOG_history = IBC('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.

[157] 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…

[158] 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

[159] 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). [160] 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.[161]

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.[162]

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$_{\mathrm{sec1}}$ – β*Close$_{\mathrm{sec2}}$) for later use in the runtime part below

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

9. Use **IBC** to stream quote data for the two securities: [163]

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

10. Use **IBC** to attach our user-defined callback processing function: [164]

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

[160] 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

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

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

[163] See §7 for details

[164] See §11 for details

16.1.2 Runtime – process TickPrice streaming-quote events

```
% Callback function to process IB TickPrice events
function IBC_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 [165]

            % If the securities diverged too much, start trading
            if deltePrice < meanSpread -2*modelData.std

               % GOOG overbought vs. IBM, so buy IBM, sell GOOG
               IBC('action','BUY',  'Quantity',1, 'Type','MKT',...
                        'Symbol','IBM');
               IBC('action','SELL', 'Quantity',1, 'Type','MKT',...
                        'Symbol','GOOG');

            elseif deltePrice > meanSpread + 2*modelData.std
               % GOOG oversold vs. IBM, so sell IBM, buy GOOG
               IBC('action','SELL', 'Quantity',1, 'Type','MKT',...
                        'Symbol','IBM');
               IBC('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  % IBC_CallbackTickPrice
```

[165] 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

16.2 Using RSI technical indicator

In this section, we use the RSI technical indicator[166] as a simple example of automated trading based on indicator value. The examples can easily be adapted to other indicators, or combinations of indicators.

In the following downloadable code, [167] we use the MathWorks Financial Toolbox's ***rsindex*** function[168] to generate the RSI indicator values. Users who do not have this toolbox can recreate the ***rsindex*** function based on RSI's mathematical definition.

Naturally, any real-life automated trading system would need to employ additional fail-safe mechanisms in the code, to protect against incorrect data, market outages/crashes etc. For example, you might stop the continuous timers if certain data conditions occur, or you might add stop-loss trading orders. The code below only shows a bare-bones implementation, in order to show how **IBC** could be used.

If you need assistance in adapting the code for your specific needs, contact us for a consulting offer.

```
function [hAxes, hTimer] = IBC_AlgoRSI(varargin)
% IBC_AlgoRSI - simple Price-RSI plot and trading algo
%
% Inputs:     list of IBC parameters e.g. 'Symbol', 'Exchange',
%             'Currency', 'BarSize', 'Quantity', 'AccountName', etc.
% Processing: Display interconnected plots of price vs. RSI
% Output:     hAxes  - handles to plot axes
%             hTimer - handle to the update timer object
% Example:    IBC_AlgoRSI('Symbol','EUR', 'SecType','cash',
%                              'Exchange','idealpro', 'Quantity',100);

   % Ensure that we have all mandatory input params
   IBC_params = varargin;
   symbolIdx = find(strcmpi(IBC_params,'symbol')) + 1;
   if isempty(symbolIdx)
       error('IBC:NoSymbol','Symbol must be specified!');
   end
   symbol = upper(IBC_params{symbolIdx});

   % Prepare the GUI window
   hAxes = prepareGUI();

   % Prepare a timer to periodically check market conditions
   period = 60;  % [seconds]
   tagStr = ['RSI_' symbol];
   hTimer = timer('ExecutionMode','fixedRate', 'Period',period, ...
                  'Name',tagStr, 'Tag',tagStr, ...
                  'TimerFcn',@timerFcn);
   start(hTimer);
```

[166] http://binarytribune.com/forex-trading-academy/relative-strength-index, http://en.wikipedia.org/wiki/Relative_strength_index

[168] http://mathworks.com/help/finance/rsindex.html

```
% Prepare GUI window
function hAxes = prepareGUI()
    % Create a new figure window for this symbol
    titleStr = ['Price / RSI for ' symbol];
    figure('NumberTitle','off', 'Name',titleStr, ...
           'Tag',tagStr, 'CloseRequestFcn',@closeFig);
    zoom on;

    % Prepare the plot axes
    hAxes(1) = axes('Units','norm', 'Pos',[.1,.50,.87,.45]); %prices
    hAxes(2) = axes('Units','norm', 'Pos',[.1,.25,.87,.20]); %RSI
    hAxes(3) = axes('Units','norm', 'Pos',[.1,.05,.87,.15]); %open

    % Link the X axes of the plots
    linkaxes(hAxes,'x');
end

% Callback function upon closing the figure window
function closeFig(hFig, eventData)
    try stop(hTimer); delete(hTimer); catch, end
    delete(hFig);
end

% Get the latest historical/intraday data from IB
function data = getData()
    % Get the historical/intraday data bars from IB
    % Note: IBC_params are persisted in the parent function
    bars = IBC('Action','history', 'BarSize','5 mins', ...
                    'DurationValue',1, 'DurationUnits','W', ...
                    'WhatToShow','MIDPOINT', IBC_params{:});

    % Ensure that enough data is available, otherwise bail out
    if length(bars.close) <= 15
        error('IBC:NoData','Not enough data points!');
    end

    % Determine the RSI values for the retrieved data
    % rsindex() requires a column vector, so transpose the data
    RSI_vector = rsindex(bars.close');

    % Create timeseries objects for plotting
    data.times = datestr(datenum(bars.dateTime, ...
                                 'yyyymmdd  HH:MM:SS'));
    data.ts_close = timeseries(bars.close, data.times);
    data.ts_rsi   = timeseries(RSI_vector, data.times);
end

% Trade based on the retrieved IB data
function data = tradeData(data)
    % Initialize the data points and trade action
    RSI_vector = data.ts_rsi.Data;
    openPos = zeros(1,length(RSI_vector));
    action = '';
    % Get the requested trading quantity
    quantityIdx = find(strcmpi(IBC_params,'quantity'))+1;
    quantity = 1;
    if ~isempty(quantityIdx)
        quantity = IBC_params{quantityIdx};
    end

    % Calculate the trading indicator signals
    for idx = 2 : length(RSI_vector)
        currentRSI = RSI_vector(idx);
        lastRSI    = RSI_vector(idx-1);

        % If position is open
        if openPos(idx-1) ~= 0
            % If RSI has crossed 50
            if (currentRSI-50)*openPos(idx-1) > 0
                % Close the open position
                if openPos(idx-1) > 0
                    action = 'Sell';
                else
                    action = 'Buy';
                end
                openPos(idx) = 0;
            else
                % Else, do nothing for now
                openPos(idx) = openPos(idx-1);
            end

        % Else, if RSI>70 going down (overbought reversal?)
        elseif currentRSI < lastRSI && currentRSI > 70
            % Place a SELL order
            action = 'Sell';
            openPos(idx) = -quantity;

        % Else, if RSI<30 going up (oversold reversal?)
        elseif currentRSI > lastRSI && currentRSI < 30
            % Place a BUY order
            action = 'Buy';
            openPos(idx) = quantity;
        end
    end

    % Trade if an action was triggered for latest data bar
    if ~isempty(action) && openPos(idx)~=openPos(idx-1)
        fprintf('%s %sing %d %s\n', ...
                datestr(now), action, quantity, symbol);
        IBC('Action',action, 'Type','MKT', ...
                 'Quantity',quantity, IBC_params{:});
    end

    % Add the trading decisions to returned data (for plotting)
    data.ts_pos = timeseries(openPos, data.times);
end
```

```
   % Plot the updated data
   function plotData(data)
       % hAxes is persisted in the parent function
       plotTS(hAxes(1), data.ts_close, 'Closing price', true);
       plotTS(hAxes(2), data.ts_rsi,   'RSI',           true);
       plotTS(hAxes(3), data.ts_pos,   'Open pos',      false);
   end

   % Timer callback function
   function timerFcn(hTimer,eventData)
       try
           data = getData();
           data = tradeData(data);
           plotData(data);
       catch err
           disp(err.Message);
           return;
       end
   end
end  % IBC_AlgoRSI

% Utility function to plot a time-series on an axes
function plotTS(hAxes, data, label, emptyXTicksFlag)
    axes(hAxes);
    plot(data);
    box off;
    grid on;
    title('');
    ylabel(label);
    if emptyXTicksFlag
        set(hAxes, 'XTick',[]);
    end
end
```

For performance considerations, we can query only the latest market snapshot, or use streaming realtime bars,[169] rather than retrieving and processing the entire historical/intraday data series as above. Naturally, this would involve additional code to store the previous data bars and trading decisions. If we use the asynchronous streaming realtime bars mechanism, we could consider replacing the timer callback with a callback on the RealtimeBar event (i.e., set the **CallbackRealtimeBar** parameter in *IBC*), to process the bars as soon as they arrive from IB. This is more complex than using a simple timer to query the latest streamed data, of course.

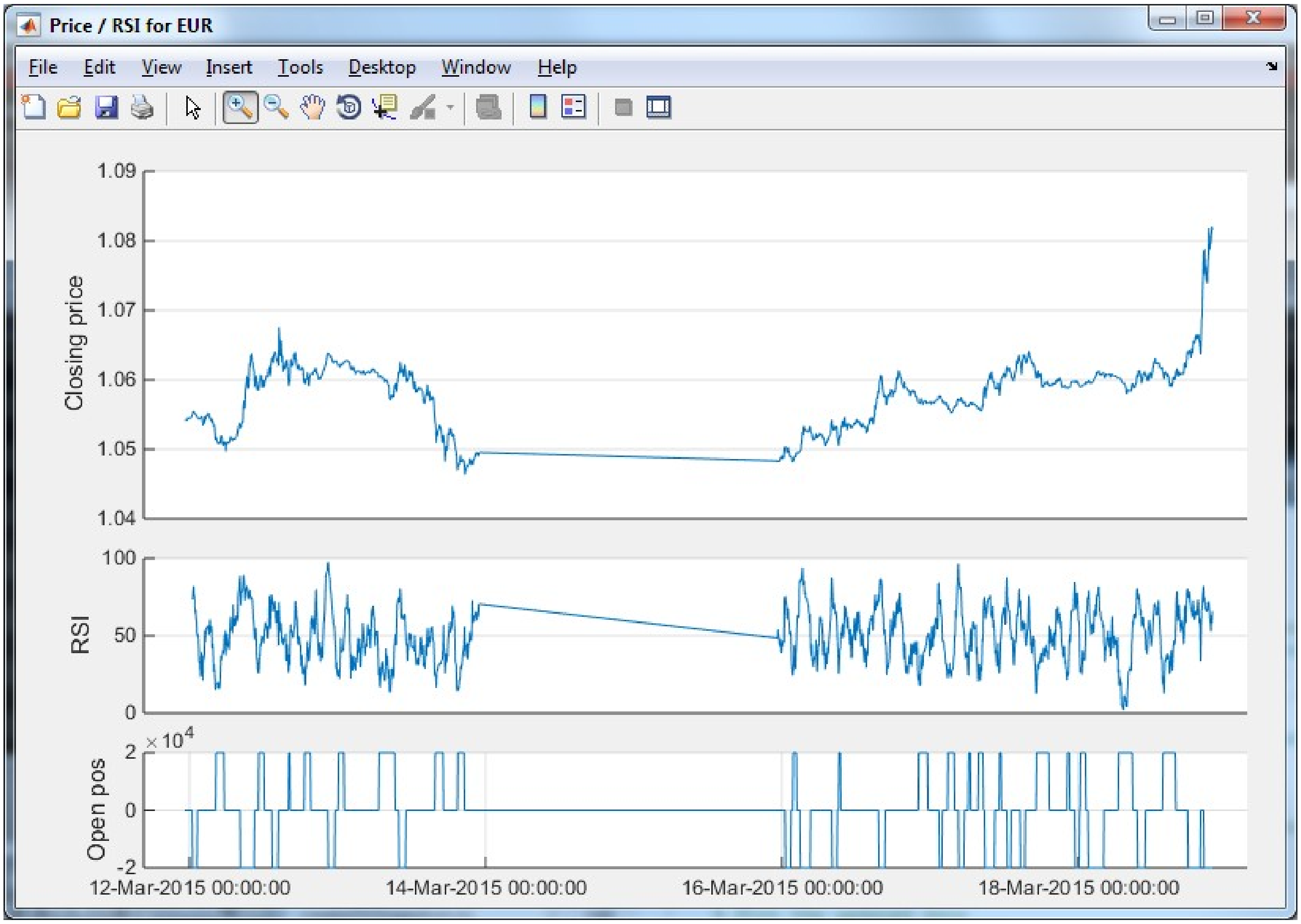
Also note that in the simplistic implementation above, the graphs are cleared and recreated in each timer invocation (every 30 seconds). This implementation can naturally be improved by updating the existing graph's data instead.[170]

[169] See §7.2 for details

[170] Many additional Matlab performance tips can be found in my book “*Accelerating MATLAB Performance*” (CRC Press, 2014)

Finally, here is a sample output from running the program for EUR.USD:

```
>> IBC_AlgoRSI('AccountName','DU12345', 'Symbol','EUR', ...
                'SecType','cash', 'Exchange','idealpro', ...
                'Quantity',20000, 'BarSize','5 mins');

18-Mar-2015 21:45:01 Selling 20000 EUR
18-Mar-2015 21:53:01 Buying 20000 EUR
...
```



Sample plotting output of the IBC_AlgoRSI program

(note the trading break over the weekend of 15 March, 2015)

Contact us to receive consulting assistance in building your trading program, based on IB, Matlab and the **IBC** connector. We never disclose trading algo secrets used by our clients, so we would not be able to provide you with any actual trading algo that was developed for somebody else. But you will benefit from our prior experience in developing dozens of such trading programs, as well as from our top-class expertise in both Matlab and IB's API.

17 Frequently-asked questions (FAQ)

1. Can IBC be used with other brokers?

IBC 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 IBC impose limitations on historical data or streaming quotes?

IBC does not impose any limitations, but the IB server does impose limitations on the frequency of the requests and the amount of returned data. [171] 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; IBC supports whatever subscription your account has, and does not limit the information in any manner.

3. Can I see a demo of IBC?

You can see a webinar showing a demo of IBC (along with presentation slides and the demo's source code). [172] In addition, you are most welcome to request a fully-functional trial version of IBC, which you can use to run the demo yourself, or to test your own trading strategies.

4. How does IBC compare to alternative products?

IBC is currently the market leader in the niche of Matlab-to-IB integration. There are other alternatives available, but IBC provides by far the best functionality, value and cost-effectiveness. A detailed comparison can be found online.[173] You are most welcome to validate all the comparison items when you test IBC's free trial.

5. How do you know that IBC trades $100M daily?

A few of the IBC 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 IBC does not send any information except to IB. IBC is used by hundreds of traders, ranging from individuals, through hedge-funds and even some banks. So the total daily trading volume by all IBC users may possibly be much higher than $100M.

[171] https://interactivebrokers.github.io/tws-api/historical_limitations.html

6. Does IBC send you any information?

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

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

Well, there is never a 100% guarantee. The product is rigorously tested. IBC has been live since 2010 and is actively used by hundreds of users on a daily basis. So far nothing major has been reported. IBC is a very stable and robust product, despite the fact that new functionality is being added on a constant basis. In fact, the professional review by the *Automated Trader* magazine has purposely tried to find limitations and bugs in IBC by specifying invalid parameter combinations etc., and reported that it could not break IBC's robustness.

8. Is IBC 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 IBC's change-log, listed in Appendix B of the IBC User Guide (this document). You can see the very latest updates in the online version of this guide.

9. I saw a nice new feature in the online User Guide – can I get it?

You get the very latest version of IBC, 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 your license immediately. However, the new license term will start from that moment onward (in other words, you will lose the unused portion of your current license). For example, if you purchased a 1-year license on 1/1/2014, it is good until 1/1/2015. If you then choose to renew and get the latest IBC version on 9/9/2014, then your new license will expire on 9/9/2015, so you gain the latest IBC version but you lose about 4 months of your current annual license. The choice is yours.

10. What happens when the license term is over?

When you purchase a IBC license (or renewal), it will work for the full license duration. A short time before the license expires, you will start seeing a notification message in your 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 license expires, IBC will stop working. You can then renew the license for an additional duration, not necessarily the same as the previous duration. If you wish to be independent of annual renewals, you can purchase a discounted multi-year license.

11. Can I transfer my IBC license to another computer?

Yes: you will need to first deactivate IBC on your existing computer
Please contact us at support@Matlab-trader.com to process this request.

12. I have a laptop and desktop – can I use IBC on both?

Yes, but you will need to purchase two separate IBC licenses. IBC's license is tied to a specific computer (unless you purchase a site license).

13. Can IBC be compiled and deployed?

Yes, IBC can indeed be compiled. There is a tricky part where you need to include the *classpath.txt* file in your deployment project – I can help you with this.
You do not need a separate license for the compiled application on your development computer, since this computer is already licensed. However, any other deployed computer will require a separate IBC license, otherwise IBC will not run.
If you wish to deploy IBC on a large scale, to multiple computers, then contact me to discuss alternatives.

14. Is IBC provided in source-code format?

IBC 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 IBC'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[176] 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.

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 IBC?

If feature ABC is supported by IB's API, then it is almost certain that it is also available in IBC. 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 IBC provides. To check whether a specific feature is available in the IB API (and by extension, in IB-Matlab), please refer to IBC's User Guide (this document), IB's online reference,[177] or contact IB customer service.

17. Can you add feature ABC in IBC 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 IBC, at no extra cost. Contact me by email if you have such a request, to get a proposed quote.

18. Can you develop a trading strategy 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 IBC features, strategy development will never be disclosed to others, and will not be integrated in IBC. It will be developed privately for you, and will be kept secret. See §18 below for additional details. If you have such a request, contact me by email to get a proposed quote.

19. Does IBC include back-testing/charting/GUI/data analysis/algo-trading?

No. IBC 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, charting, GUI 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, which will get the data from IB-Matlab, analyze it, and send corresponding orders back to the IB server (again through IBC). I have extensive experience in developing complete backtesting and real-time trading applications - see §18 below for additional detail. I will be happy to either develop a new application based on your specifications, or to integrate IBC into your existing application, under a consulting contract.

20. Does IBC 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.[178] Therefore, it is best to trial IBC and to test your strategies using your personal paper-trading account (which you automatically get with your live trading account).

[177] https://www.interactivebrokers.com/en/index.php?f=5041, https://interactivebrokers.github.io

[178] https://quant.stackexchange.com/questions/8744/what-is-the-difference-between-the-interactive-brokers-demo-account-and-a-personal-paper-trader-account; http://interactivebrokers.com/en/software/am/am/manageaccount/paper_trading_limitations.htm

18 Professional services

We have completed countless life-cycles of software requirements definition, design, development, documentation, integration, testing, deployment, handover, maintenance and support.

Anything developed under private consulting will be kept confidential and will not be disclosed to others. You will retain full IP ownership of anything developed for you.

Contact us by email (support@Matlab-trader.com) if you wish to discuss your needs or to receive a proposal.