As we keep pointing out, automated/algorithmic trading does not automatically imply ultra high frequency trading. Plenty of Automated Trader traders we talk to seem to do just fine deploying automated models that hold positions for periods ranging from seconds to days. At the same time, rather a lot of them appear to use MATLAB and more than a few use Interactive Brokers. All of which rather inevitably led the Wrecking Crew and Automated Trader’s Founder, Andy Webb, to take a look at IB-MATLAB...

First, an important message from the Automated Trader software review team. If you are looking for a high frequency trading interface to Interactive Brokers, please stop reading now; IB-MATLAB isn’t intended for you. Why not flick to Peek Ahead at the end of the magazine and read this issue’s morally improving message from the Editor instead?

IB-MATLAB is intended to provide a quick and simple path to interfacing MATLAB with Interactive Brokers’ IB API (as opposed to the FIX CTCI API that Interactive Brokers also offers). The intention is that IB-MATLAB will be primarily used to send orders and receive order fills - not for receiving real time streaming data from the IB API. For one thing, the IB API has a specified limit of 50 messages per second that would quickly be consumed by real time data from even a modest portfolio of highly active securities. For another, MATLAB is also not really intended for such a purpose and is likely to complain if its interfaces are swamped by a blizzard of market data.

While the IB API offers various means of connecting, including ActiveX, DDE/ActiveX for Excel and C++, IB-MATLAB uses the Interactive Brokers Java API, which has the advantage of being OS independent. So if you run MATLAB on Linux (as a lot of Automated Trader readers do) you can still use IB-MATLAB. Of course there’s nothing to stop you from using the MATLAB Java API directly yourself to interface MATLAB and IB, but the point of IB-MATLAB is to simplify that so you can spend more time testing trading models and less time fixing the plumbing.

Data
While IB-MATLAB isn’t intended for high-volume streaming of real time data, it happily supports one-off requests for current market prices. The syntax is pretty simple and follows the same basic method as most other IB-MATLAB functions, which consist of pairing an input parameter with a user specified parameter value. The generic syntax is:

```
IB_trade('parameter 1', 'parameter value 1', 'parameter 2', 'parameter value 2' ... 'parameter n', 'parameter value n')
```

For example, to get a quick update on Cisco, just type the following into the MATLAB console (or invoke it in a MATLAB function or script):

```
data = IB_trade('action','QUERY', 'symbol','CSCO')
```

This returns a MATLAB struct called ‘data’ that includes items such as bid, ask, open, high, low, close, volume, tick size and time stamp. As a standard MATLAB struct, this means that any of these items can be accessed in MATLAB by using ‘dot referencing’. For example, entering:

```
data.high
```

...in the MATLAB console returns the high of the session:

```
ans =
15.6200
```

It is also possible to retrieve historical data for the current trading session. For example...

```
histdata = IB_trade('action', 'HISTORY_DATA', 'symbol','IBM', 'barSize','1 min', 'useRTH',1);```

...is pretty self explanatory (‘useRTH’ means ‘use regular trading hours’) and returns the following struct:

While the IB API offers various means of connecting, including ActiveX, DDE/ActiveX for Excel and C++, IB-MATLAB uses the Interactive Brokers Java API, which has the advantage of being OS independent. So if you run MATLAB on Linux (as a lot of Automated Trader readers do) you can still use IB-MATLAB. Of course there’s nothing to stop you from using the MATLAB Java API directly yourself to interface MATLAB and IB, but the point of IB-MATLAB is
The Virtue of Simplicity

histdata =
  dateTime: {1x390 cell}
  open: [1x390 double]
  high: [1x390 double]
  low: [1x390 double]
  close: [1x390 double]
  volume: [1x390 double]
  count: [1x390 double]
  WAP: [1x390 double]
  hasGaps: [1x390 logical]

As with the previous example, because a standard MATLAB struct is returned, individual data categories can be dot referenced. For instance, typing...

plot(histdata.close(1,1:390))

...in the console generates a chart of the closing values of each one minute bar as shown in Figure 1. The output to data requests can immediately be reused as inputs to an order. For example, the bid value from the struct that results from the following...

as inputs to an order. For example, the bid value from the struct that results from the following orderStatus:
    whyHeld = whyHeld;

orderStatus:
    filled = 100

orderStatus:
    lastFillPrice = 67.73

orderStatus:
    clientID = 884191

orderStatus:
    whyHeld = whyHeld;

While on the subject of console output, it’s worth pointing out that IB-MATLAB defaults to displaying all messages in the console, which can get a bit overwhelming. Fortunately you can suppress this so only error messages are displayed by setting the ‘msgDisplayLevel’ fieldname to 1.

Though it obviously slows things up a bit, the deliberate pauses built into the underlying IB-MATLAB Java code are probably wise. Our models aren’t usually chasing the same inefficiencies as everyone else so we aren’t that time sensitive. On the other hand, because each trade can be quite complex in terms of leg count, something that fell over easily would cause all sorts of housekeeping and order management issues.

STEVE K (prop trader, European bank): Very simple to install - just drop the IBMatlab.jar file in your static Java classpath and drop two other files into any folder on the MATLAB path and off you go. IB-MATLAB suits the type of multi leg strategy that we normally use on a small star arb portfolio I’m responsible for even though that results in a fair number of orders and amends flying around, the holding period for our trades is usually measured in hours not milliseconds.

Though it obviously slows things up a bit, the deliberate pauses built into the underlying IB-MATLAB Java code are probably wise. Our models aren’t usually chasing the same inefficiencies as everyone else so we aren’t that time sensitive. On the other hand, because each trade can be quite complex in terms of leg count, something that fell over easily would cause all sorts of housekeeping and order management issues.

MARTIN S (programmer/quant, US proprietary trading firm): We already use the Interactive Brokers’ FIX CICI API having migrated to that from the IB API about a year ago. While both these APIs work fine, I wouldn’t describe either as particularly easy to get to grips with. The documentation is good as it goes, but (particularly with the IB API) you tend to get some rather cryptic error messages that don’t help much with the debugging process.

As a result, it would take very little time to get a model deployed in the market using the product. The slight downside is that not every last piece of IB API functionality is accessible, but I’d say that IB-MATLAB covers the bases needed by the vast majority of traders.

Thanks also to the other two members of this issue’s Wrecking Crew. Sorry we couldn’t include your comments, but your frustration at being unable to break anything (other than my desk) means that your views didn’t quite make it past the Editor’s new proficiency checker. There’s always next time...

Pumping trades

In order to test order throughput, we assembled the following rather pointless MATLAB function:

```matlab
function IB_testtradel
    stk = load('dow_stks.mat');
    for i = 1:100
        rndsec = randi(15,1);
        data = IB_trade('action','QUERY',stk.
                        stk{rndsec,1});
        orderId = IB_trade('action','BUY',data.
                           'symbol',stk{ rndsec,1});
        data = IB_trade('action','BUY',data.
                           'symbol',stk.
                           stk{rndsec,1},
                           'quantity',100,
                           'limitPrice',data.
                           'symbol',stk.
                           stk{rndsec,1},
                           'quantity',100,
                           'limitPrice',data.
                           'symbol',stk.
                           stk{rndsec,1},
                           'quantity',100,
                           'limitPrice',data.
                           'symbol',stk.
                           stk{rndsec,1},
                           'quantity',100,
                           'limitPrice',data.
                           'symbol',stk.
                           stk{rndsec,1},
                           'quantity',100,
                           'limitPrice',data.
                           'symbol',stk.
                           stk{rndsec,1},
                           'quantity',100,
                           'limitPrice',data.
                       end
end
```

This opens and loads the contents of a ‘mat’ file into a two column struct called ‘stk’ (see Figure 2). It then loops 100 times, while on each iteration it generates a new random integer which it uses as a row reference to select a stock symbol from the first column of the ‘stk’ struct. It then retrieves a current data snapshot for the symbol and uses that as the basis for an attempt to buy 100 shares on the bid. It then pauses for one second before starting the next iteration.
Rather predictably, this quickly filled up the pending orders tab of the Interactive Brokers’ Trader Workstation (see Fig 3). However, it didn’t cause any stability problems. So just for the sheer irresponsible hell of it, we removed the pause, and were disappointed when it didn’t appear to make any difference. We then of course realised there was another inherent pause in the code, in that the order execution line would not run until it had a populated ‘data’ struct to work with.

So in a concerted attempt to break something the interests of academic research, we did away with the data gathering step and switched to market orders and removed the pause:

```matlab
function IB_testtrade2
stk = load('dow_stks.mat');
for i = 1:100
    rndsec = randi(15,1);
    % Long leg
    orderId = IB_trade('action','BUY', 'symbol',stk.stk{rndsec,1}, 'quantity',10000, 'type','MKT');
    % Short leg
    orderId = IB_trade('action','SELL', 'symbol',stk.stk{rndsec,2}, 'quantity',10000, 'type','MKT');
end
end
```

Nope - rock solid.

So in a final attempt at chaos, we added a short sale to the code (symbol selected from column two of Figure 2) and also jacked up the order quantity for buys and short sells, so we would have multiple partial fill messages coming back through the interface:

```matlab
% Long leg
for i = 1:100
    orderId = IB_trade('action','BUY', 'symbol',stk.stk{rndsec,1}, 'quantity',10000, 'type','MKT');
end
```

By the way, we did have to actually think about it for a while, as we were sure that the fact that the Wrecking Crew couldn’t break the code immediately should have made the code much more stable. However, we were wrong.

Stop Press: Update on Streaming Quotes

The version of IB-MATLAB reviewed here does not support streaming quotes, so data requests have to be made individually. However, just as we went to press, we received an update from Yair Altman that a new version of IB-MATLAB was currently under development that does support streaming quotes.

Obviously the number of instruments for which you will be able to retrieve real time quotes will depend upon which instruments you choose. Even a small portfolio of stocks such as Google and IBM will quickly mop up the IB API’s published capacity of 50 messages per second. If on the other hand you specialise in exotic African stocks that print one trade a month, you should have plenty of elbow room.

Other factors to bear in mind will be network bandwidth, computer speed and of course what level of data you have subscribed for with your IB account.

But nothing - not even a flicker. Just an orderly procession of orders going out and an orderly procession of fills coming back; all very interesting.

Readers of Automated Trader will by now be aware that the Wrecking Crew’s approach to software testing is probably best termed ‘robust’. If they can’t make something crash by doing something completely irrational and dangerous, they tend to get a little moody (i.e. start kicking the office furniture to matchsticks). IB-MATLAB’s wholly unreasonable refusal to make either MATLAB or IB’s Trader Workstation fall over therefore irked them not a little.

However, as an alternative to mindless violence, the more cerebral element of the Wrecking Crew (total headcount: 0.01) suggested speaking to IB-MATLAB’s developer, Yair Altman to see if he could shed any light on this irritating stability. Mr Altman has clearly met people like the Wrecking Crew before - he also evidently gets a single message containing the entire response in return. Therefore the IB-MATLAB code contains a number of deliberate delays to allow all the various message fragments time to arrive, after which they are consolidated and passed back to MATLAB.

Needless to say, certain members of the Wrecking Crew felt that this wasn’t really playing the game, but even they grudgingly admitted that trading software that doesn’t turn up its toes at the drop of a hat is actually not a bad idea...

Orders - coverage and tracking

Even the more jaded members of the Wrecking Crew admitted (through gritted teeth) that IB-MATLAB’s syntax is intuitive. It says something that the explanation of this syntax in the help file consists of less than 100 lines. A nice touch that definitely assists this brevity is the way that it leverages existing Interactive Brokers API syntax. For example, its implementation of callback functions is...
While the default format for orders shown in the body of the main review work fine for US securities, the Wrecking Crew developed some anger management issues when they tried to replace their testing on European stocks. The original idea was to put together a pairs portfolio of NYSE Euronext stocks and then trigger buy and short sell orders at random time points for each pair.

We started by testing simple data retrieval for a single stock - AkzoNobel (listed as symbol AKZ in the IB symbol directory) which primarily trades in Amsterdam on the AEB. In line with our successful retrieval of Google we entered:

\[
\text{data} = \text{IB\_trade('action', 'BUY', 'symbol', 'AKZ')}
\]

This will invoke the two callback functions with the separate OrderStatus and OpenOrders events and the corresponding EventData can be searched by using the returned orderId value. One of the partial fill messages returned by this method for the ArcelorMittal order above is shown below.

\[
\text{orderId} = 89900419
\]

\[
\text{orderStatus: orderStatus = 'Submitted'
status = 'Submitted'
remaining = 2539
avgFillPrice = 22.67
permid = 13324831}
\]

Among many other things, the range of callback functions available in IB-MATLAB make keeping track of submitted orders pretty straightforward. For example, the following code tags a couple of callback functions onto our eventually (see box ‘API Tricks of the Trade: Non-US instruments’) successful order for ArcelorMittal.

\[
\text{[orderId, ibConnectionObject] = IB\_trade('action', 'BUY', 'symbol','MT', 'exchange','AEB', 'quantity',10000, 'type','STP', 'currency','EUR','sectype','STK', 'parentId' = 0, 'callbackOpenOrder', @OpenOrdersFn, 'callbackOrderStatus', @OrderStatusFn);
}
\]

This returns a MATLAB struct with each element within the struct containing the details of one security in the portfolio, as shown in Figure 4 for a position in Caterpillar.

data = IB\_trade('action', 'BUY', 'symbol', 'MT', 'exchange', 'AEB', 'currency', 'EUR', 'sectype', 'STK', 'parentId' = 0, 'callbackOpenOrder', @OpenOrdersFn, 'callbackOrderStatus', @OrderStatusFn);

The event data triggered in response to ‘CallbackOpenOrder’ and ‘CallbackOrderStatus’ can be captured and manipulated by writing two functions (OpenOrdersFn and OrderStatusFn) in MATLAB to handle it. This can be achieved using the orderId and the ibConnectionObject returned by the IB-MATLAB after either typing the following at the MATLAB console or including it in a function:

ibConnectionObject.reqOpenOrders

This returns a MATLAB struct with each element within the struct containing the details of one security in the portfolio, as shown in Figure 4 for a position in Caterpillar.

\[
\text{data} = \text{IB\_trade('action', 'BUY', 'symbol', 'MT', 'exchange', 'AEB', 'quantity', 10000, 'type', 'STP', 'currency', 'EUR', 'sectype', 'STK', 'parentId' = 0, 'callbackOpenOrder', @OpenOrdersFn, 'callbackOrderStatus', @OrderStatusFn);
\]

Hmmm. The confusion only became worse when we tried a simple market order for another AEB stock - ArcelorMittal (symbol MT). No error message in the console this time, but a pop up box appeared warning that the order would not be submitted until the start of US trading hours. Why? A bit of digging quickly revealed that ArcelorMittal also trades on NYSE, even though its primary listing is on the AEB. But the symbols for both incarnations of the stock were identical - MT. Attempts to place a manual order or view ArcelorMittal in the Trader Workstation showed the various possible alternatives that could be manually selected (see Figure 5) but how could we replicate this selection of the AEB version of the stock via the API? Our assumption that the default would be the stock’s primary listed venue was clearly wrong.

So we tried explicitly setting the ‘exchange’ field in our API order with:

\[
\text{orderId = IB\_trade('action', 'BUY', 'symbol', 'MT', 'exchange', 'AEB', 'quantity', 10000, 'type', 'STP');}
\]

So yes, we like it - a lot.