COVER STORY, surely?

You're right, but leave it.

Mashup!

Well yes OK, we know we're rather stretching the definition of "mashup" here, but it has rather more punch as a title than "we attempt to cobble together some random bits of software into a trading application/model doing a spot of (re)reviewing along the way". Though that's probably a more accurate description of what the Wrecking Crew and Automated Trader's Founder, Andy Webb, have actually been up to for the last month or two.

Yeah right.

In that perfect trading world in which we all exist, everybody has a single homogenous environment that handles everything – development, testing, live deployment, the lot. There's never even the tiniest thing that your perfect environment cannot handle. Code never has to be ported to another application, everything is seamless so you don't even have to know what API stands for – let alone use one. Nothing ever goes wrong, the word "exception" is never heard (nor are any of the other words that feature so regularly in the Wrecking Crew's vocabulary), optimisations take seconds, if that, and trade execution nanoseconds.

Over here in the real world (and believe us, a world inhabited by the Wrecking Crew can't get much more real) budgets get cut, heads of IT insist that Vista is the One True Way, and not everybody had Kernighan and Ritchie as their bedtime reading at the age of two. People want the flexibility to test and deploy trade ideas quickly without having to spend a sovereign defence budget on technology. Particularly among smaller hedge funds and proprietary trading groups, that often requires plugging various separate applications together to achieve the desired functionality, ease of use and time to market.

Old friends

While things have improved dramatically in recent years, the phrase "plugging various separate applications together" in the previous paragraph can still easily end up as a synonym for "hours of frustration and nothing working". Given that breaking things is the Wrecking Crew's forte and that trying to get multiple things to work together implies more wreckage and greater job satisfaction, we thought that in this issue we might try this "plugging together" out and revisit a few old friends along the way (while also looking at some new technology).

So we started from the premise of doing our preliminary modelling in MATLAB and deploying any resulting models via IB-MATLAB to Interactive Brokers' trading API. However, just to make things a little more demanding, some of the modelling involved sufficiently hefty workloads to justify giving our test workstation's CPUs a helping hand by using its installed GPUs – courtesy of AccelerEyes Jacket – to rev up our MATLAB number crunching. While we could have used Interactive Brokers' historical data to feed this set-up, this has some limitations regarding the amount of data available. We therefore decided to take CQG's API for a spin by hooking it up to MATLAB to provide historical data.

We could have left it there, with anyone wanting to monitor automated trading activity using Interactive Brokers' Trader Workstation (TWS). But by this stage the Wrecking Crew had its collective bit between its teeth and it was decided that we would knock up some form of monitoring station in Excel that would ▶

be fed via its COM interface with position info from MATLAB (coming via IB-MATLAB from IB) and via CQG's RTD server with real time and historical data. The extremist wing of the Wrecking Crew also wanted to add a manual trading interface to Excel on the premise that any trader monitoring automated models would soon become bored playing park keeper and would want to punt about a bit on their own. Fortunately reason prevailed (up to a point) and the moderate majority headed that one off at the pass.

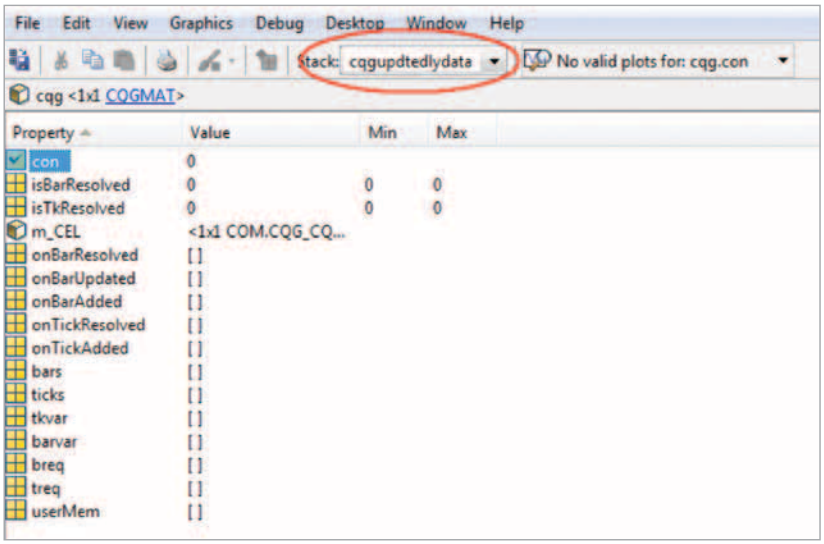


Figure 1

Unfortunately this proved to be an opening for the Crew's oldest member. Horace (see Automated Trader Q2 2010) has been grumbling because recent reviews haven't given his natural talent for distrust much of an outlet – no compliance manuals to read, no security audits to conduct. To get some peace and quiet, a further act of lunacy was added to the brew: instead of having MATLAB send the position info direct to Excel, it would send it to a local version of Excel in our server room, which would be automatically checked for updates by an instance of Excel running on the park keeper's/trader's workstation – in Italy. Or to be more precise, near the top of a mountain in Italy with only a restricted capacity community wireless internet link. Sigh...

This immediately raised the issue of security, which in turn prompted a massive row about how to protect the link between the two instances of Excel. Those of us looking for a quiet life (and to finish this review before 2014) opted for LogMeIn Hamachi, the hard geekcore wanted to use OpenVPN (and wanted to install TomatoVPN firmware on all our routers), but Horace insisted that we should use a Billion BiGuard S20 SSL VPN box that he had found in our spare parts bin covered in cobwebs and dust. (The fact that we didn't have and could no longer buy the security tokens to generate the one time passwords to use with this didn't deter him.)

Having assembled this recipe for catastrophe, off we set...

CQG => MATLAB

As the "consensus" was that we should try and put together some statarb strategies for equities, our first step was to get some historical equity data from CQG into MATLAB. From a programmer's perspective, CQG's API is thoroughly documented (although some less experienced programmers might struggle with figuring out how it all fits together). Fortunately, CQG provides a decent selection of MATLAB code samples that illustrate how to connect everything up and actually get historical and real time data through the API. (Although CQG's API supports a lot more than just real time and historical data – including account, position and order management – it doesn't as yet provide execution in equity markets so we were opting to use IB for that part of the brew.)

While the CQG examples were perfectly adequate for our purposes, we actually chose to take advantage of MATLAB's ability to define custom object oriented classes. One of our former Wrecking Crew members, Dr Yang Wang (see Automated Trader Q3 2010 and Q2 2011) had already knocked up a class packed full of overloads that let us explore just about every possible way of importing CQG data with negligible effort. We took full advantage...
In doing so, we came across a curious problem that at the time of going to press we still haven't quite got to the bottom of – though we have found a simple an effective workaround. We started out by invoking our object based on Dr Wang's class from within a function we had written to collect data from CQG, validate it for errors and to append it to an existing database. Big mistake. A frustrating few hours then ensued as the data we had just requested from our function kept disappearing. We created our object within the function ("CQGMAT" being the class name) with:

```
CQGMAT.new('cqg');
```

We could then see (see Figure 1) that our empty object had been successfully created in the function's

workspace (highlighted in red in Figure 1) from a quick look MATLAB's Variable Editor.

Running the next line of our function:

```
cqg=cqg.
QuickBar(symbols{j,1},'D',
daysback);
```

...successfully populated the necessary input arguments (see Figure 2).

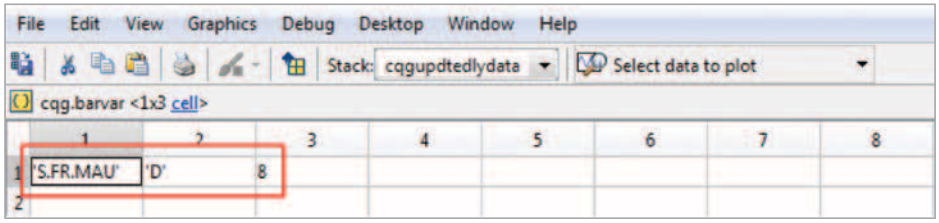


Figure 2

Then we ran the next line of the function:

```
cqg=cqg.Start();
```

...which fired up the CQG interface and passes the arguments highlighted in Figure 2 for the desired stock (in this case Maurel & Prom), the time frame of the bars requested (daily) and the number of bars required (8).

Then nothing. No data ever came back!

In MATLAB an object can obviously only remain in scope for a function while the invoking function is running (assuming the object hasn't been explicitly made global). As soon as the function exits, the object disappears. But this wasn't our problem – the calling function had definitely not exited. We "F11d" through the code above and could see the object being created in the MATLAB base workspace (complete with the price bars we wanted) but the version of the object in the function's workspace remained forever bar-less.

In the end we lost patience and simply converted the function to script (which unlike a standard function can access the MATLAB base workspace) and retrieved our data that way (we could alternatively have left it as a function and used MATLAB's evalin.m function to access the data). Hopefully when Dr W returns from Chinese New Year we'll get to the bottom of the problem and publish an update.

This was clearly not a problem related to CQG and when we briefly reverted to using the sample code provided by CQG while trying to resolve our disappearing data problem above we found the API worked perfectly. Although we were retrieving data for historical testing via the CQG API, we also did some dabbling with real time updates to try and get a feel for its capacity. Our portfolio of pairs was based upon a total of 175 securities, of which more than half were large caps with pretty frequent updates. While we didn't have any spectacularly fast markets during the test period to really stress test things, we didn't have any problems in terms of bottlenecks. Just for the hell of it we also created some custom studies in CQG and pulled the values for those into MATLAB without any problems.

However, as part of the CQG => MATLAB process, we did come across one other "interesting" potential snag – exchanges. Some exchanges have become extremely twitchy about data vendors allowing their clients to access exchange data through an API (the potential for "informal" redistribution of this data and loss of revenue for the exchange presumably being the issue at point). While CQG were extremely helpful with the authorisation process associated with this (major hat tips here to Alice Morrison and Brian Vancil of CQG – thanks both), certain exchanges didn't exactly hurry. One European exchange was first approached on December 1st 2011 about API access to their data (and then politely and regularly reminded regularly thereafter) but took until January 20th 2012 to actually authorise data access. So the lesson appears to be that even if you can see an exchange's data within your data vendor's application, don't assume that accessing that same data through an API is a trivial matter – start the paperwork and nagging process as early as possible!

First steps

In our quest to make things as difficult as possible, it was decided that any testing of our initial model (which was for pairs) should include cointegration tests across multiple time windows for each pair. We had over 1000 potential pairs to test and we ended up deciding to cointegration test 350 time windows for each pair. We tried this using both MATLAB's own egcitest.m function for the Engle-Granger test from its Econometrics Toolbox, as well as the equivalent cadf.m function provided in the freely available Spatial Econometrics toolbox.

We started by running two instances of our code on two machines – one with the code incorporating the ▶

MATLAB version of the Engle-Granger test and one incorporating the Spatial Econometrics' version. (We rather cut corners before doing this by only testing that both stocks in each pair had unit roots once for each pair across all the available data, using MATLAB's version of the augmented Dickey-Fuller test.)

We set the tests running, but after about forty minutes lost interest and went to find some beer...

Upon our return we found that both tests had finished at about the same time (around the seventy minute mark). Wanting to cut this down significantly, we decided to fire up MATLAB's Parallel Computing Toolbox (PCT), which would let us distribute the task across all four of the CPU cores available in each test workstation. That knocked the test times down to about 22 minutes by replacing our original for loops with the "parfor" loops available in the PCT.

The hard geekcore had originally wanted to use the Johansen cointegration test framework (also available in both MATLAB's Econometrics toolbox and the Spatial Econometrics toolbox), but the rest of the team had a nasty feeling where that might lead to when we tried to cut execution times, so we managed to kill that idea off. How right we were, as now the clamour was to cut execution times further by using the CUDA-enabled NVIDIA Tesla C1060s GPUs (see www.nvidia.com/object/cuda_home_new.html) in the workstations in conjunction with AccelerEyes Jacket.

MATLAB + JACKET

Since we originally reviewed it in early 2010, Jacket has changed beyond recognition. The number of MATLAB functions it supports has ballooned and among many other things now also includes support for doubles using left matrix divide (mldivide or \ in MATLAB), which was something we were whingeing about in our original review. Another significant change is that are you are no longer obliged to have the MATLAB PCT in order to run Jacket across multiple GPUs.

We had already spent a fair bit of time trying to "vectorise" our MATLAB code in order to reduce the number of for loops (which – depending on the exact circumstances – can significantly slow code execution in MATLAB). However, we were still left with a few loops that we couldn't eradicate, so we thought we would use Jacket's "gfor/gend" loops for the largest and innermost of these. Unlike conventional CPU-based looping, Jacket's gfor/gend construct runs all iterations of a loop simultaneously. It achieves this by "tiling out" the values of all loop iterations and then calculating the values of all tiles at the same time using the individual cores of a CUDA-enabled GPU. However, while this has the potential to massively reduce computation times, the gfor loop does have a few limitations some of which would require changes to the functions we were using.

A case in point relates to "if" statements, which are not officially permitted at all inside gfor loops, because they implicitly pull data back to the CPU from the GPU. This could have been a major hassle for us in our testing, as both the MATLAB and Spatial Econometrics versions of the Engle-Granger test are littered with conditional statements – as are many of the other functions that they call.

Fortunately, this wasn't as much of a problem as we initially expected. Firstly, we found that it appears that you can get away with "if" statements in gfor loops in certain circumstances. More specifically, if the "if" statement was in a function called by the gfor loop, only involved MATLAB single or double data types and was relatively straightforward, such as:

```
if (p < -1);
    error('p cannot be < -1 in
    gcadf');
end;
```

...then no errors were generated and the code would run OK. However, if the "if" statement involved Jacket-specific data types (such as gdouble or gsingle) then an error would throw.

```
if((abs(adf) > abs(crit(critval))))
      H = 1;
      else
      H = 0;
      end
```

NO!

AccelerEyes offer a number of suggested rather elegant workarounds for this including one that works by expressing the conditional statement as a multiplication by logical values (see wiki.accelereyes.com/wiki/index.php/GFOR_Usage#No_logical_indexing). We took the rather more crude approach of, wherever possible, retrieving the values that would have been involved in an "if" statement back to the calling function. So our original if statement above ended up as:

```
resultcoint = abs(adf) -
abs(crit(critval));
```

YES!

...in the function inside the gfor loop. Then when we

had built the complete table of resultcoint values and had completed the gfor/gend loop we simply evaluated them all with:

```
resultcoint(resultcoint <=0)=0;
resultcoint(resultcoint >0)=1;
```

Hardly rocket science (to put it mildly) but it worked. Incidentally, we couldn't have used the above line within the loop because apart from if statements, gfor/gend loops don't allow logical indexing.

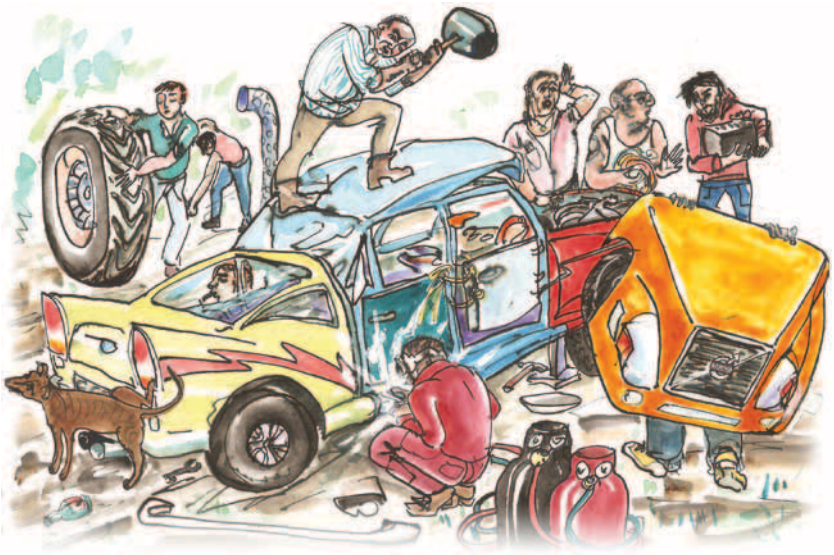
Another thing not permitted in gfor/gend loops is using the loop iterator in colon expressions, which is a popular construct for many MATLAB users. Again, there are ways around this that aren't too demanding, the most straightforward of which is pre-calculate any offsets, e.g.:

```
idx = gsingle(0:350); rc
= gnan(1,datacols); rp =
gnan(1,datacols); ra =
gnan(1,datacols);
```

```
gfor kdx = 1:(rows(xdata)-350)
     [tmprc, tmprp] = gcadf(yda
     ta(kdx+idx,:),xdata(kdx+i
     dx,:),0,1,2); rc(1,kdx) =
     tmprc; rp(1,kdx) = tmprp;
gend
```

Furthermore, if you create the offset vector `(idx = gsingle(0:350);)` as a Jacket data type such as gsingle, this "marks" it for the GPU which delivers a further speed boost.

On that point, the difference Jacket made to our code performance was pretty dramatic. Using one GPU on our single Jacket gfor/gend loop cut the total function execution time from the 22 minutes achieved by MATLAB's PCT "parfor" loops, to just over five minutes. Impressive, but then we remembered that Jacket now no longer depends on MATLAB's PCT in order to run calculations across multiple GPUs. The temptation to see whether we could further improve things by using all three of the GPUs in our workstation became irresistible...



Wrecking Crew mashing up

However, after reading the online docs and arguing for half an hour we couldn't figure out the right syntax to accomplish this, so posted a query on the AccelerEyes forum. Within less than eight hours we had the answer from Pavan Yalamanchili, one of AccelerEyes' core developers (whose tag on the forum "If it is not broken, you have not tried hard enough" has now been adopted as the new Wrecking Crew mission statement). After staring blankly at the sample code provided by Pavan for about an hour, we actually got round to reading the rest of his post properly and finally appreciated the massive significance of his phrase: "multiple gpus can be run concurrently, even if their jobs are assigned serially". In our simple serial-minded way we hadn't previously been able to figure out how GPU two could start work before GPU one had finished (ditto for GPU three/two).

Collective senior moment finally over, we were able to run our code across all 3 GPUs – completing in one minute fifty-two seconds.

butchery

It has to be said that we had to do a fair bit of ~~hacking~~ of the original Spatial Econometrics functions to produce our own "Jacketised" versions that would run satisfactorily. Including any sub-functions called, this took us probably half a day in total, which given the speed improvement and the opportunity for reusing those functions, we felt was a good return on effort expended.

We also tried to do the same with the MATLAB egcitest.m function but kept running into sub-functions that it calls – such as qr.m (orthogonal-triangular decomposition) – which Jacket doesn't yet support. With sufficient diligence we could probably have found a way round these obstacles, but the alternative of recasting a few "if" statements in the Spatial Econometrics functions was the sort of path of least resistance that appeals to the review team.

MATLAB => IB-MATLAB => IB

When we reviewed IB-MATLAB in Q3 2011, one of the things we remarked upon was its stability. From conversations with developer Yair Altman, it became clear that there was quite a lot going on under the hood in IB-MATLAB to manage message flow more efficiently and prevent lockups or crashes.

We were therefore particularly keen to take a look at the latest version of IB-MATLAB as it now incorporates streaming quote functionality that was not available at the time of our original review. In view of the published capacity of 50 messages per second of the "standard" IB API (the FIX version of the API has a higher capacity) we were intrigued to see how IB-MATLAB would deal with unreasonable users (such as?) who tried to run streaming quotes on a large number of highly active issues.

One thing that helps in this endeavour is that IB doesn't appear to transmit flat price ticks (trades at the same price as the previous trade). We certainly didn't see any when we ran the following:

```
load('dow_stks.mat');
tic;
for i = 1:rows(stk)
    stk{i,1} = IB_trade('action',
    'Query','symbol',
    stk{i,3}, 'QuotesNumber',
    inf,'QuotesBufferSize', 1000
    );
end
toc;
pause(10);
tic;
for i = 1:rows(stk)
    stk{i,2} = IB_trade('action',
    'Query','symbol', stk{i,3},
    'QuotesNumber', -1);
end
toc;
```

The first section of code requests continuous streaming quotes (by using the MATLAB inf value as an input) for all 30 stocks in the Dow and specifies a quotes buffer capable of holding 1000 ticks. After a ten second pause, the second section of code accesses those ticks (bids, asks, trades and timestamps) which have ▶

been stored in a MATLAB struct but by specifying `'QuotesNumber', -1` does not interrupt the ongoing collection of ticks. Using this mechanism, it is straightforward to write ticks to disk, build time or tick bars or do pretty much anything else with the raw data.

Although it was hardly what you'd describe as the cutting edge of high frequency, it didn't take very long to get the process going. The output below was generated the MATLAB command prompt as a result of running the code above. To register the initial requests and subscribe for the 30 stocks via the IB API took just under 10 seconds. Writing the buffered data to a cell array of structs was obviously a lot faster.

```
>> DowStocksStreamQuoteTest

connecting to IB...

Server Version:59

TWS Time at connection:20120125
17:29:11 GMT

 [API.msg2] Market data farm
 connection is OK:eurofarm {-1,
 2104}

 [API.msg2] Market data farm
 connection is OK:cashfarm {-1,
 2104}

 [API.msg2] Market data farm
 connection is OK:usfarm {-1,
 2104}

Elapsed time is 9.967258 seconds.

Elapsed time is 0.061849 seconds.
```

An important point about IB-MATLAB is that it uses MATLAB's Java interface and not COM/ActiveX. Apart from allowing it to run on both Windows and non-Windows platforms, this has important benefits when it comes to processing data. If you have a function running in MATLAB when IB's TWS fires an event via a COM interface then MATLAB will not process the event until after the function has completed, which can result in dropped ticks and subsequent calculations being applied to the wrong tick. By contrast, when using the Java API, any fired event is automatically caught by Java in the background to ensure processing of the correct price tick.

From a statarb or even just a general short selling perspective one of the most useful aspects of the new IB-MATLAB tick functionality is its ability to query multiple other data fields via the IB API. IB-MATLAB can request what IB terms "Generic Tick Types", which includes data such as stock option volume, option open interest and historical/implied vol by specifying the relevant "Integer ID Value" in any data request. However, from our perspective, the most useful Integer ID Value was 236 as this returns a value that indicates whether a stock is shortable. Furthermore, the value also indicates how much inventory IB has available for shorting. So (among other things removed to save space) this:

```
>> data = IB_
trade('action','QUERY', 'symbol',
'GOOG','GenericTickList','236')

...returns

data =
            reqId: 147771273
              ...
        shortable: 3
```

A shortable value of 3 (because it is greater than 2.5) indicates that IB has at least 1000 shares of inventory available for a short sale. A value between 1.5 and 2.5 would indicate a stock is available for short sale if shares can be located, while a value between 0.5 and 1.5 means the stock is not shortable. Granted, this information is nowhere near as comprehensive as IB's Short Stock Availability Tool, which provides far more granular data on both shortable individual stocks and a portfolio uploaded as a file, but for smaller trades it's still extremely useful. The only downside is that we could only get generic tick 236 to work for US stocks. In view of the various and ongoing regulatory changes regarding shorting in Europe, we're obviously following up with IB on this to double check whether this was just an error on our part or simply not available for European stocks. Expect an update shortly.

All told, we regarded the enhancements to IB-MATLAB since we last reviewed it as significant. The order submission process was rock solid as before, but the new capabilities really open up the possibilities – especially for trading that is analytically intensive but not high frequency. We were able to deploy multiple models in real time to IB's simulated (nope – the editorial budget still doesn't stretch to funding an account) trading platform without any difficulties or glitches.

MATLAB => Excel, CQG => Excel and Excel => Excel Up a Mountain

It has to be said that this leg of the review project was frankly a bit of a luxury. Interactive Brokers' TWS (see Figures 3a, 3b and 3c) provides pretty much every

possible tool necessary to monitor trading activity. However, the Wrecking Crew were adamant that the park keeper/trader would also want to have access to the parameters of any error correction model we were using, plus the ability to track the spread and following trading signals in real time. Since none of the statarb models we had cobbled together in MATLAB were remotely high frequency there was some degree of realism to this. We therefore used MATLAB to shunt error correction model updates and other values both daily and every ten minutes into Excel, while real time market data that could be used in conjunction with these values was fed into Excel via CQG's RTD server.

MATLAB => Excel

When it came to moving data from MATLAB into CQG, we considered a couple of options. MATLAB has the xlswrite.m function as a quick and straightforward way to shunt data into Excel. However, while it is perfectly effective for small/simple exports, it can become slow when writing to multiple worksheets as it has to open/close Excel every time it writes to a separate sheet. By contrast, establishing a COM/ActiveX connection to Excel allows for far more granular control. One quick way to get an inkling of just how much control is to start the ActiveX server in MATLAB, with (for example):

```
excel = actxserver ('Excel.
Application');
```

...then type:

```
excel.invoke;
```

at the MATLAB command prompt. This returns a list of more than 50 available methods, while typing:

```
excel.get;
```

...returns a list of nearly 200 available properties.

Apart from relatively mundane things that the xlswrite.m function can do anyway (such as write data to worksheet ranges) using a COM/ActiveX connection to Excel allows you to do things such as run macros from within MATLAB. We used this capability to auto generate charts of some of the various models – for example, Figure 4 shows the historic spread of a simple pairs model applied to two European oil stocks – as well as to run an Excel AutoFilter so that the park keeper/trader would only see tradable (rather than all possible) instrument combinations or those in which a position was already open.

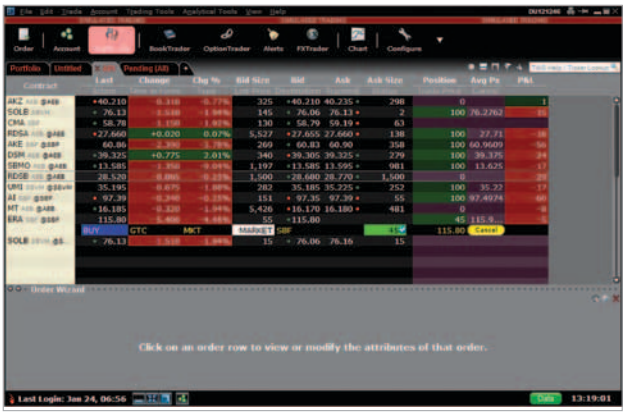


Figure 3a

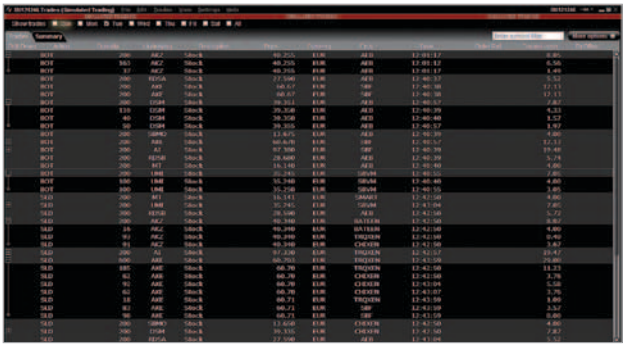


Figure 3b

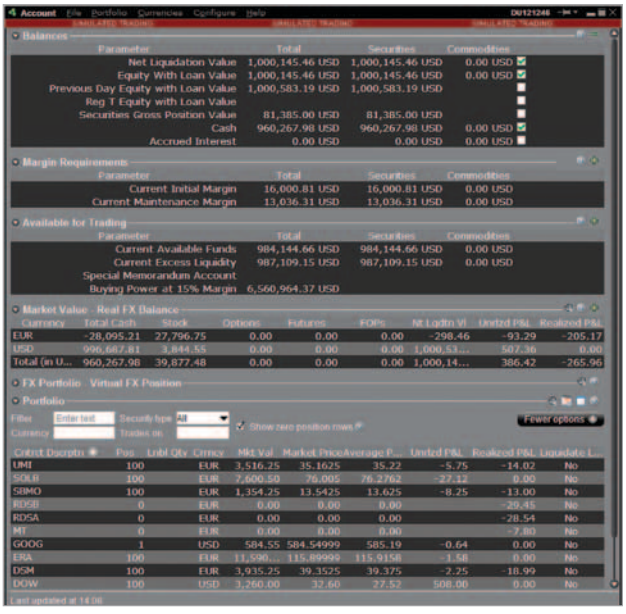


Figure 3c

This was an astonishingly painless process. We were able to do both daily and intraday updates of the models, write the parameters to Excel, run a few macros and close Excel in a matter of seconds. Well, to be strictly accurate, it was sometimes more than a few seconds when we ran up against the old problem we encountered in our original MATLAB review nearly three years ago, which wasn't actually a MATLAB issue at all. Whether you are using MATLAB's xlswrite.m function (or for that matter its xlsread.m or xlsinfo.m cousins) or your

Previous Reviews and Links

MATLAB

www.fa5t.net/rw

www.fa5t.net/rx

AccelerEyes Jacket

www.fa5t.net/ry

www.fa5t.net/rz

CQG API

www.fa5t.net/s0

www.fa5t.net/s1

IB-MATLAB

www.fa5t.net/s2

www.fa5t.net/s3

Interactive Brokers API

www.fa5t.net/s4

own COM/ActiveX connection you can sometimes find that deleting the connection doesn't actually delete it.

The underlying problem is that other applications may have silently established connections to Excel (Google Desktop Search sometimes used to be a sinner here) and won't let go. Fortunately, it's possible to run system-wide commands at the MATLAB command prompt or in MATLAB functions/scripts, so if you want to terminate Excel with extreme prejudice:

```
system('taskkill /F /
IM EXCEL.EXE');
```

...seems to do the trick pretty reliably.

Just for the hell of it, we did one additional test that wasn't in our original game plan, which was to take data received into MATLAB from IB via IB-MATLAB and immediately retransmit it to Excel. We opted to do this on a five minute batch basis, using a custom MATLAB function to compile the raw trade ticks into tick (as opposed to time) bars. Again, we were fortunate in having no difficulty in doing this, but while automated, the process could undoubtedly have been slicker. As a result, we're planning to run a review of KaiTrade's generic K2 RTD Handler in our next issue that will allow us to stream the quotes from IB to Excel in real time.

CQG => Excel

Getting real time data into our park keeper/trader's instance of Excel was similarly straightforward. Once CQG's Integrated Client application is started, its RTD server just works – period. OK, we could have done with fewer double quote marks in the formulas making us cross eyed:

```
RTD("cqg.
rtd",,"StudyData","S.
FR.MAU","Bar",,"Close
","D",,"primaryOnly",
,,,"T")
```

...but all told this was pretty painless. While Microsoft Real-Time Data Components have been around for nearly ten years and are well-established, we were still intrigued to see how far they could be pushed. Painful memories of how RTD's DDE forebear used to keel over remarkably easily (or just drop data all over the place if the going got tough) made us wonder what was practically possible in terms of throughput. In the case of CQG's implementation, the short answer seems to be "quite a lot". For the hell of it, we dumped RTD links for more than 3000 NASDAQ/NYSE stocks into an Excel spreadsheet and sat back. CQG's Integrated Client application started using rather more memory and CPU cycles but that was about all – no drama, loads of updates.

Another hat tip at this point to CQG's Thom Hartle, who seems to be making pushing RTD/Excel to the limit and beyond his life's work. Thom kindly shared some of his handiwork with us, from which we plagiarised ~~several~~ most of our RTD worksheet formulas. (Figure 5 illustrates what's possible with CQG + RTD + Excel + Thom. We didn't even attempt to emulate this, so he'll probably weep when he sees our feeble attempts in Figure 4.)

Excel => Excel Up a Mountain

That finally brings us to the connection between the two instances of Excel. Much to our amazement, the Billion S20 proved far easier to configure than we expected. While the S20 appears to have been superseded in Billion's security appliance line-up, the support documentation was still available on their site, as was the latest firmware (with which we flashed the unit before setting up the connection). With 20 simultaneous SSL VPN tunnels, the S20 was rather overkill for our needs. However, since it also provides two WAN ports for link failover and load balancing, we thought we'd better try them out with a simulated line outage. We're still a bit hazy as to how much data the S20 caches, but whatever the

case we were quite impressed when we yanked the primary phone line part way through an upload and the unit failed over instantly to the backup line without data loss or interruption.

Another pleasant surprise was the relatively compact nature of the data stream even after encryption. Even when dedicated hardware is being used, SSL encryption can chew up a fair bit of additional bandwidth, and since our park keeper/trader was at the end of a rather tenuous occasionally narrowband internet connection, this was a concern. In practice, while performance during updates was hardly what might be termed low latency, everything worked reliably and certainly sufficiently quickly for our needs. It also proved reasonably secure. One of the Wrecking Crew who insists that he's only ever had a white hat had a go at compromising the connection, firstly by attempting to dupe the park keeper/trader with a fake log on page, and then by trying to exploit possible vulnerability in the unit's implementation of block ciphers (no, us neither). No luck, but it kept Horace quiet for hours poring over log files for evidence of intrusion, so we didn't care.

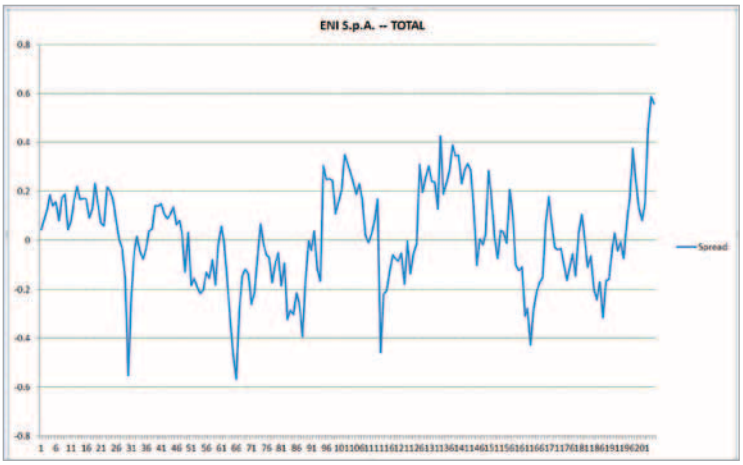


Figure 4

Conclusion

While this may appear to have been an extremely artificial exercise, it did have the benefit of establishing beyond doubt that even a diverse mix of applications can actually function effectively in a live trading environment. There was the odd thing to be tweaked and we may well have been fortunate in our choice of "ingredients", but overall it was striking how quick and easy it was to fit things together and actually be productive.

Figure 5

Obviously, most readers would have followed a far less convoluted path than ours, but the basic components we used do seem to combine well and also represent very good value. Combining first year license costs for MATLAB (without additional toolboxes which we didn't need as we mangled the Spatial Econometrics toolbox instead), AccelerEyes (single GPU license) and IB-MATLAB runs up a total bill of $3498 that would fall to approximately $576 in second and subsequent years. The only additional costs would be Excel (which we suspect most readers probably already have) and CQG to provide data. The basic CQG Integrated Client (not including exchange fees) costs $595 per month, with a further $45 for real time API access and $100 for basic historical API.

Apart from value and general "plumb-ability", the other thing that struck us during the review process was how far some of the applications had advanced in a relatively short space of time. CQG had made some significant changes/enhancements to its API sample code and MATLAB had launched its new Econometrics toolbox; but the most striking changes had probably been made by AccelerEyes and IB-MATLAB. Jacket is now virtually unrecognisable in terms of native support for MATLAB functions, while its new ability to apply multiple GPUs to a problem without also requiring MATLAB's PCT toolbox[1] represents a significant landmark in cost effective high-performance computing. IB-MATLAB's addition of streaming data support is similarly significant, as is its access to other data items via the IB API. Combining this with its existing automated trade execution capabilities means in our minds that – as long as it isn't being used for high frequency trading or a vast portfolio – then IB-MATLAB effectively contradicts the declaration we've seen on more than a few web sites that "MATLAB is not for real time trading".

1. We should mention that MATLAB has also added its own native GPU support to the PCT toolbox; we didn't test that as part of this review, but we intend doing so in a future issue.