VECTOR

The APL World in 2007:

BCS®

The Journal of the British APL Association

A specialist Group of the British Computer Society

ISSN 0955-1433
www.vector.org.uk
Vol.23 Nos.1&2 January 2008

Contributions

All contributions to VECTOR may be sent to the Journal Editor at the address on the inside back cover. Letters and articles are welcome on any topic of interest to the APL community. These do not need to be limited to APL themes, nor must they be supportive of the language. Articles should be accompanied by as much visual material as possible (b/w or colour prints welcome). Unless otherwise specified, each item will be considered for publication as a personal statement by the author. The Editor accepts no responsibility for the contents of sustaining members' news, or advertising.

Please supply as much material as possible in machine-readable form, ideally as a simple ASCII text file or an HTML document. APL code can be accepted in workspaces from I-APL, APL+Win, IBM APL2 or Dyalog APL/W, or in documents from Windows Write (use the APL2741 TrueType font, available free from the website), and MS Word (any version, ideally using the Vector template which is also available from the website).

Membership Rates 2007–2008

Category	Fee	Vectors	Passes
UK Private	£20	1	1
Overseas Private	£22	1	1
(Supplement for Airmail, not needed for Europe)	£4		
UK Corporate Membership	£100	5	5
Overseas Corporate	£110	5	5
Sustaining	£500	10	5
Non-voting Member (Student, OAP, unemployed)	£10	1	1

The membership year normally runs from 1st May to 30th April. Applications for membership should be made to the Administrator using the form on the inside back page of VECTOR. Passes are required for entry to some association events, and for voting at the Annual General Meeting. Applications for student membership will be accepted on a recommendation from the course supervisor. Overseas membership rates cover VECTOR surface mail, and may be paid in sterling, or by Visa, Mastercard or Amex, at the prevailing exchange rate.

Corporate membership is offered to organisations where APL is in professional use. Corporate members receive 5 copies of VECTOR, and are offered group attendance at association meetings. A contact person must be identified for all communications.

Sustaining membership is offered to companies trading in APL products; this is seen as a method of promoting the growth of APL interest and activity. As well as receiving public acknowledgement for their sponsorship, sustaining members receive bulk copies of VECTOR, and are offered news listings in each issue.

Advertising

Advertisements in VECTOR should be submitted in typeset camera-ready format (A4 or A5) with a 20mm blank border after reduction. Illustrations should be photographs (b/w or colour prints) or line drawings. Rates (excl VAT) are £250 per full page, £125 for half-page or less (there is a £75 surcharge per page if spot colour is required).

Deadlines for bookings and copy are given under the Quick Reference Diary. Advertisements should be booked with, and sent to Gill Smith, Vector Production, Brook House, Gilling East, YORK YO62 4JJ. Tel: 01439-788385.

Email: gill@apl385.com.

Contents

Editorial: A Lot Happening

by Stephen Taylor (editor@vector.org.uk)

A lot to report on in this issue. That's partly an artefact of how far we again dropped behind our publication schedule, so let's start by looking at that.

Vector's typesetting is unusually demanding, and requires some understanding of the code and the mathematics we publish, of the more advanced features of Microsoft Word (in which we assemble the camera-ready copy) and also of the markup and encoding schemes used on Web pages.

For many years production of *Vector* would have been beyond the means of the BAA were it not for the skills of Adrian Smith at Causeway Ltd. In the course of this work he has designed and produced the APL385 Unicode font used both for Vector and the APL Wiki. (See Kai's article.) But even minor interruptions to Adrian's availability can disrupt regular production of Vector.

Happily, the spread of Unicode browsers enables much simpler typesetting solutions. So we've begun a project to simplify the markup and composition work. We have for some time preferred to receive articles as text files or HTML documents. As part of this project we'll produce a simple guide for authors to marking up articles for *Vector*, either as HTML or as XML, using just a text editor. (We know this won't work for everything we publish, so we'll continue to negotiate format and content with authors as necessary.) This will reduce the work required to prepare an article for publication, and opens the way to publishing – eventually – articles promptly online before we collect them in a printed volume.

BAA is following Dyalog's lead to Lulu.com's print-on-demand service. We plan to use this to make printed or PDF back issues of *Vector* available on demand, and to publish a series of Vector Books, starting this autumn.

But there is also much to report. Kx Systems goes from strength to strength. Its first residential user conference is beginning in Ireland as we go to press. Much welcome news from Dyalog Ltd, which has moved into new offices, expanded to 13 staff, and has acquired Causeway's admired line of software tools. Morten Kromberg has also been spotted playing with an experimental Dyalog interpreter with native Unicode character strings, and demonstrating that while the five thousand Chinese characters in the Unicode set might look alike under Western eyes, their shape and their nub's shape are in fact the same.

Kai Jäger has, with help from Chris Burke at Jsoftware, established the APL Wiki, modelled after the successful J Wiki. It's hard to overstate the potential importance of this. The difficulty of reading and writing the traditional APLs on ordinary PCs and on the Web has been widely blamed for their fall from favour.

Now, and for the first time, APL programmers have the prospect of sharing code over Web pages, cutting, pasting and typing their code. (If you are using a Unicode-enabled APL like APLX or Dyalog, you can also cut and paste between your session and the APL Wiki.) Hats off to Kai for envisaging this, and for contributing the time to make it all function. Making it useful is a job for the rest of us!

More crossover action from the Kx Systems user conference in Ennis, Ireland (*May 2007*). Arthur Whitney's Q programming language can be thought of as an APL savagely hacked for raw speed. In the course of developing it, Whitney excludes anything and everything that might compromise that goal. Once a Q program has worked its magic with billion-row tables, it leaves its author reaching for richer tools for presenting the results. Morten Kromberg of Dyalog showed how a full-featured APL could take over the running without losing the productivity of Q's array paradigm. Expect to see more exploration of the APL-Q partnership.

I'm sorry to report that Eugene McDonnell plans to send us no more "At Play With J" articles. We've had 38 of these over the years; Eugene tells us each one took about a month to write. It's the end of an era. To mark our appreciation of this work, the BAA will collect and publish it in book form this year.

This issue contains a substantial introduction to APL by Bernard Legrand, translated from the French, a labour of love by Sylvia Camacho. Richard Nabavi discusses the design decisions behind the 64-bit version of APLX.

Much practical help in APL+Win here from Ajay Askoolum. Ian Clark reviews Ajay's new book, and we add two articles by Ajay, on writing C# COM DLLs, and on zipping and unzipping files in APL. Klaus Christiansen offers his APL wrapper for the widely-used Developers Image Library, and Ray Cannon, master of crash recovery, writes about handling and analysing CONTINUE workspaces. Adrian Smith offers his own take on classes in Dyalog 11.

This issue kicks off a new column *In Session*, successor to the long-running *Hackers' Corner*. We're looking for fragments from your session logs when you've done something elegant, impressive, useful or just curious – perhaps not worth working up into an article, but nonetheless worth sharing. Show us your moves!

As always, we have articles by people using the APLs to get results in their own fields. Sylvia Camacho adds another article to our series on the Monty Hall problem. Timothy Zirkel plays the Game of Life in 3-D, and Ralph Selfridge considers different ways to write the Black-Scholes algorithm. And Ian Clark describes his agile and innovative approach to digitalising the *Vector* archive.

News from Sustaining Members

Dyalog

Dyalog acquires IPR from Causeway Graphical Systems Ltd

Causeway is a household name in the world of APL software programming. For more than a decade the tools developed by the company have been used by software developers in a multitude of software applications over a broad range of platforms. The tools include: RainPro graphics engine – also known as SharpPlot for the Windows.Net platform, NewLeaf report formatter, and CausewayPro framework for Rapid Graphical User Interface (GUI) development. All will be incorporated into Dyalog for Windows from Version 11.1 – planned for release in early 2008. Dyalog confirms that the price structure will remain unchanged despite the implementation of the additional tools in version 11.1.

Founder of Causeway, Adrian Smith, is recognised in the worldwide array language community as a visionary systems architect and master toolsmith. Adrian will continue to work as chief architect on the Causeway product line, particularly with a focus to design further enhancements. Jonathan Manktelow has joined Dyalog as a full time developer.

Managing Director of Dyalog Ltd, Gitte Christensen, says "We're extremely pleased to welcome the Causeway team onboard. We gain additional developer resources particularly for combining the use of Dyalog with components written in C#, something which is of great value as we continue to work towards a seamless integration of Dyalog with the Microsoft.Net platform. We're also given a unique opportunity to take forward excellent products without additional cost to our customers hence making them available to many more users. Additionally, Dyalog is taking ownership of the APL-to-C# Translator developed by Causeway as well as the accompanying SharpArrays array library for the Microsoft.Net platform."

Dyalog is currently not planning to develop the APL-to-C# translator technology into a general tool for deployment of APL applications. However, the company believes that the technology is very useful for customers who wish to produce fully-managed components from APL source code. The translator will therefore be made available on a per-project basis with Adrian Smith providing consultancy, development and support.

Adrian Smith comments, "I know that it is Dyalog's aim to continue to build an even more complete and robust development environment, and integrating the Causeway tools is a big step in this direction. Dyalog can provide the professional marketing and support which the products deserve and I can now dedicate more time to designing and coding as part of the excellent group-dynamic that characterises Dyalog Ltd."

Country life

We have always believed that a development department ought to have sea view – as it enhances the creativity to have something nice to look at.

However, when you are looking for new premises in Hampshire, that is more difficult to achieve. Usually, the closest to water we get is the odd puddle on the road after a heavy rainfall.

Instead of the sea we found a place with a view of a large wheat field, where the wind creates ripples just like the movement of the waves and where a multitude of birds and animals can be spotted.

In March Dyalog moved to new premises in the small village of Bramley which is situated just off the A33, which connects Reading and Basingstoke. We now occupy South Barn, in a complex built recently. But in keeping with "old" style it sports character features such as exposed beams. It is significantly larger than our previous offices, as we plan on staying here for many years, and still have room to expand the Dyalog team as the company continues its steady growth.

Everyone has now found their feet, the dedicated machine room is buzzing with a multitude of PCs and our meeting facilities are completed.

Kx Systems

Kx Welcomes New Customers

Announcing the latest additions to our growing list of leading global financial firms: The NYSE Group, Rand Merchant Bank (RMB International), and Detica.

Mac OS X Support for Kdb+

Kx has announced kdb+ database and application support for the Mac OS X operating system, which we demoed on a MacBook at Trade Tech Paris last month. Kx added Mac support in response to increasing requests from our developer community.

> "As Apple continues to offer larger, faster servers built on Intel processors, the Macintosh is becoming an attractive alternative for financial institutions," said Simon Garland, Kx CTO. "IT departments are looking for maintainable performance, and Macintosh servers are simple to administer. Kdb+ doesn't need a raft of clusters and grids in order to deliver millisecond query response times. New customers are often surprised by how much they can accomplish using our software with a single processor."

Kx expects the Macintosh option for kdb+ to appeal to leading financial firms not only for corporate servers but also because the Mac's stable and easy-to-use development platform appeals to developers in the office and at home. "Many of our customers are trading globally 24×7, which means that growing numbers of developers using notebooks are going online to monitor applications and finish development from home or on the road. The Mac notebooks are ideal for that," commented Garland.

The port to Mac OS X was straightforward, largely because Kx architected its kdb+ database and applications for easy portability to support new operating system versions quickly. Both 32-bit and 64-bit versions of kdb+ on Mac OS X are available now. In addition to Mac OS X, kdb+ supports Linux (any version), Windows, and Solaris.

New A-Team Research on Low Latency Architectures

A-Team's report, "Faster than a Speeding Bullet – Low Latency Architectures and Building Blocks for Tomorrow's Trading Applications," explores the market dynamics that have driven low-latency requirements and reviews the technologies now being deployed.

The Cross-Asset Trading Challenge

Transaction Networks & Technologies has published Kx CTO Simon Garland's article on new paradigms in trading assets among different classes, explaining how to manage the risks and avoid the pitfalls.

BAA Annual General Meeting 2006

Anthony Camacho, Secretary (acam@blueyonder.co.uk)

Minutes of the AGM of the British APL Association on 19 May 2006 at the Royal Statistical Society. (We inexplicably failed to publish these last year. *Ed.*)

Paul Grosvenor, chairman, opened the meeting by listing the agenda. He said the Association had had a quiet year in which Ray Cannon's moot had been the outstanding event. At the morning committee meeting it had been suggested that the association support work of various kinds to promote APL. He had noted six projects:

1. Develop Stephen Taylor's approach to system production (as in his article on "Pair programming" in *Vector* 22.1) into a book.

2. Republish the "At Play with J" and/or the "J-ottings" columns in book form.

3. Produce a primer in APL covering the major implementations for which a free or nearly free interpreter is available. This might be accompanied by a CD or DVD containing the interpreters. It would require careful liaison with the vendors.

4. A series of case studies with object lessons in what APL is good at. Each of these could serve to spark interest in non-APLers. Conference proceedings and *Vector* articles could be mined.

5. Improvements and enhancements to the web site to provide an interpreter which could be used for experimenting with APL and a Wiki.

6. Make available video recordings of noteworthy presentations or sound recordings of interviews with APLer who had interesting tales to tell.

Paul concluded that the APL Association would be willing to provide seed money to help with such projects and encouraged everyone to propose a project that they were keen on and would help with. Send details by email in the first instance to Ian Clark or to any of the committee. Addresses are in the back of *Vector*.

Nicholas Small, treasurer, circulated and spoke to the accounts. The apparent increase in expenditure was not all real because he had underestimated the outstanding bills included in last year's accounts and these inflated this year's outgoings. Income from advertisements had dried up while *Vector* timing was unpredictable. There were no questions.

Ian Clark reported on two current projects: work with Mathematica (promoting APL and Mathematica together) and making the complete *Vector* archive available on the web site. On the first project little progress had been made but he does have a copy of Mathematica to work with. On the second he has a workspace (VARCH) which converts articles from the *Vector* disk circulated at Madrid into suitable HTML. Each article needs some extra work as the process is not completely automatic. If anyone needs a particular article he could probably bring it to the top of the pile and convert it soon; please ask. To Paul's list of projects he suggested adding a volume of material by and about Gérard Langlet – a sort of festschrift.

Stephen Taylor reported on *Vector*. This time last year his aim was to get back on schedule. He was sorry to say that we are as late now as we were then. However *Vector* 22.2 is at the printer and he hopes *Vector* Vol 22.3 will go to the printer within a fortnight. Furthermore the material for *Vector* Vol 22.4 is all collected and he hoped it would be ready to print in a month. *Vector* 22.3 is the Iverson memorial edition. Stephen also reported that he had done away with the *Vector* APL forum, which had been misused and made contributions vulnerable to spam.

Paul Grosvenor then asked for a proposer and seconder for next year's committee (the same as this year). John Toop and Richard Nabavi volunteered. The committee was elected.

There were no questions to officers and no other business so the AGM closed at about 2.35 pm.

The Elsinore Songsheet

collated by Stephen Taylor

My tablet! Meet it is I write it down... (Hamlet)

These lyrics were written and performed by delegates to the Dyalog User Meeting in Elsinore, Denmark in October 2006 to melodies kindly supplied by the convenors. The website has links to video recordings of the performances.

"My boss is waiting"
Lyrics: Blue Goat
Melody: *O Sole Mio*

My boss is waiting
and it's close to deadline
I have this problem
that's really geeky
I must do something
get a quick solution
to see some smiling faces in the morning

My only rescue, my secret joy
Oh APL you
are there for me
for me and for all the others
who know your secrets
who know you well
we know your secrets
and all your wealth

"That first day I saw APL"
Lyrics: Yellow Goat
Melody: *Tango Jalousie*

That first day I saw APL
I knew then that all would be well
I was rotated, transfigured and transposed
My programs were poems, no longer prose

That first day I saw APL
All I wanted to do was to write
We all stayed up later
To write operators
That first day I saw APL

"I could write code all night"
Lyrics: Blue Horses
Melody: *I could have danced all night*

I could write code all night
I could write code all night
And still have bugs and more

I could have saved the code
Before I did that)load
And then lost hours of work

I'll never know what made me type
that line in
Why all at once my code took flight

I only know that when
I started APL

I could write code, code, code — all night

"Arrays that made us free"
Lyrics: Purple Horse
Melody: *Toreador*

Coding solutions used to be a bore
It made us snore
Was such a chore

Iverson, inspired by the Lord
Started to write on his board…

A notation short and sweet
That does much more
Now jobs don't go offshore

Terse formulations are our kind of style
Some find it vile
We only smile

C, in use for years in 1973
Simply should never be

With the other scalar crap
Not hard to see
Arrays that made us free.

"Like a star in the sky"
Lyrics: Green Horse
Melody: *Tango Jalousie*

APL seems like a star in the sky
Which makes our mind soar so high
I thought about the course of my
hazzling life
And search for the reason for being there

Perhaps to be a coder on the sea
There was a bug that sounds like a melodie

But suddenly I see an array of shooting stars
Was it heaven or version eleven?
"All my ops are slow"

"All my ops are slow"
Lyrics: Blue Cow
Melody: *Mein Herr. Marquis*

Who can help me out
I need some help – for sure
Clients start to go
Who can help me out
I need some help – for sure

I tried using Oracle – NO! Ha-ha-ha
I tried using SAP – TOO! Ha-ha-ha

HUM HUM HUM
…
…
…

Now I'm using APL
All is working YES YES YES
[Repeat to the END]

"And Each and Or…"
Lyrics: Pink Goat
Melody: *Toreador*

Alpha, Omega, Epsilon and Rho
It all sounds like Greek, unless you are a Pro!
Functions, returning functions …
Always … something new!
How will we cope with this
What can we do…
We need a drink or TWO

Upgrade or Downgrade, Scan or Domino
It all sounds like Geek unless you're
in the know
Classes, and Fields and Triggers…
Always … something new!
But we will cope with this
We always do…
After a drink or two….

"Requirements have changed again!"
Lyrics: Red Goat
Melody: *Mein Herr Marquis*

Client comes to us
Business in a mess
"Requirements have changed again!"

Client do not fuss
APL is best
We'll jointly see this through

You need to consider our fee
And forget any grand strategy

Your problems may all come in threes
Arrays of thought take each of those
Agile interpretation is the way!
Agile interpretation is the way!
Iota, Rotate or Transpose
will lead your client to repose

COBOL & LISP?.. AH-HA-HA
FORTRAN, JAVA? AH-HA-HA
4GL, C? AH-HA-HA
C# or VB!? AH-HA-HA HA!
[REPEAT]

You've come to the right place
(now please) sign here!

"I could have danced all night"
Lyrics: Yellow Cow
Melody: *I could have danced all night*

I could have danced all night
But chose APL to write
And still I beg for more

APL scanned my string
And did a thousand things
That never worked before

I think I know – what makes it so – exciting
from APL, I take a byte
I only know – that it – began to work for me

And now I A—P—L
All night

"My wife has dropped me"
Lyrics: Purple Cow
Melody: *O sole mio*

My wife has dropped me,
→ to my best friend.
My colleagues shun me,
my life is near its →0

I've sunk to C sharp
and I work for Enron,
there is no ending
to the pain I suffer.

But there is Dyalog,
it is my life.
I ↓ my problems,
↑ back my life.

And soon I'll have Eleven
Oh Happy Day
:With Dyalog!

Pronunciation: →, 'branching'; →0 'end'; ↓ 'drop'; ↑ 'take'.

The singing was backed up by three professional opera singers and a first-class pianist, and the whole experience was quite remarkable. The Pink Goats (led by Paul Mansour and Lynne Shaw, with Carlo adding that ineffable Italian touch) still treasure their trophies.

APL's 40th Birthday Party in Stuttgart

notes by Adrian Smith (adrian@causeway.co.uk)

Everything has an official birthday, and in APL's case the first witnessed success of `)LOAD 1 CLEANSPACE` seems to be accepted as the moment. Which is why we joined APL Germany and the GSE APL working group at the IBM Germany office in Ehningen for a splendid celebration and party on November 27th 2006. Thanks are definitely due to IBM for hosting the excellent meal and party, and to APL Germany for organising some really interesting material. And for printing the apples! The T-shirts were pretty spectacular too, although very little of the APL code would have executed in 1966.

Apples, Quoted, Printable

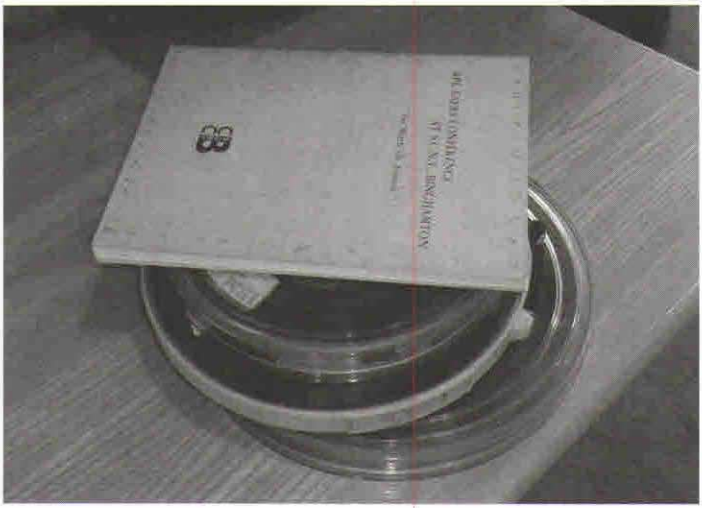
Historical 40th APL Anniversary

The two great survivors from the early days of the modern era are FORTRAN (just past 50) and APL which is some 10 years younger – indeed the first practical implementation was submitted in batch as a bunch of FORTRAN cards. The meeting was enhanced by several exhibits of early APL material, with keyboards, installation tapes and even the odd card-deck on show. Much of the early literature is still as relevant today as when it was first written, and it was good to see so much collected in one place.

The talks given at the afternoon meeting was a mix of pessimism (the decline in APL as measured by the delegate-count at the annual conferences) and optimism (the power of APL as illustrated by its achievements in the past and its continued substantial revenues for IBM). Dieter Lattermann apologised that his involvement with APL only stretched back 39 years to 1968, when he worked on the original *Complementary Functions*. Remarkably, this was immediately classified IBM Confidential and apparently it still is, as no-one alive today can work out how to remove the tag! Dieter represented IBM on the language group of the international standards committee, and organised what may have been the most successful APL gathering ever – at Heidelberg in 1982 where 800 participants made enough profit for the organisers to found the *APL Club Germany*, and to continue to fund it to this day.

Reiner Nussbaum brought us up to date with a very neat APL2/PC system that interfaced with his GPS and MapQuest (used very effectively to create the instructions for finding the meeting) and Morten Kromberg reviewed his own long history with IP Sharp (where he worked for 10 years) and more recently with Dyalog. A major theme in Morten's talk was the power users gained from the early open designs (just leave the user at the 6-space prompt with a bunch of handy keywords) – much of which was lost when AP124 pushed us all down the menu-driven route, and then lost again as everything morphed into clicks and grunts and we spent all our time designing icons for toolbars rather than delivering power to the user.

One contribution in particular stood out as a fascinating record of success in the very early days, so I am going to skip by the pessimistic stuff and just tell you about Yves le Borgne and *Early APL in Europe*.



An original APL\360 and the March on Armonk

NASA and the First Steps in Europe

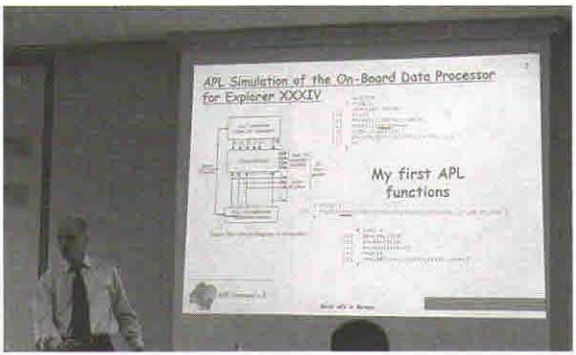
Yves covered the very early days of APL at the NASA Goddard space-flight center which used the APL notation as early as 1965 to document the design of satellite experiments. By 1969:

> "People who were normally playing pétanque at lunchtime were staying inside to learn APL."

The first release of APL\360 was the turning-point in 1968-70 and the marketing of APL began in earnest in Europe in the early 1970s. The rise of APLSV in the period 1973-76 was the beginning of the period which was recognised by many of the other speakers as 'the high point of APL' around the world.

First steps in APL at Goddard

The thing that Yves's talk brought home was just how much of the power of APL was available in the very early days, and how much really useful work was done with it over 10cps tele-typewriters with flaky 300 baud connections to distant mainframes. Here is one example:



APL in Satellite Design

1973: The Battle for APLSV = the Rimini Plot

APLSV was ready to go out by the end of 1972, but IBM politics began to slow things down as the strategists wanted to wait for the 'main line' products to be ready. SEAS APL meetings in Heidelberg and Rimini helped to build sufficient pressure to allow APLSV to be announced in May 1973 and demonstrated during the APL Congress at Copenhagen in August. Things were looking up, and the APL community grew quickly, until ...

1976-1986: the environment gets too complex

By now VS APL had become the IBM APL horse, but the road ahead was much less smooth. There were some serious flops which began to damage APL's image, as the 5100, APL\DPPX and APL\CICS failed badly. VSPC was pushed (in place of VM/CMS) as the preferred environment but it was complex, buggy and badly-performing by comparison. The PC era was just around the corner ...

Onward to the Party and Blossom Time Sing-song

There was time for Jim Brown to entertain us with more stories from the early days of APL2, and for Dave Liebtag to re-assure us (backed up by a tele-conference with IBM's VP of enterprise software) that APL was still a big time revenue earner and productivity enhancer for IBM. Then it was on to the excellent buffet supper provided for us by IBM Germany, and back in time to celebrate the moment (in the US timezone, of course) when the first workspace announced itself to the world. The ceremonial singing of APL Blossom Time may one day be available for download as a very tacky video, but for now let's draw a veil over the evening and enjoy one last APL apple and the visiting Vogon spaceship ...

Yes, you can print on an Apple!

As it says in the book, they fly just the way bricks don't

Incidentally I kept one of the apples well into 2007, and the motto remained visible until the last. We will have to plan something pretty special for 2016 if we can wait that long. Maybe we should mark 42 years of APL next autumn instead!

Building OO Applications in Dyalog Version 11

Stephen Taylor (sjt@lambenttechnology.com)
Gilgamesh Athoraya (e9gille@googlemail.com)

We were privileged to be invited to help Dyalog prepare Version 11 for release. And we abused that privilege. Roundly. Instead of trying to make the interpreter misbehave, we misbehaved. We used the release candidates to rebuild the major application we were developing that year for a pension company.

Like any developers working on a mature system, we itched for a new start to exploit the insights we'd won into the application. And, like good APL programmers, we were keen to see how *lightly* the system could be written, especially using the new tools.

We were already making extensive use of the object model in a subsystem that marks up plain text to produce RTF files, that Microsoft Word is happy to open, edit and print. But our crude object model, implemented with Dyalog namespaces, made instance objects simply by taking deep copies of the class object. This is inefficient. Worse, changes to instance methods don't propagate automatically back to the class. But primarily, we were excited about seeing how OO might clarify and simplify our code, what our GUI code might look like, and what our code management would be like using scripts. (Could we collaborate using Subversion, for example, as so many writers of other languages do?)

Stephen got our first release candidate on Kefalonia in May 2006 and, to his partner's horror, sat up nights under the stars writing the first version of what came to be called Surrender to the sound of goat bells tinkling over the mountainside. This looked very exciting when brought back to England. Looking for maverick shortcuts, and inspired by A+, he'd derived the object model from GUI classes. Every instance had a native GUI representation – no mapping to be done! Objects had their own Browse methods, which would let you edit them.

Exhilarating as this was – the code had just melted away – we discarded this as a dead end. (It supports only single views of each object's contents; worse, each GUI object consumes limited Windows GUI resources. It would be inefficient in batch – i.e. non-GUI – use, and unlikely to scale to arrays of objects.)

We then set out on a more conventional path, separating the presentation layer from the object model. Because we thought we would probably want to deploy the system over an intranet from behind IIS, we used a structure isomorphic to

ASP.NET, reasoning that an eventual port to ASP.NET would require rewriting only the modules we had modelled on ASPX pages. But for ease of development, and because we thought Remote Terminal a plausible alternative, we wrote all our GUI in Dyalog. Focused on progress rather than testing, we didn't often stop to investigate what we couldn't make work, more usually simply finding another way around.

By September, working in what we pleased to call our spare time, we had a simple version of the application creating, saving, opening XML claim files using object serialisation and deserialisation, interpreting and importing data from mainframe extract files, browsing the claims and producing correspondence in RTF and XML formats. All the code was held in UTF-8 `.dyalog` script files, using SALT, and assembled at runtime with a `.dyapp` file. We hadn't reported many bugs, but we were able to show Dyalog that their release candidate would support an application using the new extensions. We took this system to the Dyalog conference in Elsinore, where we attempted to share what we had learned with those attending Dan Baronet's workshop on the OO extensions. We considered that unsuccessful, largely because Dan's students had come simply to learn what the OO extensions were about; we were able to share very little of our experience.

Nonetheless we thought it would be of value to anyone else setting out down the same path, and offered to run a longer course, for students who had already worked through Morten's tutorial on the OO extensions. This eventually turned out as a 5-day workshop in January this year, generously hosted by The Carlisle Group in Scranton, Pennsylvania. Stephen always gets a kick out of visiting Scranton. Our Californian students were politely but explicitly dismayed at spending days in the heart of the American rustbelt. But for fans of 1920s provincial architecture, Edward Hopper paintings or Sinclair Lewis novels, Scranton is full of delights.

Gauging the schedule after Elsinore was challenging. By the third day of the workshop we saw how the remaining material could be reorganised around a practical task: building an address-book application, with a browser, images for contacts and so on. We negotiated with the students that we would start that after lunch that day, dividing into pairs to work. Everyone agreed it would be a useful way to work, even if no application got completed before we finished. In the event, everyone had a simple but working OO address-book application by five that afternoon.

Next time we run this we will offer it as a 3-day course, with an optional one or two days coaching on students' own applications, to follow either directly afterwards, or after an interval. If you think this course might be of value to you, please contact the authors.

The APL Wiki

by Kai Jäger (kai@aplteam.com)

Wikis are widely accepted and used nowadays, although not everybody is necessarily familiar with the concept of a wiki. Before telling you about the APL Wiki, it might be a good idea to review what a wiki is.

Basically, a wiki is a website which can be changed by everybody. It's really that simple!

Change means: edit, add and remove pages. But why should one create or edit a page at a particular web site? To add a statement, fix a typo or contribute or improve content!

The word *wiki* is a Hawaiian word meaning *quick* [1]. Wikis succeed in attracting content from a wide range of people, in part because they can be edited so quickly. Wiki interfaces are designed to be understood and mastered easily. They focus on the essentials, and nothing else. That is what has made wikis such an overwhelming success.

Problems

What about spammers and malicious hackers? This is surely a problem, but the success of Wikipedia [2] tells us that wikis work anyway. Every day thousands of pages in the Wikipedia are deleted or flooded with rubbish. It works nevertheless.

A wiki keeps track of all changes. That means a good guy can restore a destroyed page with a single click. Nevertheless the problem is a large one, and almost all wikis nowadays require contributors to log in before editing. That makes it much harder for our beloved spammers to flood a wiki with rubbish.

In the APL Wiki you can freely add attachments, but this might not last. (Many other wikis are flooded with porn attachments.)

Why an APL Wiki?

A wiki is a perfect environment to let people collaborate. Since Jsoftware Inc. introduced the J Wiki [3] it has become a valuable source of information for the J community. By the way, the J Wiki was a model for the APL Wiki. One of the administrators of the J Wiki, Chris Burke, helped me to install and configure MoinMoin – thank you very much Chris!

My hope is that the APL Wiki becomes a recognised source for valuable information about APL: articles, examples, tasks, papers, solutions, libraries, frameworks...

Getting Help

Especially for beginners it is important to get a helping hand with problems. The APL Wiki uses MoinMoin [4], which offers a large number of help pages. The names of all help pages start with `HelpOn`. So entering `helpon` in the search box and pressing Enter returns a list of all the help pages.

Please note that pressing Enter triggers a title search. Clicking on the `Text` button has MoinMoin perform a full text search across all pages. Understandably, this type of search can take a while.

Unicode and Fonts

The APL Wiki is encoded in UTF-8. That means that all modern browsers should display APL characters. As usual, Internet Explorer might cause a problem. The behaviour of IE5.5 and IE6 is unpredictable, as is version 7; but in general version 7 is much better. However, on some machines even IE7 does not display UTF-8 correctly, and nobody understands why.

Strictly speaking, you do not need a particular font: any Unicode font will do. The APL Wiki will use the APL385 Unicode font [5] if it is installed on your machine, otherwise Courier is used. That should still result in APL symbols, since nowadays your Courier is likely to be a Unicode font with all the APL symbols. But few fonts have been designed to display APL *attractively*: you will see the best results by downloading and installing the APL385 Unicode font file.

Most of the advice on *Vector*'s help page "Displaying *Vector* pages" applies to the APL Wiki.

APL Code: Edit, Copy, Paste

Why is Unicode so important? Because you can copy and paste code between platforms supporting UTF-8! For example, you can copy APL code from an edit window in Dyalog APL or APLX and insert it into an article on the APL Wiki. Put {{{ and }}} around it and you are ready.

That also works the other way around: copy APL code from the APL Wiki, insert into the session of Dyalog or APLX and it will run.

Google Mail optionally supports UTF-8, too. So you can exchange APL code between sessions, edit windows, the APL Wiki and Google Mail.

If you have installed Dyalog APL/W on your machine, you can use its Input Method Editor to edit APL code in the APL Wiki. From within Microsoft Word or the MoinMoin article editor you can change the keyboard layout to APL and then type APL just as you would in Dyalog.

The Test Wiki

Of course you should get some experience before starting serious contributions to any wiki. For that, many wikis provide a *sandbox*. This is a small wiki, basically a copy of the master wiki, where everybody can play around without danger of destroying something valuable. The APL test wiki address is:
http://aplteam2.com/testwiki/

Note that this address is very different from the APL Wiki address:
http://aplwiki.aplteam.com

Certain pages are protected to ensure an appropriate structure for the root of the wiki, but otherwise pages are editable, and you can add any new pages you think are needed.

Creating an Account

Even in the test wiki you need to create an account first. (Otherwise the test wiki would soon become a porn site.) It is recommended you use CamelCase for your username: e.g., `JohnDoe` rather than `John Doe`. This way you can easily add links pointing to your internal "home page". (You might not wish to create such a page right now but there is a good chance you will later.)

Note that passwords and usernames are case sensitive.

In case of a forgotten password you can ask the wiki to send you an email with your password hash, which MoinMoin will accept instead. Then it would be smart to change your password! (MoinMoin does not save passwords but hashes, so it can tell you only the hash. A hash can be generated from a password, but there is no way to recover a password from its hash.)

Editing

To edit a website, normally you need to know HTML. Although HTML is simple enough, wikis make editing even easier with an extremely simplified markup. For example, to mark up a top-level header one would say:

```
= Heading =
```

A second-level header would look like this:

```
== Subheading ==
```

Here you see the markup for an unordered list:

```
 * a topic
 * another topic
```

An example for an ordered list:

```
 1. First item
 1. Second item
```

It does not matter which number is used – any number will do. Note that for list definitions the first character on a line must be a blank.

Finally an example for a bold word:

```
This is a '''bold''' word
```

Besides lists and headers, you need to know how to create a link. In general, there are two different kinds of links: *internal* and *external*. The easiest way to create an **internal** link is to use CamelCase syntax. Words written in CamelCase are automatically created as an internal link. There are two possible problems:

1. Sometimes you want to use CamelCase without creating a link. For example, mentioning the most popular version-control software, SubVersion, results in an internal link. Fortunately, there is an easy way to tell MoinMoin that you do not want such a link: `!SubVersion` will do the trick.

2. Sometimes you want a link to something like `DyalogAPL`. That does not work, but `DyalogApl` looks ugly. In those cases you can use some of the more sophisticated markup to create a link. Refer to `HelpOnLinking`.

External links can be created with expressions like this:

```
[http://www.dyalog.com Dyalog APL Version 11]
```

`Dyalog APL Version 11` then becomes a hyperlink.

Since we are talking about an APL Wiki, we need to know how to enter code. Everything between `{{{` and `}}}` gets a box with a distinctive background colour. (All whitespace is preserved.) The syntax we've discussed so far should already be good enough to start your career as an author on the APL Wiki. Don't hesitate!

How to Find an Article

Apart from the search facilities offered by any Wiki, linking is of course very important. A page that no other page links to is hard to find. MoinMoin will identify those pages easily: enter `Orphan` in the search box and you get a list of pages with no inbound links.

MoinMoin also offers *Categories*. The idea is to put a string like `CategoryFoo` at the end of a particular page. Since this is a CamelCase word, a link is automatically created from this. If there is a page with the name `CategoryFoo`, it is supposed to create a list of all pages that have the word `CategoryFoo` on them. That makes it easy to keep pages together that share something. But even more important, the mechanism needs no attention: as soon as a new page is created which contains the word `CategoryFoo`, the CategoryFoo page will automatically include this page in its list.

Of course you can create new categories, but please be careful when doing this. Having too many categories is clearly counterproductive. And when you create a new category, you are supposed to create the category page itself, of course. (You can copy the content from an existing one.)

Conclusion

We now have a collaborative platform in place. Let's start to use it. For those of us making our living with APL it is also a way to share tools and frameworks.

My hope is that the APL Wiki will become a valuable source of information and code, attracting old hands and newcomers. It clearly has that potential – but only the APL community can make it a success. It is up you!

References

[1] Ward Cunningham, The WikiWikiWeb, http://c2.com/cgi/wiki

[2] Wikipedia, http://en.wikipedia.org/

[3] Jsoftware Inc., The J Wiki, http://wiki.jsoftware.com/

[4] MoinMoin, The MoinMoin Wiki Engine, http://moinmoin.wikiwikiweb.de/

[5] Adrian Smith, APL385 Unicode – download from http://vector.org.uk/?area=dnld&page=content/fonts

BAA Annual General Meeting 2007

Anthony Camacho, Secretary (acam@blueyonder.co.uk)

Minutes of the AGM of the British APL Association on 18 May 2007 at the British Computer Society

Paul Grosvenor, chairman, opened the meeting at 1:48pm. He announced some work that had been agreed at the committee meeting earlier in the day:

1. We would make efforts to complete the archive on the web by putting early issues of Vector into electronic form.
2. Stephen Taylor would complete the D-book he had proposed, hoping to publish it in the autumn.
3. We would produce a collected volume of the "At Play with J" articles.
4. We would shortly be publishing a double issue of Vector (Vol 23 N°s 1&2).
5. We hope to organise a 2-day conference in London next spring (one day with developer emphasis and one with business user emphasis).

The Treasurer, Nicholas Small, circulated the accounts: no one had a question for him. The motion to accept the accounts was agreed without demur (prop. A. Camacho; sec. R. Cannon).

As there was no contested post for the committee the chairman suggested the proposed slate be taken as a whole. This was agreed and the proposed committee was elected (prop. S. Camacho; sec. R. Cannon).

The new committee is:

Chairman	Paul Grosvenor
Secretary	Anthony Camacho
Treasurer	Nicholas Small
Editor, Vector	Stephen Taylor
Activities	Ray Cannon
Education	Alan Mayer
Projects	Ian Clark

Rowena Small will continue to handle any necessary administration. The meeting closed at 1:53pm.

Kx Systems User Meeting 2007

reported by Stephen Taylor (editor@vector.org.uk)

Soft grey clouds roll in over the low green hills, swollen with evaporation from the Gulf Stream. We're on the western edge of the continent, the first Europeans to taste this treat from the Atlantic. It sifts down onto us through the warm air, glossing leaves and cars with a thin lacquer, darkening the golf links and stone of this mad hatter's castle. This is Castle Dromoland in County Clare.

The setting is appropriate for the second residential conference of users of Kx Systems' insanely fast, unthinkably compact, Q programming language. Most of what is reported here is hilariously different from the ever-more-complex abstractions of mainstream computing. But Q is what the big beasts of the financial markets use to get results when conventional products and practices fail them.

Three conference presentations focused on how to exploit an investment in kdb+. Brian Fitzpatrick & Felix Lungu described the Q-based tools, modules and applications that First Derivatives has built around kdb+. Veteran K author Charles Skelton spoke of the design issues critical to successful deployment of software using kdb+. And Morten Kromberg of Dyalog demonstrated how, once Q has worked its magic with billion-row tables, richly-featured Dyalog APL provides access to GUI, graphics, classes and .Net assemblies without sacrificing any of Q's array-language productivity.

Niall Dalton, Kx's new Chief Solutions Architect, addressed the challenges of getting from today's powerful 64-bit PCs anything like the performance of which they are theoretically capable, and why Q is such a good environment in which to tackle that challenge. Performance was also the focus of another speaker, neither whose name nor affiliation can be reported here, who spoke from his experience of the huge difference that disk configurations make to actual application performance.

Response speed was the focus of the banquet speaker, Wendy Morgan, from the London Stock Exchange, who talked of the work the exchange has done to strip latency out of its market data feeds. Private trader and Q programmer Mark Sykes explained how the speed at which an algorithmic trading translates into 'slippage' – the difference between the price reported on a market feed and the price at which a subsequent order actually gets filled. He showed how the same

model that trades profitably with a fast response time will lose money if it runs even slightly more slowly. The differences are measured in milliseconds.

Charles Skelton had addressed in his presentation matters of 'style' in writing Q programs, and he offered a set of prescriptions and proscriptions, observing that whether we agreed or disagreed with them individually, writing style has important consquences for the cost of optimising and maintaining programs.

A consequence of Q's success is strong demand for programmers. Jeffry Borror addressed the question of "manufacturing" expertise in Q and kdb+, looking at who is likely to succeed with the technology, and how best to introduce them to it. Like Charles Skelton, he emphasised the importance of writing practices and made specific recommendations.

Something is stirring in the Q and kdb+ world. For the first time, conference speakers seem generally willing to have their presentations published. Expect to see much material from this conference in *Vector* 23.3.

Can One Be Fit on a Starvation Diet?

by Sylvia Camacho (sylviac@blueyonder.co.uk)

> Is Mathematica really the way to go to express our creative thoughts – viz. back to a 17th-century notation designed for parchment instead of forward to a twentieth century one designed for computers? Or to slant it differently: a widely understood and internationally accepted notation, rather than a "starved palette" of terminally overloaded nouns, verbs and adverbs which will only ever really be understood by a few narrow specialists? [...]
>
> This is something I'd like to explore – and I invite others to explore – in a future article. What is it about APL that gives me an edge over somebody armed with Mathematica (or any of its rivals, like MAPLE) when it comes to programming a sophisticated application?
>
> Ian Clark

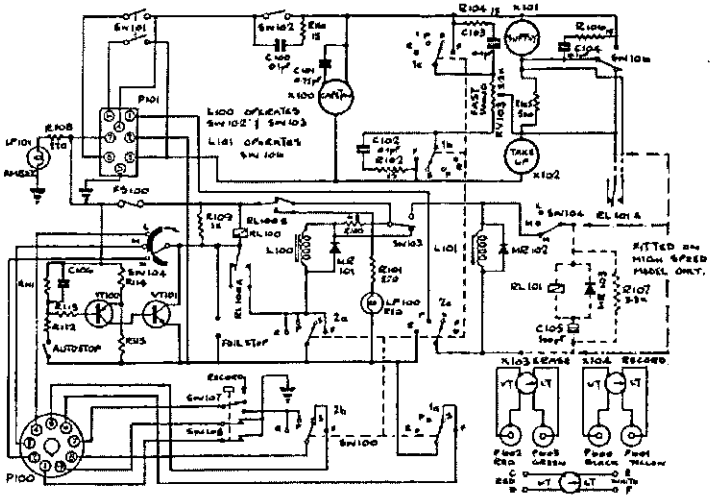
It happened that Ian's challenge [1] dropped at my feet when I was already in the Lists, jousting in a different arena with those nouns, verbs and adverbs, which Ian suggests are unfit for the mathematical service into which Ken Iverson recruited them. I was doing some research into the origins of and relationships between various natural languages; a matter, as we know, very close to Ken's heart. I had just found out, too late to thank him again, why it was he had given us a copy of The American Heritage Dictionary – with the stern inscription "Aug. 1990 for Sylvia and Anthony, the RIGHT version". I guess he had seen that we had only the concise edition, lacking the extensive commentary on the history of language and etymological roots that was his delight [2]. Even as I was discovering its virtues, I was also using an encyclopaedia of language by David Crystal [3] and had reached his brief entry on "The language of science" in which he says:

> The knowledge base of a subject upon which all scientists depend, is accessible only if previous generations have managed to express their findings in a precise and unambiguous manner. Similarly, present day scientists, hoping to make their own contributions to this knowledge base, must satisfy the same linguistic constraints if their work is to be correctly interpreted and accepted by their peers. [...]
>
> The mathematical expression of relationships promotes an extensive use of numerals, operators, letters and other special symbols, which are frequently used in word-like and sentence-like combinations (as formulae, equations, etc.). Lengthy sequences of text can be written in logographic form, thus giving the language of science its highly valued status as a universal medium of expression.

Or, as Ian Clark put it, 'a widely understood and internationally accepted notation'. As Crystal points out, mathematics has that in common with other types of scientific expression; models, charts, pictures, maps and diagrams, but these are almost never totally independent of verbal material to amplify the usage and meaning of non-verbal representations. What is more, comparison of statistics for natural languages shows that the same motivation which has made the various scientific symbolisms increasingly widespread, is making English the favoured worldwide *lingua franca* and the alphabet which it inherited via Rome from the Phoenicians, a universally acknowledged written medium, even for the transliterated version of its language that the Chinese government is promoting. So what is so special about mathematical notation that it should resist this linguistic tide?

A comparison between APL/J and conventional mathematical notation is interesting. Standard APL uses about 50 symbols which are not represented on the normal keyboard. J has 120 entries in its vocabulary of primitives, all of course comprising one or two ASCII characters. In my great tome called *Mathematics from the Birth of Number* [4], I find an index of mathematical "Symbols in Common Use" of 180 entries, of which about 70 can be accessed directly from a normal keyboard, providing italic and boldface are allowed. So conventional mathematics uses rather more than 100 special symbols, that is less than the total of J primitives, to supplement the conventional literary fonts. Why then resist the conversion of all of them, like J, into sets of ASCII characters, thus reaping the same benefits that the Chinese expect from their use of the Roman alphabet? One possible answer is to be found among the types of scientific notation quoted by Crystal.

For example:



Given an electronic circuit of this type it would be possible to provide a verbal description:

> ...but it would be so long and complex as to be unintelligible. On the other hand, the verbal description of the elements of such circuits is an essential feature of training programmes in the subject.

Anyone who has watched an electronic engineer designing a piece of equipment will have seen such pictures being drawn. What then about mathematicians: do they use similar techniques? Of course they do. They draw geometric diagrams, they draw graphs, they arrange ideas as tables; but this is not what is meant by 'conventional mathematical notation'. In the imagination of most people that looks more like this [5]:

> **Simpson's Rule** The area under the approximating parabola can be obtained by integrating between the points x=0 and x=2

$$\begin{aligned}\overline{A}_{02} &= \int_{-1}^{+1} f(1=t)dt \\ &= \left| f(10t + \partial f(1)\frac{t_2}{2} + \frac{\partial^2 f(1)}{2}\frac{t^3}{3}\right|_{-1}^{+1} \\ &= 2f(1) + \frac{1}{3}\partial^2 f(1) \\ &= 2y_1 + \frac{1}{3}(y_2 - 2y_1 + y_0)\end{aligned}$$

Yet all except two or three of these characters can be named by any literate person, so it is not the symbols alone that make this script mathematical: it is the way they are disposed on the page.

Crystal makes this same point by his choice of a very simple example of a mathematical expression:

$$\sqrt{\frac{250}{3}} + 7$$

This simple mathematical expression is unambiguous in its non-verbal form, with all elements simultaneously present. But as soon as we attempt to read it aloud, in a serial way, complications arise. The verbal version would be: 'the square root of two hundred and fifty divided by three plus seven'. But this written formulation could be interpreted in several ways such as:

$$\frac{\sqrt{250}}{3}+7 \text{ or } \frac{\sqrt{250}}{3+7} \text{ or } \sqrt{200}+\frac{50}{3}+7 \text{ or } \sqrt{200}+\frac{50}{3+7}$$

Crystal is right. It is not possible to express in a sentence what this picture means without referring to the spacial disposition of its symbols. However, Iverson has demonstrated that its meaning can be rendered unambiguously as character strings in either APL or J:

```
7+(250÷3)*0.5   ⍝ APL

7+%:250%3       NB. J
```

For that matter this can be done equally well in Mathematica; as is shown by Ian's example of linear input for a quadratic expression. It looks like this:

```
(-b + Sqrt[b ^ 2 - 4ac])/(2a) /. (b=>0, c=>1)
```

which Mathematica simplifies, if given incomplete values, to:

$$\frac{\sqrt{-a}}{a}$$

or, if given all values thus:

```
(-b + Sqrt[b ^ 2 - 4ac])/(2a) /. (b#>0, c#>1, a#>2)
```

although it gives the result in CMN unless a numeric result is specified:

$$\frac{i}{\sqrt{2}}$$

An APLIWIN equivalent might look like this:

```
   ∇ r←quad coeffs;a;b;c;dsc;d
[1]   a←coeffs[3] ⋄ b←coeffs[2] ⋄ c←coeffs[1]
[2]   dsc←(b*2)-4×a×c  ⍝ discriminant
[3]   d←1 ¯1×dsc*0.5
[4]   r←(d-b)÷2×a
    ∇

      quad 1 0 2
0j0.7071067812 0j¯0.7071067812
```

While J looks like this:

```
   sel=:'a=:2&{'; 'b=:1&{'; 'c=:0&{' NB.values from
argument
   dsc=: (b^2:) - 4:*a*c NB. dsc is discriminant
   r=:(-@b(+,-)%:@dsc)%+:@a
   r cba=:1 0 2 NB. argument assumes ascending
polynomial
0j0.707107 0j_0.707107
```

These examples perfectly illustrate the nub of Iverson's criticism of conventional mathematical notation. Although, just as with the electronic diagram, all of its components can be named, what CMN lacks is a grammar. Its – sometimes very elaborate – arrangement on the page is a substitute for the conventional sentence structure (the syntax of the nouns, verbs and adverbs) of a natural language but, as Crystal shows, this is not sufficient to allow its meaning to be put into words without circumlocution. In this respect, therefore, Mathematica must be subject to the same constraints as APL/J or any other mathematical programming language. Its own specification for computer operations, whether they are to be numeric or symbolic translations, must be unambiguous and complete. Those favourites of mathematicians '...', rules of precedence for operators and so on, must be fully spelled out.

Ian says that, for Mathematica, everything is an 'expression' and an expression is 'a string of ASCII characters to be filtered', which it surely has in common with most programming languages, including J. APL uses an extended character set requiring a non-ASCII code, which we know to our cost has practical commercial disadvantages, but this is not enough to warrant Ian's opinion that Mathematica, 'contrasts strongly with the APL view of the world, where the whole time you're conscious of handling vectors of binary "num" elements.' Everyone knows I am a great admirer of Gérard Langlet and he famously championed vectors for most intermediate processing, but to suppose that this can be called 'the APL view of the world' seems perverse in the light of Iverson's own stated 'core ideas' [6]:

- The adoption from Tensor Analysis of a systematic treatment of arrays, in which every entity is an array, and different ranks lead to scalars, vectors (or lists), matrices (or tables), and higher dimensional arrays (or reports).

- Operators (in the sense introduced by Heaviside), which apply to functions to produce related functions.

What Mathematica does, which is outside the scope of APL/J, is to resolve algorithms in terms of the symbols themselves. So the result of the quadratic which Ian input in linear form is first displayed in CMN and then, if all values are specified, the result is displayed, simplified, but still as CMN symbols. If the numeric equivalent is wanted it must be specifically requested. As a tool for teaching and using CMN this must be invaluable but it is not needed for a general purpose programming language. Ian suggests that, by reformulating the symbolic algorithms, Mathematica bypasses some of the imprecision which may be introduced by numeric resolution of intermediate results, of the sort described by Donald McIntyre [7]. Gérard Langlet [8] cites Mathematica as a useful example of software which allows a wide range of user-defined precision.

Those of us who have grown up with computing since the 1950s can fully sympathise with Ian's impatience with legacy systems, incorporating successive attempts by competing vendors to engineer a GUI interface for their APL products; but that Mathematica, dating from 1988 and from a single Vendor, is less cumbersome, should scarcely be a surprise to him. The question at issue is whether Mathematica should now be his software of choice for programming a 'sophisticated application'. I want to answer this with a question. "Is there anything about your application that requires more than simple arithmetic?" If nothing, then base your choice on purely commercial considerations – of price and support and availability of programmers and concentrate your attention on the computer/user interface irrespective of software. On the other hand, if there are subtleties of algorithms and/or it is the users who are mathematically sophisticated, one must consider their level of commitment to the conventional notation.

Ian is obviously very comfortable with CMN and wants to use it as a teaching medium. For this purpose Mathematica may be his ideal tool. The Chinese have no intention of abandoning their ideograms for use within their own community, although they are also intent on communicating with a world becoming technologically committed to the English language and the Roman alphabet. Analogously, if an application requires a fundamentally mathematical or, even more importantly logical, approach and the users are not committed to CMN; APL/J may well be the equivalent of English in the Roman alphabet, the new mathematical *lingua franca*.

As Ian explains it, there are aspects of input to Mathematica more like J than CMN. The 'linearised format' input displays in the standard keyboard typeface except that `#/>` gives right arrow. Alternatively, input can be by the graphics package device of selection from a palette of styles, which allow two-dimensional input, matching the CMN default output format. Unsurprisingly, the 'behind the scenes' text, which specifies the display of CMN on the screen, is as verbose and unfriendly as the Visual Basic controls set up by the GUI front-end to J.

Which brings us back to Ian's question: is it only a matter of prejudice or does their 'fitness for purpose' have any bearing on a choice between APL/J and Mathematica and, by implication, the many other tools on the market? I think it does, but that he is thinking only in terms of programming bespoke computer applications. The implementation of J is already itself a 'sophisticated application', and it is written in C. I have the impression that one popular use for Mathematica is to facilitate publication in mathematical journals and that another is to provide teachers of mathematics with a versatile tool with which to interact with their pupils. Let us remember that Iverson was, above all, a teacher who attracted his pupils from all walks of life. I think of APL as a philosophical tool, and J even more so. Iverson understood mathematics to be a set of formalisms chosen to add brevity and precision to natural language, which then evolved to become an international dialect. Nevertheless, mathematics cannot stand alone: it is rooted in and dependent upon natural language. Consider the page from Lanczos' *Applied Analysis* above. We can only understand the purpose of his CMN in the light of his verbal description of the relevance of Simpson's Rule to the method called *quadrature*, which he illustrates with a graph. He might have welcomed Mathematica as a manuscript-preparation tool but that would probably have been the limit of his need for a computer.

However, computers need control languages, which must incorporate both mathematical and logical terms. Iverson's notation supplied both and would have been a tool of choice, if the capacity of 1960s hardware had been up to the challenge. As it turned out, programming languages were devised using a 'starved palette' of words and abbreviations, somewhat reminiscent of English, each having its own exotic grammar, requiring a telephone directory-sized textbook for its exposition. Ken studied the etymology of the language of mathematics and logic and, in the spirit of Esperanto, devised APL and J to be a versatile language, with a simplified and rigorously consistent grammar, in which to talk about arrays whether alphabetic, numeric or logical.

Iverson's greatest achievement was to extend the field of mathematics itself and an appreciation of this is very likely to be confined to a 'narrow élite'. It cannot be expected to engage the attention of most mathematicians unless they need both

computer power and the unique insights of APL/J. So my answer to Ian's question would be that the market for applications requiring sophisticated mathematics will always be small and I guess that there will be little competition between APL/J and Mathematica on technical grounds. Like the languages we speak, the programming languages we learn will be decided by the communities we are born into or decide to join. There is no reason why we should not be bilingual in CMN and APL/J but we must recognise that the formal structures we set up by the use of either will always be embedded in some natural language, with English leading the field and already ahead by a good many lengths.

References

[1] Ian Clark, "Review of Mathematica 5.2" *Vector* 22.4, pp60-78

[2] Roger Hui, "A Lifetime of Working with Ken" *Vector* 22.3, pp94-108

[3] David Crystal, *The Cambridge Encyclopedia of Language*, C.U.P., 1987, ISBN 0-521-42443-7

[4] Jan Gullberg, *Mathematics from the Birth of Number*, 1997, Norton & Co., ISBN 0-393-04002-X

[5] Cornelius Lanczos, *Applied Analysis*, Pitman, 1957, London.

[6] Kenneth E. Iverson, "APL in the New Millenium" *Vector* 22.3, pp5-12

[7] Donald McIntyre, "The Perils of Subtraction" *Vector* 11.4

[8] Gérard Langlet, letter *Vector* 12.1

DISCOVER

APL – A Glimpse of Heaven

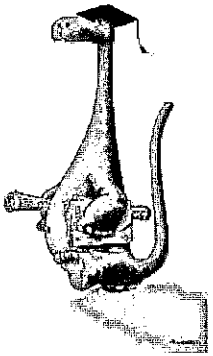
by Bernard Legrand (bernard.legrand@interfluences.fr)
translated by Sylvia Camacho

This document was created for a conference on 22 June 2006, organized by AFAPL, *Association Francophone pour la promotion du langage APL*: a special conference dedicated to our friend Henri Sinturel, now no longer with us.

This conference was not designed for those proficient in the language: their needs have been met for many years and I have nothing to teach them. I am only attempting to demonstrate this attractive intellectual tool to those who do not know it, in the hope of making one or two converts willing to promote it.

In fact the following presentation is intended, above all, to construct a bridge between two generations:

> The "diplodocus" generation familiar with information technology before the PC and the advent of screens (remember, they were the same ones who knew about punched cards!) and who found in APL a means of handling all the problems which large computers could not handle within a reasonable time. That generation is on the way to extinction.
>
> The generation who know only about micro-computers, the Internet and hypertext; on which they have been brought up, with substantial help from Excel, ever since their schooldays and who would like to think everything can be done with three clicks of a mouse.



But to describe the APL language, whether in 3 pages or 30, is as difficult as describing a tennis match or the flight of a seagull: a written document is not capable of matching hands-on experience. Thus the following pages only give a very limited and fragmentary view of the whole wealth of APL.

The abundance of APL riches is a glimpse of heaven. Here's to you, Henri!

I hope to stay true to the spirit of APL, and to the enthusiasm of those who have praised and promoted it over the years. I shall try also to reply to the most frequently asked questions and the most often expressed criticisms, but I cannot encompass all aspects of the debate. So I invite all those who are curious about the language to spend an hour with me at a computer screen.

Fasten your seatbelts: we're off!

The First Step is Easy

In the following pages, for maximum clarity, the text typed in by the user is indented, whereas the computer's response begins at the left margin.

The first impression of APL is that of a hand calculator:

```
      27 + 53
80

      1271 - 708
562

      59 × 8
475

      86 ÷ 4
21.5
```

The first surprise is that multiplication is represented by its proper symbol × and not by the abominable star * which prevails in all other computer languages. The same goes for division.

To create a variable it is enough to key in the chosen name and to show by the arrow ← that it is to be given the value or values which follow.

For example:

```
      VAT ← 19.6     ⍝ read as: VAT gets 19.6

      Years ← 1952 1943 1986 2005
```

To learn the contents of a variable it is sufficient to type its name thus.

```
      Years
1952 1943 1986 2005
```

A Global Approach

It is characteristic of APL that it can operate simultaneously on two sets of numbers of the same "shape". Below, for example, there are two variables, a list of prices of 5 products and the quantity bought of each:

```
      Price ← 5.2  11.5  3.6  4  8.45

      Qty   ← 2    1     3    6  2
```

When the two variables are multiplied together, they are multiplied item by item to produce a result of the same shape:

```
      Costs ← Price × Qty
      Costs
10.4 11.5 10.8 24 16.9
```

This global approach eliminates most of the "loops" which greatly overburden programs in most current languages.

This property remains true for arrays of values of the same shape or number of dimensions.

So a Sales Director can make forecasts for sales of 4 products over the coming 6 months, and assign them to the variable `Forecast`.

At the end of 6 months he can assign the somewhat different real values to the variable `Actual`.

```
      Forecast                    Actual
150 200 100  80  80  80     141 188 111  87  82  74
300 330 360 400 500 520     321 306 352 403 497 507
100 250 350 380 400 450     118 283 397 424 411 409
 50 120 220 300 320 350      43  91 187 306 318 363
```

It is clear that the first reaction of our Director will be to evaluate the differences, which he can get very easily by writing:

```
      Actual-Forecast
¯9 ¯12 ¯11   7   2  ¯6
21 ¯24  ¯8   3  ¯3 ¯13
18  33  47  44  11 ¯41
¯7 ¯29 ¯33   6  ¯2  13
```

Note that the sign is part of the number. Negatives take a high minus to distinguish the sign of a negative value from subtraction.

To get a similar result by means of a traditional computer language requires many instructions, which hides the object of the calculation behind arcane programming. Here, for example, is what one would write in PASCAL:

```
DO UNTIL I=4
  DO UNTIL J=6
    DIFF(I,J):=ACTUAL(I,J)-FORECAST(I,J)
  END
END.
```

Ah well, believe it or not, one can find at the heart of French university establishments people who maintain that the second way of writing these things is the simplest, those who resurrect the well-known Shadoks maxim: [*Trans*: a 1960s French cult animation series, according to Google]: Why make it simple when one can make it complicated?

We have seen that APL works between two variables of the same shape; it also works between such variables and a single item, which is called a scalar.

Such will be the case if one wants to calculate the total of 19.6% VAT applied to the variable `Price` above.

Whether one writes `Price × 0.196` or one writes `0.196 × Price` the result will be the same:

```
      1.0192 2.254 0.7056 0.784 1.6562
```

A result which will require rounding, but that is not important here.

New Symbols

Human intelligence is not confined to four or five basic operations, though that is truly the limit imposed by the majority of, even the most modern, programming languages.

The inventive creator of APL, Kenneth E. Iverson, therefore added to the set of all the usual symbols a number of new ones:

The symbol for the function *maximum* returns the greater of two numbers, or of two arrays of numbers compared item by item. There is also, as one might expect, a symbol for *minimum*.

```
      75.6 ⌈ 87.3
87.3
```

```
      11 28 52 14 ⌈ 30 10 50 20
30 28 52 20
```

The function *Minimum* works the same way:

```
      11 28 52 14 ⌊ 20
11 20 20 14
```

APL supports about 70 symbols. Consequent upon certain symbols being defined in two ways one could argue at length about the exact number.

That is nothing to worry about: some of the symbols are familiar (such as × or ÷ or again + and -, but also ! and a good many others).

Double Meaning of Symbols

This is not a peculiarity of APL; within algebra we are familiar with the use of symbols as common as the minus sign being used in two ways.

In the expression $a = x-y$ it means the operation subtraction, whereas in $a = -y$ it is the result of the *inverse* operation, which changes the sign of the number.

The first form is called the *dyadic* use of the symbol. The second form is called the *monadic* use of the symbol.

It is the same in APL, where most of the symbols can have two meanings.

For example, to find the shape (the dimensions) of an object, one uses the Greek letter ρ (*rho*), which can be read "shape of", in its monadic use.

```
      ρ Price      ⍝ Monadic: Price comprises 5 items
5

      ρ Forecast   ⍝ Monadic: Forecast comprises 4 rows of 6 items
4 6
```

Used dyadically, the same symbol will organise values into the specified shape.

```
      2 3 ρ 1 2 3 4 5 6
1 2 3
4 5 6
```

For example, to create the table below requires two pieces of information:

```
25 60
33 47
11 44
53 28
```

first the *shape* to give to the table: `4 2` (4 rows of 2 columns), and next the *contents* of the table: `25 60 33 47 11` etc.

It is the symbol *rho* which makes the connection:

```
Tab ← 4 2 ⍴ 25 60 33 47 11 44 53 28
```

A new variable `Tab` is thus created, and this is also how the variables `Forecast` and `Actual` above were made.

Conventions

In APL, one gives the name *vector* to a list of values; which may be composed of numbers like `Price` and `Qty`, or of letters like `'Once upon a time'`. One calls a table of two dimensions, like `Forecast` or `Tab`, a *matrix*, and a single value, a number like `456.18` or a single letter like `'Q'`, a *scalar*.

Reduction: a Linking Notation

Remember that we calculated some costs: `10.4 11.5 10.8 24 16.9` .

So what must we do to work out the total? Mathematicians are creative people who long ago devised the symbol Σ, always with a pretty collection of indices above and below, which is not very compatible with elementary text processing, which must put symbols on a single line.

In APL, the operation is written thus:

```
      +/ Costs
73.6
```

Simple isn't it? What's the point of the indices of the first and last items? This gives the total of *all* the items of the data without mentioning them!

One speaks of the *plus reduction* of the variable `Costs`. To gain a better understanding of the process:

When we write an instruction such as `+/ 21 45 18 27 11`
it is as if we wrote `21 + 45 + 18 + 27 + 11`
to obtain the sum `122`
in fact as if we had *inserted* the symbol + between the values.

But then, if we write `×/ 21 45 18 27 11`
it is as if we had written `21 × 45 × 18 × 27 × 11`
and obtained the product `5051970`

Similarly, if we write `⌈/ 21 45 18 27 11`
it is as if we wrote `21 ⌈ 45 ⌈ 18 ⌈ 27 ⌈ 11`
and so had obtained the largest item `45`

Reduction belongs to a special category of symbols called *operators*.

The other symbols (`+ - × ⌈ > ⍴`...) represent *functions* (addition, subtraction ... maximum ... shape, etc.).

The arguments of a function are *data*: `Price × Qty`

Whereas the left argument of an operator is a *function*: `+/ Costs`

(A more rigorous definition is beyond the simple framework given here.)

We may say that reduction allows as many different operations to be carried out as there are symbols for functions (or program names!) to its left: it is an idea of great generality.

Just think: in fact in mathematics we invented Σ for the sum, Π for the product, *min* and *max* for the minimum or the maximum, and still more notable *inf* (lower bound) and *sup* (upper bound)!

In APL, the sole symbol / suffices to regularise all this notation!

APL contains six mathematical operators in its most basic versions, and nine in the version of Dyalog APL which is used for this document.

First Program

We want to calculate the average of the following numbers:

```
Val ← 22 37 41 19 54 11 34
```

We must divide one expression by another:
first for the *sum* of the values: `+/Val` which gives `218`
next for the *number* of values: `⍴Val` which gives `7`
The calculation can be written as the single formula: `(+/Val)÷(⍴Val)`

As it is quite likely we shall often want to make this sort of calculation, it is preferable to store this sequence in the form of a program.

In APL we prefer the name *defined function* to the name *program*.

Defined functions are produced by a feature of the language which builds them so that they may be used in the same way as the symbols (+ - × ⌈ > ⍴...) which are called *primitive functions*.

It is outside the scope of this document to explain how to define such a program: having said which, it will look something like the following:

```
    ∇ R ← Average V
[1]   R ← (+/V)÷(⍴V)
    ∇
```

`Average` is the program name.
`V` represents the list of values which are defined as the right argument.
`R` represents the result of the calculation, which will be returned at the end.

The typographical sign ∇ (called *del*) marks the beginning and end of the printed form of the program.

Once defined, this function may be invoked in a very simple way:

```
      Average Val
31.1428571428

      Average 12 74 56 23
41.25
```

One may thereafter include it in a more complex expression:

```
      10×Average 12 74 56 23
412.5
```

Indexing

Returning to our vector of numbers: `Val← 22 37 41 19 54 11 34`

In order to extract the fourth item, we write `Val[4]`

In other languages one uses parentheses instead of square brackets; this is not very different.

What is new is that one can extract *several* items in one instruction.

```
      Val
22 37 41 19 54 11 34

      Val [2 4 7 1 4] ⍝ note extracting the same item
twice
37 19 34 22 19
```

And, of course, in the same way one may modify one or more items of `Val` designated by their indices, providing as many values are specified as there are items to modify, or a single value for all (modified items in bold type):

```
      Val[3 5 1] ← 300 77 111

      Val
111 37 300 19 77 11 34
```

It is often necessary to extract the first items from a list of values, for example the first 5. Nothing could be easier:

```
      Val[1 2 3 4 5]
111 37 300 19 77
```

But if one needs to extract the first 500 items from a long list, typing the integers from 1 to 500 is naturally impossible.

This is why APL has been given the symbol ⍳ (*iota*), which produces the set of the first *n* integers.

Thus, instead of writing `1 2 3 4 5 6 7 8`, it is sufficient to write `⍳8`. And to extract the first 500 terms of a large vector, one may write `Big[⍳500]`.

Calculating without Writing Programs

The twenty salaries of a business are divided into three hierarchical categories, denoted simply `1 2 3`.

One assigns to two variables the salaries and the categories of these salaries; a part shown here:

```
Salaries ← 4225 1619 3706 2240 2076 1389 3916 3918 4939 2735
Categories ← 3    1    3    2    2    1    3    3    3    2
```

Do they never want to upgrade these salaries? (What has our poor world come to!)

A rumour reaches us about their plans: they want a different percentage increase for each category, according to the following scale:

Category	Required Upgrade
1	8%
2	5%
3	2%

How much is that going to cost the business?

We create a variable containing the above three rates, recalling that we can divide three numbers by a single number:

```
      Rates ← 8 5 2 ÷ 100

      Rates
0.08 0.05 0.02
```

Then, as the first salary is in category 3, the rate which applies to it is:

```
      Rates[3]
0.02
```

It follows that the first five salaries, being in categories 3 1 3 2 2 respectively, require the following upgrades:

```
      Rates[3 1 3 2 2]
0.02 0.08 0.02 0.05 0.05
```

More generally, the rates applied to our twenty salaries are obtained like this:

```
      Rates[Categories]
0.02 0.08 0.02 0.05 0.05 0.08 0.02 0.02 0.02 0.05 0.05 ...
```

Having the 20 rates it suffices to multiply by the 20 salaries to obtain the individual up-grades:

```
      Salaries × Rates[Categories]
84.5 129.52 74.12 112 103.8 111.12 78.32 78.36 98.78 ...
```

Finally, by adding them all, one will know how much it will cost the business:

```
      +/ Salaries × Rates[Categories]
2177.41
```

One notes that:

- this method remains valid whatever the number of salaries or categories.
- the result has been obtained without writing any program.
- and this expression can be read as the simplest possible English: The sum of Salaries multiplied by Rates according to Categories.

This example shows clearly that there are ways of reasoning other than those which have dominated information processing for 40 years but they are, alas, still extremely rare. This difference and originality, introduced by APL, are major

features. They typify the open and welcoming intellectual spirit of the people who practise it.

Our Binary Friends

APL makes much use of binary data. It is most often created by means of relational functions such as = or > .

```
      Salaries > 3000
1 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1 0 0 0 0

      Actual > Forecast
0 0 1 1 1 0
1 0 0 1 0 0
1 1 1 1 1 0
0 0 0 1 0 1
```

One sees the favourable results instantly. It is a prime novelty of APL that it is the *only* computer language we know of which has the six relational functions, represented in their conventional mathematical form: < ≤ = ≥ > ≠ .

For sure, other languages manage somehow but it seems to us, at the beginning of the 21st century, not totally unreasonable to ask that the inequality ≥ should not be represented as => and that ≠ should not be represented by the diabolical <>!

Naturally one can operate on this binary data using all the functions of Boolean algebra and, moreover, the symbols used are those familiar, throughout the world, to mathematicians of all nationalities:

Function *and* is well known ∧ (denoted AND in many languages).

Function *or* is well known ∨ (denoted OR in these languages).

Thus, if I am looking for people in category 3 whose salary is less than 4000 euros, I can write:

```
      (Categories=3) ∧ (Salaries<4000)
0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 1
```

In fact APL offers *all* the functions of Boolean algebra, including some functions like NOR and NAND (Exclusive OR and Exclusive AND) not familiar to managers but very useful in electronic automation.

Incidentally, Exclusive OR (sometimes called XOR) can be simply ≠ because either symbol acts like Exclusive OR (either not both):

```
      0 0 1 1 ≠ 0 1 0 1
0 1 1 0
```

Finally: these binary vectors can be used as we have described but also for novel purposes, as good tools for denumerating and selecting.

Denumeration

Having found which salaries are less than 2500 euros by means of the following expression:

```
      Salaries<2500
0 1 0 1 1 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0
```

it is easy to add all the 1s and 0s to calculate how many people earn less than 2500 euros:

```
      +/Salaries<2500
8
```

Selection

One can also use the binary vector as a "mask" to select from other data those items corresponding to the binary 1s:

```
      1 1 0 1 0 0 1/23 55 17 46 81 82 83
23 55 46 83
```

The procedure is identical for text data:

```
      1 1 0 1 0 0 1/'Bernard'
Bend
```

This operation, called *compression*, is particularly useful for extracting from a variable the items conforming to a given criterion. For example, to display the salaries in Category 2, one writes:

```
      (Categories=2)/Salaries
2240 2076 2735 3278 1339 3319
```

Powerful, isn't it?

Discovery

To practise our skill some more, let us find in our variable `Val` the positions of numbers greater than 35.

Here are the stages of our journey:

```
      Val
22 37 41 19 54 11 34

      Val>35
0 1 1 0 1 0 0

      ⍴Val
7

      ⍳⍴Val
1 2 3 4 5 6 7
```

Let us compare two of these results:

```
      Val>35
0 1 1 0 1 0 0

      ⍳⍴Val
1 2 3 4 5 6 7
```

One sees that if one eliminates (by a compression) the terms which correspond to zeros in order to retain those corresponding to 1, one easily gets the positions required: 2 3 5. Thus the job may be done as follows:

```
      (Val>35)/⍳⍴Val
2 3 5
```

This expression applies in many different situations.

Here is a similar use, but applied to text data: to find the positions of 'a' within a phrase; the method is the same.

```
      Phrase ← 'The Argentinian tango is not in fashion'
      (Phrase='a')/⍳⍴Phrase
14 18 34
```

Keep it Dark!

Proudly having found all the 'a's, we may wish to find all the vowels.

Alas, although we can write `(Phrase='a')`, because a vector can be compared with a single value, one cannot write `(Phrase='aeiou')`, because that would require the item by item comparison of a phrase of 39 letters and `'aeiou'` which has only 5.

Well, one may compare 39 letters with 39 other letters, or compare them with one only, but not with 5.

So we must have recourse to a new function: *membership* denoted ∈, also used in mathematics.

The expression A∈B shows, by a binary result, which elements of the variable A appear (wherever they may be) in the variable B. And that works no matter what the shapes, dimensions or type of data of A and B, a small marvel!

For example:

```
      5 7 2 8 4 9 ∈ 3 4 5 6
1 0 0 0 1 0

      'pissenlit'∈'jardin'
0 1 0 0 0 1 0 1 0
```

(Only the 'i's and the 'n' appear in 'jardin'.) So in pursuit of our enquiry we shall write:

```
      (Phrase ∈ 'aeiou')/⍳⍴Phrase
3 8 11 13 14 18 21 23 27 30 34 37 38
```

One can also use membership between a vector and a table (matrix), as shown below (the list of towns is a variable created earlier).

```
      Towns

Martigues
Paris
Strasbourg
Granville
Nantes
Fréjus

      Towns ∈ 'aeiouy'
0 1 0 0 1 0 1 1 0 0 0
0 1 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 1 1 0 0 0
0 0 1 0 0 1 0 0 1 0 0
0 1 0 0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
```

Note that the result has always the same shape as the data at the left:

```
      'aeiouy' ∈ Towns
1 1 1 1 1 0
```

None of the towns contains a 'y'.

Making a Point

Programmers have often said to me that the symbol ∈ constitutes for them the one irrefutable proof that APL is an advanced mathematical language, not suitable for all users. I am inclined to agree this is the general view, are you?

Even if it has only been used in education since the beginning of the 60s, this symbol has appeared (if I am not mistaken) as part of the apparatus of "modern" mathematics since the second half of the 19th century. Knowing that we are in the 21st century, we can say that it is at least 100 years old.

What is more, when my elder son was 11 years old, membership was taught to Year 7 mathematics classes. I can say, having followed his learning closely, that it did not pose any greater difficulty than if that lesson had been delayed until much later.

In other words, those learned programmers who find membership too difficult to understand are unreasonably claiming that things which have been within the powers of a child of 11 for the past 100 years, have suddenly become too advanced and difficult.

In these circumstances I see that one cannot put APL into anybody's hands; in particular those of these programmers. But is this a fair criticism of APL?

A Generally Powerful Function

We have a very useful method to look for the positions of letters or numbers in a vector, but it has some small problems we have not yet covered. There is another way, which uses the dyadic form of the symbol ⍳ (iota).

```
      Vec ← 15 40 63 18 27 40 33 29 40 88 ⍝ vector to
search

      Vec ⍳ 29 63 40 33 50                ⍝ values
sought
8 3 2 7 11
```

It is the case that 29, 63, 40 and 33, occur respectively in positions 8, 3, 2 and 7.

The first surprise: the value 40 occurs three times in `Vec`, but only the first occurrence is mentioned. If the response for each value sought has to be a position; how may one, looking for five numbers, obtain seven results?

Second surprise: the value 50 assigned position 11 ... in a vector comprising only ten items! This is how the function *index of* (dyadic ⍳) reports that a value is absent.

At first sight that seems strange but in fact it is the characteristic which makes this function so generally powerful.

An Example

A motor manufacturer decides he will offer his customers a discount on the catalogue price. (Now you know that this example is imaginary!)

The discount rate will depend on the geographic area according to the following table:

Area	Discount
17	9%
59	6%
84	5%
89	4%
Other	2%

The problem is to calculate the discount rate which may be claimed for a potential customer who lives in area `D`; for example `D←84`.

Let us begin by creating two variables:

```
      AREA  ← 17 50 59 84 89
      DISCT ←  9  8  6  5  4  2
```

Let us see if 84 is in the list of favoured areas:

```
      AREA⍳D
4
```

84 is all right: the 4th item in the list. Let us find the current rate of discount for this index position:

```
      DISCT[4]
5
```

This customer can claim a 5% discount; that's good. One may simply write: `DISCT[AREA⍳D]`.

If a customer lives in any area such as 75, 45, or 93, the expression `AREA⍳D` will give in all cases the response 6, and `DISCT[6]` will always find the rate 2%, as intended.

The importance of this approach is that it is vector-based. Suppose that publicity attracts crowds and that therefore D is no longer a scalar but a vector, the solution is still valid:

```
      D←24 75 89 60 92 50 51 50 84 66 17 89
      DISCT[AREA⍳D]
2 2 4 2 2 8 2 8 5 2 9 4
```

All that without a program, neither "loop" nor "test"; readers who know other programming languages will have no difficulty in making the comparison.

Generalisation

In truth, the expression we just wrote is an example of an algorithm for "changing the frame of reference". Not to panic, the name is recondite, but the concept is simple: a list of area numbers (the initial set) is translated into a list of discount rates (the final set).

Let us imagine the initial set to be an alphabet composed of lower case and upper case letters, and the final set to be composed of only upper case letters:

```
      Almin
abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ

      Almaj
ABCDEFGHIJKLMNOPQRSTUVWXYZ ABCDEFGHIJKLMNOPQRSTUVWXYZ*

      Fable ← 'Le petit Chaperon-Rouge a bouffé le Loup'
```

The expression below converts from lower to upper case.

```
      Almaj[Almin ⍳ Fable]
LE PETIT CHAPERON*ROUGE A BOUFF* LE LOUP
```

As one might expect, the characters - and é, which are absent from the initial alphabetic set have been replaced by the * of the final set, but the conversion is acceptable. This solution can easily be improved.

Once more, the rational steps to be taken to resolve a computing problem are entirely different from the ways traditionally taught, and the programmer has thereby gained a much more extensive insight.

From Foundation to Structure

Traditional computing languages do not handle tables of numbers. They hold them in memory, but when they are required for processing they can only handle them one number at a time. It is unsurprising in these circumstances, that these

languages do not concern themselves with the difference that might result from controlling the shape of the data.

It is quite otherwise with APL, which offers many tools for working with the shape of the data. We shall only look at a few of them.

Take and drop

The functions Take (↑) and Drop (↓) serve, as their name suggests, to take or drop part of a variable. Here we shall show only examples based on vectors, but all the other shapes of data can be treated in a similar way.

Recalling that Vec has values 15 40 63 18 27 40 33 29 40 88

```
      4 ↑ Vec        ⍝ take the first 4 items
15 40 63 18

      5 ↓ Vec        ⍝ drop the first 5 items
40 33 29 40 88
```

If the left argument is negative, these same functions count from the end of the vector.

```
      ¯3 ↑ Vec       ⍝ take the last 3 items
29 40 88

      ¯7 ↓ Vec       ⍝ drop the last 7 items
15 40 63
```

If one drops the last 7 items, it leaves only the first 3, which we could have accomplished with 3↑Vec. To have two functions seems unnecessary. What purpose does this serve?

Let us imagine a business with a turnover which has grown over 12 years. The variable Tome is turnover in Millions of Euros.

```
      Tome
56 59 67 64 60 61 68 73 78 75 81 84
```

We want to calculate the difference between each year and the next; how to do it?

```
      1↓Tome
59 67 64 60 61 68 73 78 75 81 84

      ¯1↓Tome
56 59 67 64 60 61 68 73 78 75 81
```

We see that all that remains is to subtract item from item:

```
      (1↓Tome)-(¯1↓Tome)
3 8 ¯3 ¯4 1 7 5 5 ¯3 6 3
```

Without a program or loops; all very simple!

In place of a subtraction, one division calculates the rates of growth instead of the differences, with some obvious adjustments:

```
      100 × ((1↓Tome)÷(¯1↓Tome)) - 1
5.35 13.56 ¯4.48 ¯6.25 1.67 11.47 7.35 6.85 ¯3.85 8 3.70
```

Mirrors and Transpositions

APL is also well endowed with functions which pivot data about any axis, as suggested by the shape of the symbol used. It applies both to numeric and text data; as we are going to show by applying these functions to the variable `Towns` met above.

Initial Variable	Reverse left-right (Mirror)	Reverse top-bottom (Flip)	Exchange rows & columns (Transpose)
`Towns`	`⌽Towns`	`⊖Towns`	`⍉Towns`
`Martigues`	`seugitraM`	`Fréjus`	`MPSGNF`
`Paris`	`siraP`	`Nantes`	`aatrar`
`Strasbourg`	`gruobsartS`	`Granville`	`rrrané`
`Granville`	`ellivnarG`	`Strasbourg`	`tiantj`
`Nantes`	`setnaN`	`Paris`	`issveu`
`Fréjus`	`sujérF`	`Martigues`	`g biss`
			`u ol`
			`e ul`
			`s re`
			`g`

The symbols used ⌽ ⊖ ⍉ are self-describing, no effort is required to remember any of them. They also have dyadic uses, with different but always interesting results.

Back to Primary School

Remember when we learned our multiplication tables. In that practically palaeolithic era, to make sure that we knew all our tables, my instructor made us calculate the multiplication table for the integers 1 to 9:

```
×  1  2  3  4  5  6  7  8  9

1  1  2  3  4  5  6  7  8  9

2  2  4  6  8 10 12 14 16 18

3  3  6  9 12 15 18 21 24 27

4  4  8 12 16 20 24 28 32 36

                ...         ...
```

You see, I haven't forgotten! Probably you have done all this just like me. And then we quickly forgot the imposition: thus sidestepping a very powerful tool, one which APL provides, under the name *outer product*.

The task consists of taking pairs of items of two vectors, (the column and row headings) and making them the left and right arguments of the function at the top left. Then we shall go on to see what we get if we change the values a little:

```
×    8   5  15   9  11  40

5   40  25  75  45  55 200

4   32  20  60  36  44 160

10  80  50 150  90 110 400

3   24  15  45  27  33 120
```

This operator is written thus in APL:

```
      5 4 10 3 ∘.× 8 5 15 9 11 40
40 25  75 45  55 200
32 20  60 36  44 160
80 50 150 90 110 400
24 15  45 27  33 120
```

Now imagine replacing the symbol for multiplication by any of a number of other functions, or programs which you could have defined yourself, and you will

understand, as for reduction already encountered, that outer product is an operator of amazing power.

Let's have fun:

```
(⍳5)∘.=(⍳5)   (⍳5)∘.<(⍳5)   (⍳5)∘.≥(⍳5)

 1 0 0 0 0     0 1 1 1 1     1 0 0 0 0
 0 1 0 0 0     0 0 1 1 1     1 1 0 0 0
 0 0 1 0 0     0 0 0 1 1     1 1 1 0 0
 0 0 0 1 0     0 0 0 0 1     1 1 1 1 0
 0 0 0 0 1     0 0 0 0 0     1 1 1 1 1
```

And go on to practical applications.

An Example

Suppose the vector `Ages` contains the ages of 400 respondents to an opinion poll. We want to establish how many people there are in each of the following categories.

0 – 25 – 30 – 35 – 45 – 50 – 55 – 65 or above.

In addition we decide that those who are on a borderline will be assigned to the lower category.

Here is an extract of the data:

```
Ages ← 32 19 50 33 23 65 46 26 31 58 51 23 51 36 28 42
Category ← 0 25 30 35 45 50 55 65
```

We are going to invoke the outer product `Category ∘.< Ages`, and here are the first items calculated as shown above:

```
 <  32 19 50 33 23 65 46 26 31 58 51 23 51 36 28…
 0   1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
25   1  0  1  1  0  1  1  1  1  1  1  0  1  1  1
30   1  0  1  1  0  1  1  0  1  1  1  0  1  1  0
35   0  0  1  0  0  1  1  0  0  1  1  0  1  1  0
 …   …
```

If one adds up this binary table, one obtains for each row the number of people who are older than 0 years, 25 years, 30 years, etc. This is the expression:

```
cum ← +/Category∘.<Ages
```

The example above is reduced to the value cum which is: 17 14 12 8 6 4. In other words there are 12 people older than 30. But among them there are 8 older than 35. In order to know how many people are between 30 and 35, it is necessary to perform 12-8, to give 4.

If one wants to reproduce this calculation for all categories, it is necessary to perform a series of subtractions as here:

```
      cum
17 14 12 8 6 4

      1↓cum,0
14 12  8 6 4 0

       cum-1↓cum,0
 3  2  4 2 2 4
```

So, as we shall have occasion to notice again, we find that the comma joins variables together. This is a function called *catenation*.

If one no longer works with a small extract of data but with the 400 people, this is what one has:

```
      cum←+/Category∘.<Ages
      cum-1↓cum,0
83 65 79 79 32 17 36 9
```

All that without proper programming, and it works whatever the number of people or categories... what luck!

The outer product allows us to find typical solutions to some very classical problems.

I Have not Told you Everything

In the course of these pages we have flown over APL country and glimpsed certain bold ideas which explain the attraction of the language. A thousand other things remain to be seen!

It would be possible to talk about *inner product*, which is an extensive generalisation of matrix multiplication, of which our students retain only some formulae, learned by heart as a never-well-understood rigmarole, to reinforce

"sigma of A_{ij} B_{jk}". But realistically, among life's continuing problems, WHAT PURPOSE is served by stuffing our brains this way?

Having used APL as a teaching tool, I can assure you that one can teach linear algebra in a fast and realistic way, and show students that one can use it for comparing the management accounts of two companies. After that they will never forget it!

We ought to talk about generalised arrays, about the *execute* function, about... and it would take 400 pages... that is not our intention here.

Allow me to give you a quick illustration of a very unusual use: for files of data in an *inverted* form.

The Structure of Data

In the course of the preceding pages it has been assumed that the only appropriate way to organise data in APL is to group it by type: one variable contains names, another salaries, another codes, etc ...

Now, this is not at all the way that information is organised in traditional text files. Take, for example, a personnel file. In each line of text there is, one after another, the fields of data for the given individual: surname, forename, code, salary, etc. So that it looks a bit like this:

```
Sabatier      Eugene      1 1933  2997 E D 4 2737 93 C
Depond        Alain       1 1943  1732 E C 0 1489 77 C
Laure         Rose        2 1967  3813 E D 0 2082 75 C
Japroutsy     Véronique   2 1962  3115 E M 3 1934 77 U
Perdoux       Véronique   2 1961  1685 M D 0 2559 94 U
Trinque       Kate        2 1968  1747 E C 0 2902 92 P
Foucault      Jean        1 1934  2962 M M 3 1641 94 U
Fossey        Nicole      2 1961  2370 E C 0 1640 94 A
Boudinoy      Juliette    2 1945  2705 E M 4 1131 75 U
Louvier       Laurence    2 1932  1972 E M 2 2228 93 U
```

In data organised this way, the numeric information (salary, date of birth, ...) is encoded as text: one can only calculate with it after conversion to numerals. Handling it in this form will be a heavy burden for APL.

The sample file above contains 11 fields for 1000 people; thus the file has 1000 "records".

APL does it like this: one cuts the contents of the file into 11 columns, each containing only one type of information (surname, forename, etc.), one converts

the numeric data into numbers and then records each type of information, as one of 11 records, in a new file.

Thus one converts the 1000 occurrences of 11 disparate fields into 11 occurrences of 1000 homogenous fields.

This is why the new file is described as an *inverted file*. (One speaks sometimes of a *vector-file*.)

In practice, things are a bit more complex. When the number of people becomes very large (for example 500,000), it is unwise to hold 500,000 values as a single record. One segments the record and puts, for example, 10,000 salaries in each of 50 records, then the dates of birth into 50 records of 10,000, etc.

How is it used?

If one has small, simple variables it is easy to treat them as seen in earlier pages. I will show you how to file all or part of the data by writing short programs permitting incomparably flexible interrogation.

For example, to extract people with salary (variable `SAL`) between 1800 and 3500 euros and for whom the marital status (variable `SIF`) is 'M', one could write:

```
Select Staff (SAL Between 1800 3500) And (SIF = 'M')
```

(Just in the section that follows, functions are distinguished from variables by *italic type*.) The result might take the following form:

Forename	Surname	Sex	DoB	Salary	Status	SiF	Dept
Véronique	Japroutsy	2	1962	3115	E	M	77
Jean	Foucault	1	1934	2962	M	M	94
Juliette	Boudinoy	2	1945	2705	E	M	75
Laurence	Louvier	2	1932	1972	E	M	93

Thanks to small functions (programs) like *Select, Staff, Between, And,* but also: *Or, Save, Select, All, Decile,* one can easily interrogate the data. One can, of course, freely add to the vocabulary.

But, you say to me, this is not a large project, handling variables relating to a 100 or 200 people. What would happen if one had to deal with 10,000, 100,000, or even more people?

This is where inverted files are justified.

In fact, one can erase the small variables, (`SAL`, `SIF`, `ENF`, etc.) and then create the equivalents as small programs, each of a single instruction which will read the

corresponding information from the inverted file and to which we will give the same names as the erased variables (*SAL, SIF, ENF*).

In other words, the act of calling *SAL* fixes the contents of the variable `SAL`, which used to hold a few dozen salaries. Now, when one calls *SAL* one executes a program which reads the inverted file and returns several dozens of thousands of salaries.

The user's normal practices are not upset: he can continue with his armoury of small enquiry programs. He can also increase their range: a program which works on a variable of 10 or 20 values will work just the same on 10,000 or 100,000.

Didn't I tell you APL is magic?

FAQ

I am going to finish by responding to some questions I have often been asked. I am speaking for myself only: I do not lay claim to special expertise.

APL: is it a professional tool?

I will mention three examples of which I or my associates have experience:

- Long-term Board level planning for the TOTAL group, working with them over 12 years.
- The management of supplies required from 'today + 2' to 'today + 3 months', by the assembly lines of the 6 principal factories of the Renault group.
- Risk Management for the Allianz-AGF group.

These three have common characteristics placing them at the level of major industrial applications.

- They are particularly crucial because considerable finances are at stake.
- They must be absolutely reliable. A major Renault works such as Flins or Sandouville must not be brought to a stop by a computer bug.
- The first two are extremely changeable: as their requirements are always changing, the programs undergo constant mutation.

So I reply: yes, for a reasonable cost in labour, APL makes possible large, sensitive applications of the highest level of quality and reliability.

What niche does APL occupy today?

The niche for APL is any applications which are urgent and changeable, these characteristics usually going together.

Traditional development teams only work for contracts which require at least six months of planning, after which the writing and testing will take as long again. It takes a considerable time to get what is asked for... and sometimes one does not even get that! Then system requirements change suddenly and one spends months of work on amendments.

Unfortunately some problems cannot wait. Some unforeseen events last two months or less, as was the case with the first Gulf War: that is to say, less time than it takes computer technicians to amend their programs to meet unexpected circumstances.

Great flexibility and speed is the true commercial foundation for APL. For with APL one can develop in direct contact with the users and involve them from the outset in the continual modification of the object of the development. Afterwards, as it continues to evolve, it is still the speed of development which makes APL a tool especially well adapted to changeable environments.

Is the language readable?

If APL were a specialist, complex language, it would only attract the "Boy Wonders" of IT, those with A-Grades in everything, whose horizons are limited by bits and bytes.

So it is paradoxical that the great majority of IT people have never really understood APL. Those who have used it successfully have very often not been computer-literate, or have only a slight knowledge ... and they have frequently learned APL in isolation. That is to say, the language serves anyone prepared to explore it without prejudice.

To believe that "plain language" programming would be more readable is Utopian, even intellectually dishonest. For if I say, "a linear function of a variable is equal to the sum of a constant and of the product of a variable and a second constant", it is incontestably English but completely obscure, even incomprehensible!

But if I now say $y=ax+b$ (a notation undoubtedly abstract and symbolic), I know I shall be understood by most of my hearers who have received a similar education. It is self-evident: it is all a matter of upbringing.

The 80 lines of C++ (or of Java, or whatever) which often replace 5 or 6 lines of APL, seem completely obscure to anyone who has never studied C++. It is necessary to compare like with like and stop judging APL in the light of the opinions of people who have not been willing to learn it.

Let us put it precisely. Would one accept the view of a lecturer, about a poem by Pushkin, that the poetry is bad; if he could not read Russian? Certainly not! It is the same if one asks programmers inexpert in APL to form a judgment concerning the readability of programs written in APL. Relying on their status as professionals, they assert that these programs are unreadable... and people believe them!

To convince? – an impossible task!

To be honest, I must admit that APL has a number of new symbols, which makes translation impossible for any uninitiated person. How can you expect a programmer brought up on C++ or PASCAL to be able to understand an expression such as: `R←((V⍳V)=⍳⍴V)/V` ?

And who will believe me when I say that this expression does not require any "reading" or "analysis" for an APLer. It is read and understood instantly, as a whole, just like the word "MUMMY" is fixed in our mind without having to read and interpret it letter by letter, as a small child does it.

Certainly, to understand "MUMMY" one must have learned to read; it is the same for APL, it is necessary to learn it. After all one learns C++ or PASCAL, so why not APL?

Because of its cryptic appearance, it is almost impossible to convince anyone who might become interested in the beauty of APL, simply by showing him (even as I have tried to do here) some subtleties and some attractive algorithms.

Do not try to convince anyone by showing that you can do with 10 symbols, what would take him 100 convoluted instructions: all the world prefers reading 100 lines of good (or even bad) English, to remaining dumb, faced with 10 Chinese ideograms! You will only convince those who are willing to learn.

How to learn it?

It is of no importance that one can simply key `2+2` on an APL keyboard to get the response `4`. It is a mistake to imply, as too many APL enthusiasts have done, that three days is sufficient time in which to learn and practise this language.

Beyond knowledge of the basic elements, correct APL usage assumes knowledge of methods for organising data, and ways specific to APL, of solving problems. That cannot be learnt in a hurry, in APL or any other language.

It is necessary to devote to APL the same time that one would devote to any other language (2 or 3 weeks) and to work with professionals who are able to teach the best practice.

Your Time is Come!

The diplodocus we know was condemned to extinction!

All of us who have believed in and still believe in APL would love to see renewed youth and vigour appear, capable of finding new openings for APL, and new credibility and legitimacy.

This is the challenge thrown at you: **your time is come.**

System Building with APL+Win

by Ajay Askoolum

reviewed by Ian Clark (earthspot2000@hotmail.com)

This book is about building "systems" using the APL+ interpreter sold by APL2000 Inc. to operate within the Microsoft Windows environment. "System" here means "application", not "operating system" as a developer might at first suppose. According to the Oxford English Dictionary a (computing) system is "a group of related hardware units or programs or both, especially when dedicated to a single application" – which is exactly how the author means it.

From a developer's point-of-view, MS Windows is a junk-box of parts from which the dextrous builder can, with modest effort, throw together a prototype, and with considerably more effort get it into a state fit to ship. By contrast the APL environment with `CLEAR WS` is highly generalised and tightly integrated: everything works with everything else every-which-way – and has done for the last 40 years. This enables you, the experienced APLer, to build functions from scratch with confidence and despatch. It also offers a strong disincentive to bolting-on ready-made components. Doing so is often far less satisfying than anything you might write yourself in APL, because you have to spend time getting used to each component's funny little ways.

However, what seems satisfactory to you during the process of development can appear less so to your customers when they come to adapt the application at some future date – and to the programmers they employ to do the job. The author approaches the topic very much from the point-of-view of a maintenance programmer. He has an understandably sour view of "legacy code": that is, old code built out of matchsticks (my term, not his). Matchstick systems soon looked dated and are not easily upgraded without considerable redesign. They can't take advantage of improvements to the platform. Conversely applications written to use Windows components reap the benefits from upgrades to those components free of charge and (generally) without pain.

The standard Windows application, i.e. one written in Visual Basic (VB), is largely stitched together from ready-made components. And that's all VB is good for, say its detractors. But Microsoft have put hundreds of man-years into tuning it up for that purpose, smoothly extending the language with dotted identifiers to handle the methods and properties of ready-made components and, most significantly, equipping it with a massive library of component documentation: exposition,

worked examples and "IntelliSense", all comprising a system of on-line help that is manifestly better-integrated than the assorted components of Windows itself.

The Windows gospel is that third-party software products to run on Windows should look as far as possible like extensions to the operating system, all using the same screen furniture and data access methods. The end-user is meant to leverage this to switch easily between different vendors' applications. Customers expect the Windows look-and-feel to be honoured as a mark of software quality, even if they don't necessarily like it. The book fulfils a real need in this regard, one which the reviewer too has spent years of his precious life in the service of.

The author has a message: legacy code is locked in the past and is a burden to maintain. "Real" Windows apps must call Windows DLLs for all essential services which the user can possibly see: screen interaction, national settings like decimal point/comma and thousands separator, date and time manipulation, data access. But, as the book makes clear, "real" Windows apps aren't the sole preserve of the MS Visual Studio stable of languages which includes VB. They can be written in APL – and here's how to do it.

One nevertheless has to accept that VB is the vendor's flagship language. More pertinently it is the reference language for all function-header descriptions and coding samples in Microsoft's online Help. All other languages feast in a sense on these crumbs from the rich man's table.

The author is at pains to promote APL+Win as at least as good a medium for coding COM as VB. This is a lost cause, as the author himself accepts at times: it challenges VB on its strongest territory. Exhibiting examples side-by-side of corresponding COM calls in VB and APL+Win like this:

```
VB:         Set objXL = CreateObject("Excel.Application")

APL+Win:    'objXL' ⎕WI 'Create' 'Excel.Application'
```

not only sends the message that VB is in its element, maintaining the illusion that COM is a natural extension to the Basic language, but that the APL+Win programmer can't help coding through a keyhole: a ⎕WI-hole, you might call it. The ⎕WI-hole actually offers an occasional advantage. Sometimes you badly need to see where COM stops and user-code begins. Generally this is in code written by somebody else. VB blurs that boundary, but APL+Win emphasises it: the user-command]WSLOC reliably listing every ⎕WI in your workspace.

Nevertheless the Windows interface of APL+Win is in all honesty a distressing one. Even the author departs from loyalty to the cause to complain about "opaque

APL syntax" (p157). But it's an interface which has made me a good living in the past – so who am I to kick against the pricks?

So in me the author is preaching to the converted, albeit forcibly so. I approached the book as a professional, hoping to improve my Windows skills – and I was not disappointed. The author delivers the one thing that APL+Win badly needs to be a serious alternative to VB: copious samples of tested working code, accompanied by practically-minded explanations. He presents the major Windows components (both COM and older DLLs executed via `⎕WCALL`) in the form of APL cover functions: which proves to be the most efficient way to deploy this knowledge. My copy is going to get well-thumbed.

I looked in vain for a CD of working code inside the back cover. The Internet has made all that obsolete: instead you get a URL to the publisher's site which lets you download a free zip-file of miscellaneous data-access bits-and-pieces, plus the thing you want: an APL+Win workspace (`SBWA`) containing all the cover functions described in the text. This way the reader gets the latest revisions of working code: no more out-of-date CDs. However there is still some way to go before `SBWA` looks presentable: the version I downloaded had been saved with a suspended function (*viz.* the introductory fn, itself called `SBWA`).

The book (plus working code) earned its £40 cover price for me in the first hour of use. The author had solved a problem which I was planning to tackle in APL+Win: how to run a spelling check on data input into my app. I consider myself a reasonably competent APL+Win coder, but the author's solution amazed me with its simplicity. I would have imagined it necessary to create a document on disk of the material to be checked – but no: you can hold the target text in an APL variable and put it into the `Selection.text` property of Word.

This example alone made me very satisfied with my acquisition – but it continued in the same vein, teaching me a host of things I didn't know about Windows, confirming things I knew (or suspected) about COM and delivering it all as working code which I could use immediately.

To give a flavour of what's on the menu, there are chapters on working with Excel, Word, Access, ADO, SQL and VB itself – in various COM positions. In a chapter entitled Working with Windows you find out about Date/Time formatting, managing the Registry, creating a shortcut on the Desktop, app configuration with INI and XML, and reliably managing files on disk and their folders. The author avoids an encyclopaedic approach (which would have combinatorially exploded) in favour of telling you just what you need to know about everything (and only) what matters. Considering how much room for debate there is in that, I was struck by how far I found myself in agreement over

what was, and was not, important. Though how to tackle the issues the author correctly identifies gives plenty of scope for debate. For which, alas, there is little room here.

Whilst predominantly about APL+Win, the author mentions other major contemporary APLs: APL2, APLX, APL/W and discusses their special strengths, identifying that of APL+Win as lying in the COM area. Having done serious work with two of the other three APLs, I would wholeheartedly agree, the reason being the APL+ history of developing the "⎕WI-hole" as a schematic to parallel what VB can achieve with GUI coding, to the extent of being able to leverage VB's own documentation and worked examples (up to a point). Since COM is in effect a generalisation and standardisation of the VBX interface for the benefit of non-VB languages, ⎕WI extends easily into COM. How far it succeeds in this approach and where it falls short is something the book usefully discusses – naturally enough, for it is the author's main concern in writing it.

But primarily it is the code samples which give the book its value: functions which work first-go without you having to be creative in an unfamiliar area. If you already know APL, this book turns the tables on VB as the language of choice for building pukka Windows apps, downgrading it to language of last resort.

So where could I have hoped for more?

I looked in vain for coverage of internet access using ⎕NI. A cover-function to download a file from a URL would have been really nice, one to upload even nicer. This is not to minimise the difficulty of coding ⎕NI to engage in a FTP session with a remote site. Perhaps the author thinks it is better to buy one of the available tried-and-tested COM DLLs for the purpose than to call ⎕NI? I do sympathise.

I felt the author was too laid-back about communicating with COM-objects, especially Excel. `0 0⍴⎕WI⍵` is a little bit sanguine. Though of course sample code is clearer when it is not cluttered with diagnostics. Nevertheless, problems with Excel as an ActiveX tend to appear far down the line (generally the Sunday night before code release). The day may come when you'll need to trace that code and look at what's returned by ⎕WI⍵.

A slick code-hook for diagnostics is what is needed. The author should give some space to the time-wasting problems which arise from unwise choice of convention in dealing with returned values: especially those you think you're never going to need. APL2000 in-house tends to use:

```
∇ SINK ship ∇
```

so that `0 0 ρ⎕WIω` can become: `SINK ⎕WIω`. Besides being faster, this has the forensic merit of holding the ship for inspection at some `⎕STOP`-able point before it sinks without trace, and even of being substitutable at need by:

```
    ∇ SINK Z
[1]   →(z≡expected1)/0 ⍝--the most frequent instance
[2]   →(z≡expected2)/0 ⍝--the next most frequent
[3]   →(z≡expected3)/0 ⍝--etc
[4]   ⎕SAVE'DUMP'
    ∇
```

You should then arrange for the runtime initialisation function to fix the simplest form of `SINK`, thus avoiding swabs left in the patient.

Code hygiene, like fire safety, is all about making the improbable impossible. Likelihood must be factored by Regret. Some coding strategies simply farm bugs. Experience alone shows which these are. Gerry Weinberg's "Worst Bug Ever" [2] records how thousands of man-hours were once wasted on an elusive bug that turned out to be a certain memory-location labelled ONE occasionally containing something else, *viz.* 2. Ah, those good old days of Assembler coding! But isn't APL, with no sensible facility for defining a constant, prey to the very same gremlin? The author warns us against using globals as constants. I applaud his recommendation to define constants as niladic functions: I thought I was the only person who did this routinely. For instance:

```
    ∇ z←ok
[1]   z←0
    ∇
```

for use like this:

```
→(ok=↑⎕WI ...)/LABEL_OK
```

Which highlights another problem concerning return codes: there are two conflicting popular conventions – a fact which digs no end of elephant traps in little-used code:

- 0=ok; 1=Error#1, 2=Error#2... etc. (as above)
- (1,xxx)=ok; (0,n)=Error#n

So when you come across `:If ~↑⎕WI` ... what does it really signify? The construct `:If ok=↑⎕WI` ... does at least make clear the programmer's assumptions about what is being returned and what it means.

Some argue that such declarative information is best recorded in a comment because it entails zero execution overhead. But maintenance programmers frequently alter code in disregard of comments, which may then no longer apply. They argue that the interpreter doesn't read the comment, so why should they? Self-documenting code does have its merits, in spite of having 99 different names for 0, not to mention 1.

To those who have never had to spend their weekends tracing bugs to a deadline (and see old friends cropping up time and again) these precautions must look utterly paranoid. But the heart of good programming practice surely is this. Modern programming techniques stress "rapid application development", which is taken to mean ever-quicker ways to cobble-up a prototype, oblivious of the trouble being stored-up for the future. VB is especially bad in this: it is deceptively rapid in the initial stages of development – but it makes you pay for it all later.

VB is oversold for serious development work, particularly where COM-objects are involved. Where, for instance, does this ubiquitous VB statement have any place in professional code?...

```
On Error Resume Next
```

...It's Chernobyl programming! Yet the author shows you how to emulate it in APL+Win, using ⎕ELX and →⎕LC+1. (I must confess, though, to having done so myself.) Tellingly the topic comes up in the chapter on working with ADO.

The time spent debugging a professional product often grossly exceeds that spent on coding the first draft. So why all this emphasis on coding speed nowadays? Are we developers (or more likely our managers) simply being sold snake-oil by the likes of Microsoft? There is a case for sacrificing some of that heady starting velocity, to recover it with interest down the line by squashing bugs known to be huge time-wasters. I would have liked to see more attention given to this matter.

One fault in the book, serious enough in my opinion to fix with an erratum sheet, is corrupt page-numbering in the index. It is a Cinderella of an index, overshadowed by the detailed 3-level table of contents. However every sample APL function is listed here alphabetically: a potentially useful facility. A corrected index can be downloaded from the same URL as the code-samples.

But these are minor grumbles – merely a wishlist for the second edition, which I hope and expect will appear shortly, to clue us APL+Win users up on .NET.

Apart from the function list in the index, the table of contents is better for finding things than the index itself: you can see a link to this at [1]. I haven't covered all the good things in the book and it's well worth taking a look here to get an idea of what I've not had space for. Notably the first two chapters: a scholarly overview of system-building and a useful portfolio of APL techniques to help you build a seaworthy product. Outright professionals may feel they don't need to know all this (but will still prize the book for the sample code). But trainees might well find these chapters immensely valuable: it is hard practical advice which you simply won't find elsewhere. The more product development experience you get, the more you'll come to appreciate how right the author is.

To sum up: this book fills a sizeable hole in the armour of APL+Win when it comes to do battle with VB, namely in the provision of meaty worked examples of ActiveX interfacing and Windows DLL calls generally (at which not even VB is well-documented!). Worth its price for the time it'll save you – even at floor-cleaning rates of pay – and a must-have for any "APL+Winner's" bookshelf.

References

[1] Ajay Askoolum, *System Building with APL+Win*, Wiley, 2006, ISBN 0470030208 http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470030208.html

[2] Gerald M. Weinberg, *The Psychology of Computer Programming*, Coriolis, 1988, ISBN 0442207646

Design Decisions in APLX64

by Richard Nabavi, MicroAPL Ltd (richardnabavi@microapl.co.uk)

When MicroAPL launched its first APL microcomputer in 1980, the CPU was a Zilog Z80, which could address a maximum of 64 Kb. By squeezing the system and APL interpreter software down to an absolute minimum, we were able to offer a workspace size of around 28Kb. Remarkably, users were able to write some quite sophisticated APL systems with this tiny resource, but it was tight!

The big breakthrough came in 1983, when the MicroAPL Spectrum was launched, with a 68000 processor and a maximum of 16MB of RAM (24-bit addressing). For the first time, microcomputer users could enjoy a workspace size as big as, or much bigger than, what a mainframe could offer. Within a couple of years, MicroAPL was selling 68020-based systems with full 32-bit addressing.

At this time, IBM PC users were still largely limited to a total of 640Kb, but this was segmented into 64K chunks which limited the maximum size of arrays. Eventually the PC world caught up, and unsegmented 32-bit systems, addressing up to a theoretical 4Gb of memory (of which 2Gb is the most that Windows applications can address), have been the norm for the last decade or more. Because they use (signed) 32 bit integers internally, most APL interpreters available today have a maximum workspace size of 2Gb, and the maximum number of elements in an array is at most 2,147,483,647 (`¯1+2*31`).

But now a new standard is emerging. First AMD, and subsequently Intel, have extended the original x86 architecture to a full 64 bits. Desktop computers now often contain a 64-bit processor (such as the Intel Core Duo or AMD Athlon), and Intel have now standardised on 64-bits for their entire range of desktop and server processors. Currently, nearly all of these are used in 32-bit mode, but the fact remains: low-cost 64-bit systems are here. And APL is one of the few software products which can make good use of the new power and enhanced memory addressing which is now available.

APLX64 is a fully 64-bit version of APLX, which is designed to take advantage of this new memory addressing capability. It is currently available for Linux and Windows.

Representation and Conversion of Numbers

Integer size

In designing APLX64, one of the first design decisions was: how big should integers be? Our starting point was that we did not want to impose any artificial restrictions on workspace size or array dimensions, so in APLX64 all array dimensions and all internal pointers are 64-bit. This means that, in theory, the maximum workspace size is 8,589,934,592 Gb, and the maximum size of an array is 9,223,372,036,854,775,807 elements (`¯1+2*63`). That should be enough for another decade or two, and takes us well beyond the current generation of 64-bit operating systems, which are typically limited to 128Gb of physical RAM.

In order to index these potentially massive arrays, we decided to implement full 64-bit integers. This was partly to avoid having to use floating-point numbers to index arrays, but also because 64-bit integers are needed in other contexts in 64-bit systems. These other uses include positions in native files, record numbers and IDs in SQL databases, and handles, pointers and other 64-bit values returned from external calls (`⎕NA`). In addition, we wanted the APL user to be able to do full-precision 64-bit integer arithmetic.

Booleans remain as one bit per element, making it possible to handle huge Boolean arrays without excessive memory requirements.

Representation of floating-point numbers

In most 32-bit APLs, including APLX, floating-point numbers are represented in 64-bit IEEE format. This representation has 53 bits of precision, and a range of `¯1.797693135E308` to `+1.797693135E308`.

In APLX64, we decided to keep this same 64-bit format for floating-point numbers. The principal motivation for this decision was that current processors and compilers support 64-bit floats directly, whereas higher-precision representations (such as 80-bit or 128-bit) are not available on many platforms. It does not look as though this will change in the near future. A secondary motivation was to save space on large floating-point arrays.

Conversion between integers and floats

The choice of 64-bit float types presents a potential problem. Up to `2*53`, integers can be represented exactly as 64-bit floats. Above `2*53`, the floats start to lose so much precision that a given float bit-pattern covers a range which includes more than one integer (maybe many thousands of integers). So what happens to the rules for converting integers to floats and vice versa in APL?

In APLX64 the Floor ⌊ and Ceiling ⌈ primitives have been modified so that, given a float number greater than or equal to `2*53`, the number is considered to have overflowed precision, and hence the primitives return the float value unchanged (as a 64-bit float). This is effectively the same behaviour as already happens in 32-bit APLs at `2*31`. The reasoning here is that it is wrong to appear to create a spurious precision by choosing one particular 64-bit integer to represent the floor or ceiling, when the interpreter could equally validly choose many other integers.

For the same reason, any float greater than `2*53` cannot be used in expressions which require an exact integer (for example, to index an array, or as a file pointer). A `DOMAIN ERROR` will be reported.

Integer tolerance

According to the *APL2 Programming Language Reference*:

> A number R is treated as integer if the difference between R and some integer is less than approximately `1E¯13×1⌈|R`.

This definition would have strange consequences for large numbers. It would mean that ALL floating-point numbers greater than `1E13` (approx `2*43`), would be regarded as integers.

To avoid this problem, APLX64 applies the following rules:

- If the resulting integer would fit in a 32 bit integer, we adopt the existing APL2 rule.
- For larger integers, we use the a fixed distance, the same as that which we would use for `2*32`, i.e.
  `1E¯13` × biggest 32-bit integer => 0.0000488

This has the desirable consequence that 10,000,000,000,000.5 is *not* regarded as an integer.

Comparison tolerance

In APLX64 the default value of `⎕CT` has been reduced from `1E¯13` to `3E¯15`. This is a compromise between a value which is small enough to distinguish `x` from `X+1` at high values of `X`, and not giving false negatives for true float comparisons because of calculation and representational inaccuracies. The new default value gives means that, for `X` up to `2*48`, the expression `X=X+1` always returns `0`, irrespective of the internal representation of `X`.

Default display of numbers

In APLX64 the rules for the default display of numbers has been changed. Numbers represented internally as integers are displayed in full precision irrespective of ⎕PP (this is also true in most 32-bit APLs, although it may not be obvious because of the limited allowed range of ⎕PP). In addition, numbers internally represented as floats which are less than 2*53, and which are 'exact' integers, are also displayed in full precision irrespective of ⎕PP. The practical effect of this is that, at the point where the floats lose precision and cannot be converted back to integers, the default display switches into E format. Below that, true 64-bit integers, and floats which are close to or exactly integers, both display in the same way (full precision).

Example

The following sequence illustrates how this all works:

```
      BIGINT←2*48
      BIGINT                    ⍝ 64-bit integer
281474976710656

      ⎕DR BIGINT
2                               ⍝ Data representation 2 means Integer

      BIGFLOAT←1.0×BIGINT       ⍝ Multiply by float
                                ⍝ forces result to float

      BIGFLOAT
281474976710656                 ⍝ Looks the same as BIGINT, though.
                                ⍝ It could be used as a file position,
                                ⍝ array index, etc

      ⎕DR BIGFLOAT
3                               ⍝ .. but Data Representation 3
                                ⍝ i.e. float

      ⌊BIGFLOAT
281474976710656                 ⍝ Floor produces same whole number.
                                ⍝ Good!

      ⎕DR ⌊ BIGFLOAT
2                               ⍝ Internally converted to integer

      BIGINT = BIGINT+1
0

      BIGFLOAT = BIGFLOAT+1
0                               ⍝ Distinct numbers at default ⎕CT

      VERYBIGINT←2*62           ⍝ A rather bigger 64-bit integer
      VERYBIGINT
4611686018427387904
```

```
      ⎕DR VERYBIGINT
2

      VERYBIGFLOAT←1.0×VERYBIGINT ⍝ Force it to 64-bit float form
      ⎕DR VERYBIGFLOAT

3                              ⍝ Data Representation 3, i.e. float

      VERYBIGFLOAT
4.611686018E18                 ⍝ Lost precision:
                               ⍝ displays in E format.
                               ⍝ It can NOT be used as a file
                               ⍝ position, array index, etc

      ⌊VERYBIGFLOAT
4.611686018E18                 ⍝ Floor cannot restore the lost
                                 precision

      ⎕DR ⌊VERYBIGFLOAT
3                              ⍝ so it returns the same float
                                 number

      VERYBIGINT+1
4611686018427387905            ⍝ Great! We can add 1 to a
                                 64-bit integer!

      VERYBIGINT = VERYBIGINT+1
0                                    ⍝ Integer comparison:
                                     ⍝ They are distinct

      VERYBIGFLOAT=VERYBIGFLOAT+1
1                                     ⍝ Float comparison:
                                      ⍝ Same (within ⎕CT)

      VERYBIGFLOAT+1
4.611686018E18                 ⍝ Actually, the addition does
                                 nothing.

                               ⍝ We have only 53 bits of precision,
                               ⍝ so the extra 1 is lost off the end
                               ⍝ for a number of magnitude 2*62
```

Summary of integer-float conversion issues

The practical effect of these design choices is that, for whole numbers below 2*48, the APL programmer does not need to know or care whether the number is internally represented as a float or as a 64-bit integer; it will behave and display in the same way, and comparisons will always give the expected result. Any conversion between the two internal forms loses no precision, and hence is reversible (e.g. using Floor or Ceiling). Either representation can be used to index an array, or represent a position in a huge native file.

For numbers between `2*48` and `2*52`, the same is true, except that the APL programmer might need to reduce `⎕CT` to avoid comparison problems, or alternatively use Floor or Ceiling to force the numbers to integer before doing a compare.

Above `2*52`, if the APL programmer needs exact integers (for example, for doing high-precision arithmetic, or if the integers are 64-bit database record numbers), APLX64 can correctly handle this requirement. However, in this case the APL programmer needs to be careful to ensure that the integers do not accidentally get converted to float (for example, by mixing record numbers and float values in a single N×2 matrix, or by doing arithmetic operations which are intrinsically non-integer, such as divide). Fortunately, if this does happen, it should be obvious, because the display will flip into E format at the point where precision has been lost, and operations which require an integer will give `DOMAIN ERROR` rather than giving the wrong answer.

The Client-Server Architecture

Internally, the APLX64 product comprises two separate programs. The 64-bit APL interpreter itself runs as a 64-bit application (called `aplx64_server` on Linux, or `aplx64_server.exe` on Windows). This is the *Server*. The front-end, which is the program you use to edit, run, and debug APL workspaces, and which implements all of the user-interface elements and `⎕WI`, is a 32-bit program (called `APLX.exe` on Windows). This is the *Client*. The Client and the Server can run on the same physical machine, or on separate machines connected by a TCP/IP network. Typically, the Client runs on a desktop Windows system, and the Server runs on a Windows or Linux server system, although other combinations are possible.

A given Server can support any number of Clients (each of which may be running more than one APL session on the Server), subject to having sufficient memory and CPU resources and the license agreement in force. Also, a given Client can connect to multiple Servers, so you can run several 64-bit sessions simultaneously on different servers.

As well as the 64-bit interpreter, APLX64 also includes a 32-bit version of the interpreter, which is part of the Client program. This allows you to develop and test 32-bit APLX applications as well as full 64-bit applications. A given Client can run both 32-bit and 64-bit APL sessions simultaneously.

Communication between the Client and Server

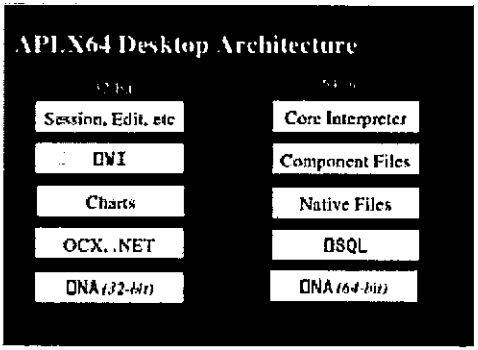
The Client and the Server communicate with each other using the TCP/IP network protocol (this is true even if they are physically on the same machine). The official IANA port number allocated to APLX is 1134.

Security and Firewalls

The network communication used by the APLX client-server architecture is not encrypted, and could in theory be snooped on, or used to run malicious APL code over the network. For this reason, we strongly recommend that the Server should be protected by a firewall so that it is not exposed to attacks from untrusted sites. The firewall should normally be set up to disallow all traffic on port 1134 except between the Server and authorised Client machines on an internal network.

It is possible to run the Client remotely from the Server (for example, for an employee to run the Client on a machine at his or her home, accessing the Server in the corporate data centre over the internet). However, the only safe way to do this is to use a secure VPN (Virtual Private Network), which has been correctly set up to fully protect traffic between the two machines.

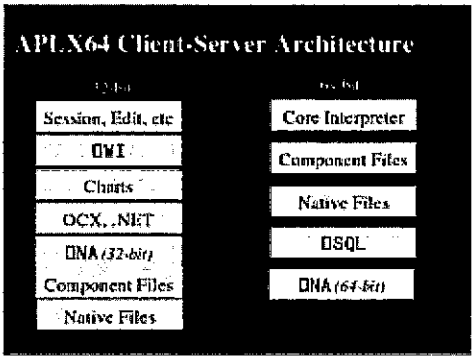
Running APLX64 on a 64-bit Windows Desktop system



In APLX64 Desktop Edition, the Client and the Server run on the same machine, usually under Windows XP64 or Vista. When you start the Client program, normally the Server program is started automatically, so the fact that there are two separate programs running is transparent to the user. When the last APL session ends and the Client program exits, the Server program will also terminate automatically.

Because most of the program is 32-bit, it installs by default in the `Program Files (x86)` directory. This is true even of the 64-bit interpreter itself. Also the Registry entries are 32-bit, sitting in the `WOW6432Node` area of the Registry.

Running the Client and Server on separate machines



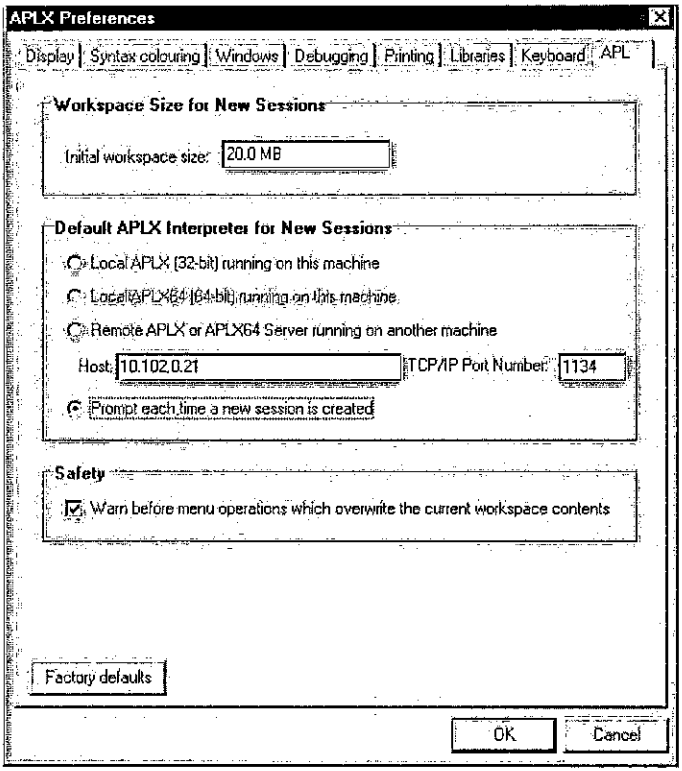
Alternatively, the Client and the Server can run on separate machines. The Client usually runs under Windows, whereas the Server can run under 64-bit versions of Windows or Linux. When you start the Client program, normally it will try to connect to the APLX Server program running on the server machine specified in your Preferences (see below).

The Server program must already be running when the Client tries to connect to it. The Server starts as a small 'listener' program which waits for a connection. When it receives a connection request from an APLX Client, it starts another process which is the actual APL interpreter associated with that connection.

32-bit and 64-bit APL Tasks

Customizing the creation of new APL tasks created from the menus

Using the APL tab of the Tools->Preferences dialog, you can alter the way in which new APL sessions, including the initial session at start-up, are started:



The choices are:

- Use the 32-bit APL built-in to the Client program.
- Use the 64-bit APL Server running on the same machine as the client. If the Server is not already running, it will be started automatically. On a 32-bit Client system, or if you do not have the 64-bit interpreter installed on the same machine as the Client, this option is not available and will be greyed out.
- Connect to a remote APLX interpreter over the network, in which case you need to specify the host and port in the normal way (the default port is 1134). You can specify the host as either the IP address (for example, 10.102.0.21), or as the network name of the server machine (for example, `'server23@bigcorp.com'`), or as `'localhost'`, which always means the same machine as the client. Note that the APLX Server program must already be running and accepting connections on the specified system – it will not be started automatically.

You can also say you want to be prompted each time you start a new session. This applies even to the initial session which opens when the APLX Client starts up.

Creating Tasks under Program Control

You can also create new tasks under program control, using the ⎕WI APL object in the same way as in standard 32-bit APLX. The default is that the new APL session starts in the same execution environment as its parent, so if you create a new APLX task from a 64-bit task, the child will also be a 64-bit APL on the same server.

However, you can tune this by setting the `host` (and optionally `port`) property of the APL object before calling the `Open` method. If the host property is an empty string, the task will be created a 32-bit APL on the Client system. If it is set to the string `'localhost'`, it will be a 64-bit APL on the client machine (assuming the Client is running on a 64-bit system), and the Server program will be started automatically if necessary. If it is anything else, the front-end will attempt to create the new APL task by connecting to the specified remote machine (which must already be running the APLX Server program).

This example will create a new 32-bit APL session:

```
'Session32' ⎕WI 'New' 'APL' ('host' '')
'Session32' ⎕WI 'Open'
```

This example will create a new 64-bit APL session running on the local machine (assuming it is a 64-bit machine with APLX64 installed):

```
'Session64' ⎕WI 'New' 'APL' ('host' 'localhost')
'Session64' ⎕WI 'Open'
```

This example will create a new 64-bit APL session running on a remote machine:

```
'SessionRemote' ⎕WI 'New' 'APL' ('host'
'server23@bigcorp.com')
'SessionRemote' ⎕WI 'Open'
```

If you do create child APL tasks in this way using ⎕WI, they can share APL data by using the `data` property of the `Child` object (in the parent task) and the `data` property of the `System` object in the child task, or by using property names beginning with delta. However, these are held as 32-bit objects, and (like all ⎕WI properties) will be converted to 32-bit variables before they are sent from the 64-bit APL task to the front-end.

Operations on Client and Server

Where the Client and the Server run on different machines, you can specify where you want certain operations to take place, by the file name or command string with either a ↑, meaning the Client, or a ↓ meaning the Server. For example, you may want to)SAVE a workspace either on the Client machine, or on the Server machine.

The choice of where an operation occurs applies to:

-)LOAD)SAVE etc where a library number is used. In this case, the corresponding line of the ⎕MOUNT table of library numbers is used to determine the library path, and the first character of this path can be either ↑ or ↓ to indicate which machine is being referenced.
-)LOAD)SAVE etc where you specify the full file name
- Native files
- Component files
- ⎕NA
- ⎕HOST and)HOST

This all makes no difference if you are running the Client and the Server programs on the same machine, except for ⎕NA, which now allows you to call either a 32-bit or 64-bit DLL (see below).

File Accesses

Both component and native files can be accessed either on the Client, or on the Server. For example you might have saved some data from Excel on your desktop PC. The 64-bit APLX application, running on a remote server, can open this as a native file over the network completely transparently using ⎕NTIE. It can then save the results of some large calculation based on this input as another native file, this time on the server. All that is necessary to make this work is to prefix the name of the file (when you open it or create it) with a ↑ or ↓.

If you do not specify, the operation will take place on the Client. This may seem a bit surprising, but it means that file selector dialogs still work and give the expected result.

The ⎕WI sub-system

The ⎕WI sub-system is part of the Client program, so it always runs as 32-bit code. When you make a call from a 64-bit interpreter, the request is converted to 32-bit

form and sent over to the Client program for execution. Any ⎕WI windows and dialogs, therefore, appear on the Client system – which is what you want! In addition, any references in ⎕WI objects to files and directories are from the viewpoint of the Client system.

⎕HOST

⎕HOST can be used to execute an operating-system command or run another program on either the Client or the Server. In this example, the Client is running under Windows, and the Server under Linux x86_64:

```
      ⎕HOST 'tcmd /c vol c:' ⍝ Execute on Windows Client machine
Volume in drive C has no label.
Volume Serial Number is 07D0-0B11

      ⎕HOST '↓uname -nsp'    ⍝ Execute on Linux Server machine
Linux Server23 x86_64
```

The ⎕NA system function

In APLX64, ⎕NA has been extended so that it allows you to call either a 32-bit DLL (from the Client program), or a 64-bit DLL (from the Server program). The implementation is as follows:

For clarity, we will assume that the 64-bit Server is running on one machine, and the 32-bit Client is running on a different machine. (In fact, they might be different operating systems, e.g. a Linux 64-bit server and a Windows 32-bit client).

When you use ⎕NA, you might want to call a function on either end. For example, it would make sense to make a 32-bit call to Windows to discover something about the screen or registry on the Client. Equally, you might want to invoke an OS service or library on the Server, for example to call a Linux file-encryption API.

APLX64 use the same conventions as for file names to allow you to specify which you want. If you prefix the ⎕NA specification (i.e. the right argument) with a ↑, the call takes place on the Client. If you prefix it with a ↓, it takes place on the Server. The default (if you do not specify either) is that it takes place on the Client. (This is for compatibility with existing 32-bit APLX Windows applications).

If you make the call on the Server side, the APL task directly calls the requested library. There is no special handling. Everything is 64-bit, and there are no extra tasks involved.

If you make the call on the Client side, the APL task bundles up the request and sends it over the network (which might be just an internal pseudo-network if the

two are on the same machine). It then blocks and waits for a response. On the Client side, a 32-bit task picks up the request and makes the (32-bit) call. It returns the results over the network to the 64-bit Server, which wakes up and continues APL execution.

To support 64-bit calls, new data types for 64-bit integers have been added (I8 and U8 for signed and unsigned 64-bit integers).

As an example, suppose you are running the APLX64 Server on a twin-processor Xeon Linux 64-bit system, and running the 32-bit APLX client on a 32-bit Windows PC. In these circumstances, you can define two external functions, one which makes a call to get the Windows system directory (a 32-bit Windows call) on the Client machine, and another which makes a 64-bit Linux call to get the current working directory on the Server:

```
      'GetSystemDirectory' ⎕NA '⍞U
kernel32|GetSystemDirectoryA >C[256] U'
      GetSystemDirectory '' 255
17 C:\WINNT\system32

      ⎕NA '⍞/lib64/libc.so.6|getcwd >C[512] U8'
      getcwd '' 512
/home/david/aplx64
```

If you are running the Client and the Server on the same physical machine, the above all remains true. Under Windows XP64 or Vista 64-bit, 32-bit applications (such as the Client program) run within a virtual 32-bit environment, and access different versions of the operating-system libraries (confusingly, the 32-bit versions reside in `C:\WINDOWS\SysWOW64`, and the 64-bit versions in `C:\WINDOWS\system32`). You can therefore make either the 32-bit or 64-bit version of a system call or library access from within APLX64. This capability is very unusual – perhaps unique, since normally 32-bit programs can make only 32-bit calls, and 64-bit programs only 64-bit calls.

Conclusion

APL makes an excellent partner to the new 64-bit hardware and operating systems, taking full advantage of the new power and memory addressing. This, combined with the Client-Server architecture, makes possible new kinds of distributed APL applications. It will be fun designing and implementing new types of APL applications in this new environment!

LEARN

Image Files with Dyalog

by Klaus Klug Christiansen (kkc@apl.it)

In building a web gallery of (some of) my photos I quickly realised that I need as much of the process as possible to be automated. Making an APL program to do that shouldn't be too hard, but at least in Dyalog loading and saving JPEG images would not be straightforward. From previous experience I knew of a comprehensive open source image library, which abundantly would fulfil the requirements of my present task.

Developer's Image Library

The open source image library is available from SourceForge As can be seen from the link the project was originally named OpenIL, but it seems that in order to effectively avoid violation of trademarks etc. they decided for changing the name to Developer's Image Library a.k.a. DevIL.

There are two manuals to document the use of DevIL: *The Reference Guide*, and the *Developer's Manual*.

The former gives a list of calls in the DevIL API. Unfortunately this list is not comprehensive and gives details on neither arguments nor results. Besides it has not been updated for quite some time. The latter gives many more of the needed details but still it's not comprehensive.

Details missing from both documents can be read from the header files: `lib\include\IL\il.h` holds all the constant definitions and function headers.

The whole package is easily accessible from modern APLs as it is nicely wrapped in three DLLs, which are easily called using ⎕NA: `DevIL.dll`, `ILU.dll`, and `ILUT.dll`.

- `DevIL.dll` contains all top-level functions, which will suffice if you don't intend to do much but loading, converting, and saving.
- `ILU.dll` contains the more sophisticated functions to use when doing image manipulation like e.g. resizing.
- `ILUT.dll` contains all remaining low-level functions, mainly for use in the other two DLLs , and so far I have not come to need any of these.

APL

I have prepared a Dyalog APL Version 9 workspace that contains functions which should make it easy to take advantage of the facilities of DevIL. At least it takes away the pain of figuring out the arguments for (some of) the calls to the DLLs.

There are at least 2 ways of approaching the code in the workspace: keeping image data in the APL workspace and simply using DevIL for reading and writing image files, or as well employing DevIL as a tool for manipulating the read images. In either case all code is contained in the `#.DevIL` namespace in the `devil.dws` workspace.

Approach 1: Simple read and write

Two functions allow for simple reading and writing of image files: `#.DevIL.FileRead` and `#.DevIL.FileWrite`

`#.DevIL.FileRead filename` reads an image file and returns a vector of three elements: Error message, `bits`, and `cmap`.

- The error message is an empty vector if the operation completed without generating an error.
- `bits` holds the image data. If the image is RGB then the bits are encoded in the usual Dyalog style: `PIXEL←256⊥RED GREEN BLUE`
- `cmap` holds the image palette. Like `bits`, `cmap` adheres to the Dyalog convention and represents a palette as a matrix with 3 columns. If there is no palette then `cmap` has no rows.

In the function are listed the image file formats that DevIL is capable of reading. It automatically detects the format of the file when reading it.

Images with an alpha-channel can be loaded though the alpha-channel is discarded and so not returned. Alpha-channels are however fully supported by DevIL so it's simply a question of editing the `#.DevIL.GetBits` function to get to it.

`(bits cmap) #.DevIL.FileWrite filename` writes an image file, returning an error message, which is an empty vector if no error occurred.

The left argument is a vector with two elements:

- `bits` holds the image data. If the image is RGB then the bits are encoded in the usual Dyalog style: `PIXEL←256⊥RED GREEN BLUE`
- `cmap` holds the image palette. Like `bits`, `cmap` adheres to the Dyalog convention and represents a palette as a matrix with 3 cols. If there is no palette then `cmap` has no rows.

DevIL automatically detects the desired format of the image file from the extension of the file name. The image file formats that DevIL can generate are listed in the function.

Please note that neither of these two functions accepts any attributes of the image file, like e.g. the compression ratio of a JPEG image file. All attributes will take default values, which in the case of the JPEG compression rate is maximum image quality.

Approach 2: Deploying DevIL

A second way of approaching the use of DevIL could be to keep the image in the library and not load it into the APL workspace. To accomplish that first an instance of the DLL is required:

```
img←#.DevIL.New
```

The variable `img` now holds a reference to a namespace, where all cover functions can be found. This means that all operations can now be performed from within the namespace. E.g. loading an image into this instance of the image library can be done like this:

```
img.Load 'photo.jpg'
```

As most of the other functions the `load` function returns an error message, which is empty if no error occurred. Once an image is loaded it is possible to query its properties like this:

```
      bits←img.GetBits
      cmap←img.GetCMap
      cbits←img.GetCBits
      img.GetSize
600 800
```

The function `GetBits` returns the unprocessed image data and `GetCMap` the palette. `GetCBits` always returns RGB image data, i.e. it will convert any indexed (using a palette) image to RGB, thus rendering the palette superfluous.

In the same way it is possible to invoke the functions in the image library for which cover functions have been implemented. Say, you want to resize an image:

```
      img.Resize 200 320
```

As can be seen the sequence of the coordinates follow the usual Dyalog convension, giving vertical before horizontal. This goes for the cover functions only, though. Please note as well that the scaling on either axis can be chosen freely.

A left argument to the `Resize` function can be specified, which will indicate to the image library what resizing method to use. No translation to/from the enums employed by the library has been provided but all relevant ones are listed in the function.

Please note that only the three simplest resizing methods can be used on indexed images, i.e. images with a palette. The other six require the image not to be indexed. If needed, an indexed image can be easily converted:

```
      img.ilConvertImage 6407 5120
1
```

Like this the image is converted to RGB with 1 byte per channel per pixel. The untranslated error code 1 indicates success.

The two numbers in the right argument indicates the desired format and type of the resulting image. The format indicates the image encoding (indexed, RGB etc.) and the type the channel pixel size (byte, word etc.) No translation to/from the enums used by the library has been provided but all relevant ones are listed in the `#.DevIL.New` function.

In order to save space (to allow for more photos in the gallery), maybe the quality of the resulting JPEG image file should be reduced. No dedicated cover function has been implemented, but the desired quality level can easily be set with the variable `jpegQuality`.

```
img.jpegQuality←80
```

Now that the image has been resized and desired quality set it might be time to save the result.

```
img.Save 'newimage.jpg'
```

A left argument can be supplied, which will indicate the desired file format. The possibilities and corresponding enums are listed in the function. Normally this is not required however, as DevIL automatically detects the file format from the file name extension.

In the case of the photo gallery a thumbnail might come in handy. This is a simple operation, just make the image even smaller and save it again:

```
img.Resize 50 66
img.Save 'tn\newimage.jpg'
```

Beyond

As already hinted this workspace only covers a fraction of what DevIL has to offer. More cover functions can easily be built for the facilities that are not already "covered". It should not present great difficulty to find the desired API function call in the mentioned `lib\include\IL\il.h file`, and once the structure of its arguments and result has been established a suitable ⎕NA call can be built. The ones provided in the `#.DevIL.New` function should provide all information required to perform this task.

Interesting aspects to explore could be DevIL's capabilities of e.g. keeping several images in the same instance or the interface to OpenGL.

Heading on

Though the framework presented here is limited in scope there are a number of uncertainties and missing answers, which would require a more thorough examination of the C source code. One such is e.g. if generation of GIF image files is indeed supported or not. The DLL itself refuses to save GIF files whereas the project front page at SourceForge seems to imply that the required code is actually present. Judging from messages in one of the forums it's the expiry of the Unisys patent that has not been fully adopted.

Another very interesting aspect about DevIL is its WinCE/PocketPC distribution, which would allow for its use even with Dyalog on a PocketPC. At least there used to be one such distribution but I have not investigated if it is even generatable anymore.

Building C# COM DLLs for APL

by Ajay Askoolum (ajayaskoolum@ntlworld.com)

Introduction

In this article, I present a visual guide to building and deploying a C# COM (Component Object Model) Interrop DLL (Dynamic Link Library) for APL using Visual C# 2005 Express Edition. This is not a tutorial on either C# or on Visual C# 2005 Express Edition, but simply an illustration of the process for building COM DLLs in C#.

For a limited period and subject to conditions, Visual C# 2005 Express Edition is available as a free-download, together with installation instructions [1].

The continuing evolution of the .Net platform has spawned a lot of documentation on the internet which appears to be misleading, at least from an APL point of view; a contributing factor in this confusion is that the documentation applies to several versions of the .Net framework and Visual Studio.net. This article redresses the situation by providing a template for building COM DLLs for APL. The actual code in the DLL is of a trivial nature and intended solely to illustrate how to build it.

A COM DLL is a server to a client; in the present context, the client is APL. The COM server exposes methods, properties, and events that the client can use irrespective of the native language of the DLL. In order to deploy such a DLL, the target computer must have the .Net Framework 2.0 installed; this is available as a free download [2]. Visual Studio.Net 2005 uses the same framework and although the interface presented by the Express and full versions are different, it is possible to use Visual Studio 2005 instead of the Express edition. This article is based on the Express edition.

File | New Project

Start the Express edition and click `File|New Project`. Figure 1 shows the dialogue that appears.

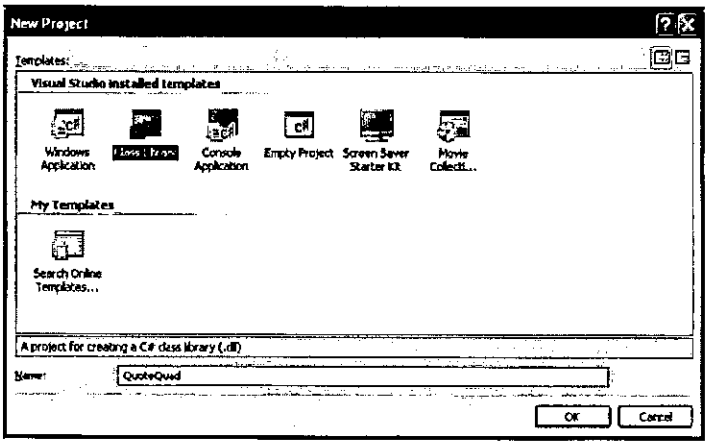


Figure 1. Selecting the type of project

Select the Class Library template and specify the name as `QuoteQuad`, as shown above. This ensures that `QuoteQuad` is the name of the DLL. By default, the class library is not configured for COM usage.

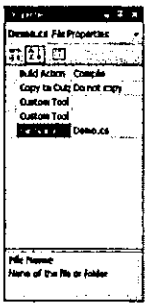


Figure 2. Naming the class

Modifying the DLL properties

In order to modify the properties of the DLL to enable it for COM, click `View|Properties Window` and then select `Class1.cs`; the objective is to change the class name. Figure 2 shows the class name *after* `class1.cs` has been renamed `Demo.cs`.

Clients will instantiate the DLL by using the *Library.Classname* convention; in this context, the reference is `QuoteQuad.Demo`.

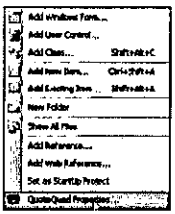


Figure 3. Properties

Setting COM properties

Next, click `Project|Quote Quad Properties`; this step is shown in Figure 3.

Next, Figure 4 shows the screen that becomes visible.

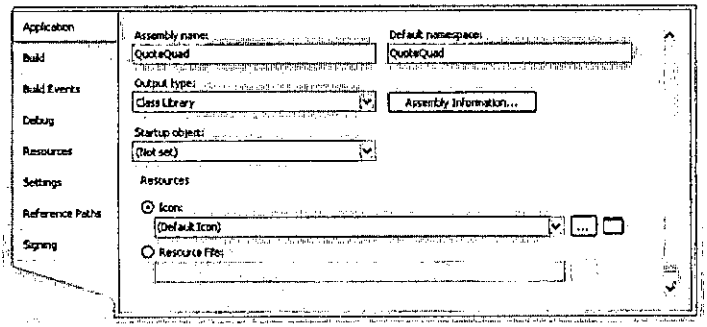


Figure 4. Specifying COM properties

Click `Assembly Information` in order to display the dialogue shown in Figure 5.

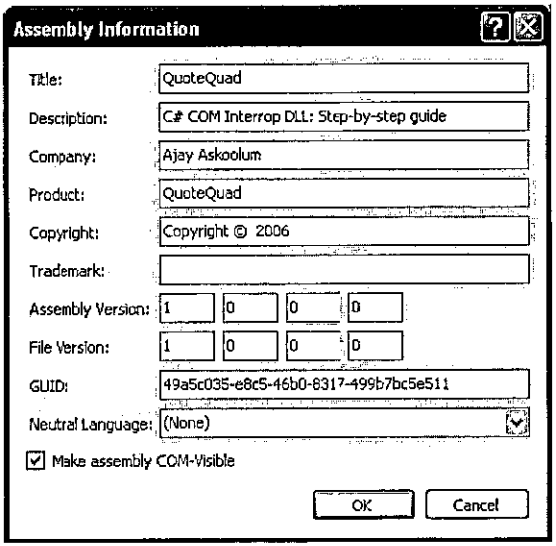


Figure 5. COM visibility

Tick the `Make assembly COM-Visible` the option.

Next, click `Build` in the left pane and specify the output path and tick `Register for COM interop` as shown in Figure 6.

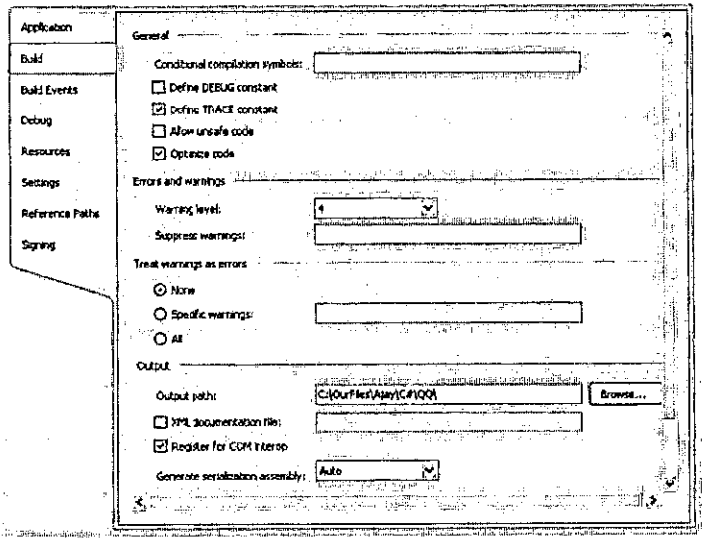


Figure 6. COM Registration

Save the project

Click `File|Save All` in order to save the project. The dialogue shown in Figure 7 appears:

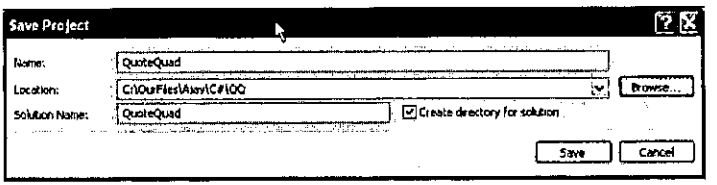


Figure 7. Location of DLL

For `Location` specify the fully qualified name of the location and then click `Save` to save all the files for the project. Note the option to `Create directory for solution`.

Compiling the DLL

The DLL is ready for compilation. Click `Build|Build Solution` or press F6. However, the DLL does not have any methods, properties, or events to expose; therefore, there is little point in creating the DLL at this stage.

Contents of the assembly

The first file, `Assembly.cs`, is ready and contains the following information:

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the
information
// associated with an assembly.
[assembly: AssemblyTitle("QuoteQuad")]
[assembly: AssemblyDescription("C# COM Interrop DLL: Step-by-step guide")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Ajay Askoolum")]
[assembly: AssemblyProduct("QuoteQuad")]
[assembly: AssemblyCopyright("Copyright © 2006")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
// Setting ComVisible to false makes the types in this assembly not visible
// to COM components. If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(true)]
// The following GUID is for the ID of the typelib if this project is
exposed to COM
[assembly: Guid("49a5c035-e8c5-46b0-8317-499b7bc5e511")]
// Version information for an assembly consists of the following four
values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can default the Revision and Build
Numbers
// by using the '*' as shown below:
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Note that the code in this file is auto-generated from the configurations set using the graphical user interface dialogues.

Adding methods, properties, and events

As mentioned, this does not provide a tutorial on C# or on C# Express; the objective is to provide a template for building C# COM DLLs. The functionality of the DLL is manually coded in the `Demo.cs` file. Its content is as follows:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.InteropServices;
namespace QuoteQuad
{
  public delegate void EventDel();
  [InterfaceTypeAttribute(ComInterfaceType.InterfaceIsIDispatch)]
  public interface UserEvents
  {
    [DispId(5)]
    void MyEvent();
  }
  [InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
  public interface _Demo
  {
    [DispId(1)]
    int MyMethod(int numarg);
    [DispId(2)]
    string MMethod();
    [DispId(3)]
    int MyProperty { get; set;}
    [DispId(4)]
    void eventm();
  }
  [ClassInterface(ClassInterfaceType.None)]
  [ProgId("QuoteQuad.Demo")]
  [ComSourceInterfaces(typeof(UserEvents))]

  public class Demo : _Demo
  {
    public event EventDel MyEvent;
    public Demo()
    {
    }
    private int myVar;
    public int MyProperty
    {
      get { return myVar; }
      set
      {
        MyEvent();
        myVar = value;
      }
    }
    public int MyMethod(int numarg)
    {
      return numarg + 10;
```

```
        }
        public string MMethod()
        {
          return "Ajay Askoolum";
        }
        public void eventm()
        {
          MyEvent();
        }
      }
    }
```

The critical parts of the code are:

- The `DispId` attributes must be unique integers.
- Events are declared in the `public interface UserEvents` block and methods and properties are declared in the `public interface _Demo` block.

At this point, a C# language manual is necessary in order to be able to build reasonable functionality in the DLL.

Building the DLL

In order to make the assembly, click `Build|Build Solution` or press F6. This process creates several files, including the DLL at the following location

```
C:\OurFiles\Ajay\C#\QQ\QuoteQuad\QuoteQuad\obj\Release
```

The location will vary depending on the specification for `Location`; see Figure 7.

The only file that is required for COM operation is:

```
C:\OurFiles\Ajay\C#\QQ\QuoteQuad\QuoteQuad\obj\Release\QuoteQuad.dll
```

On the *development* computer, no further action is necessary before using the DLL as the build process has already registered the DLL silently, and can be seen from APL+Win:

```
      '#' ⎕wi 'XInfo' 'QUOTE'
QuoteQuad.Demo ActiveObject QuoteQuad.Demo
```

Deploying the DLL

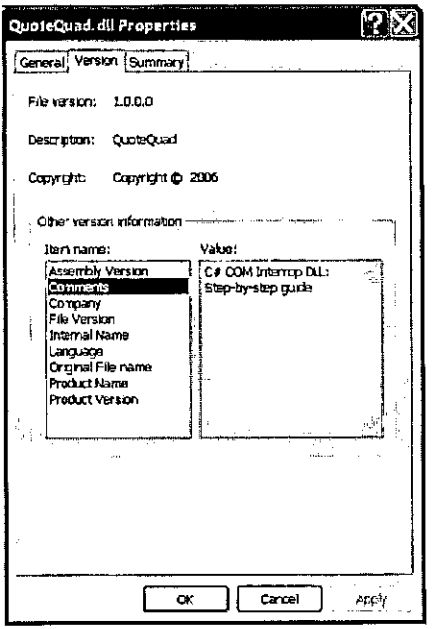


Figure 8. QUOTEQUAD.DLL properties

The DLL's properties can be examined by collating it within the filing system and clicking the right mouse button: see Figure 8.

On the target computer, execute the following steps:

1. Ensure that there are no existing sessions of a client that will use the DLL.
2. Ensure that it has the .Net Framework 2.0 installed already.
3. Copy the DLL to a target computer at any suitable location.

In principle, it is inadvisable to copy the DLL to a system folder such as `System32`; a good location is the folder housing the application that will deploy the DLL. For the purpose to hand, the DLL is copied to `C:\MY APL APPLICATION`.

In order to make the DLL available, click `Start|Run` to display the dialogue shown in Figure 9.

In the `Open` box, type:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\RegAsm.exe …
 … /codebase "C:\MY APL APPLICATION\QuoteQuad.dll"
```

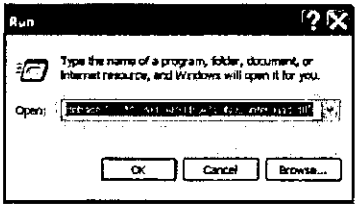


Figure 9. Registering the DLL

Click OK.

The code in the box shows the fully qualified name of REGASM.EXE followed by the switch /codebase and then the fully qualified name of the DLL. The file locations vary on your computer.

Testing the DLL

At this point, the DLL is available on the target computer and is readily deployed by APL.

Using APL+Win, an instance of the DLL is created as shown in Figure 10.

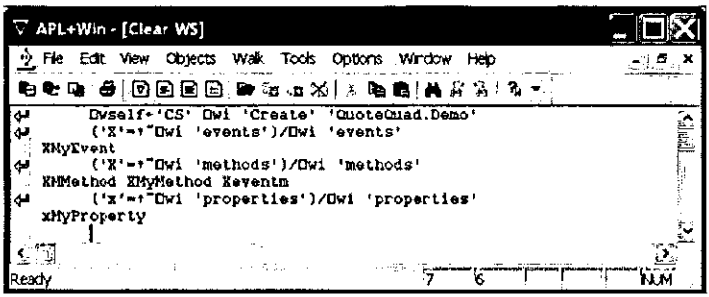


Figure 10. APL+Win

Using APLX, an instance of the DLL is created as shown in Figure 11.

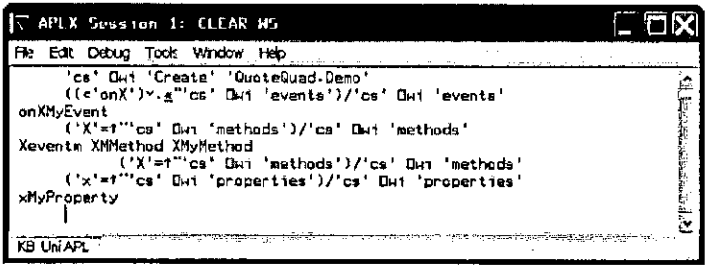


Figure 11. APLX

Using Dyalog, an instance of the DLL is created as shown in Figure 12.

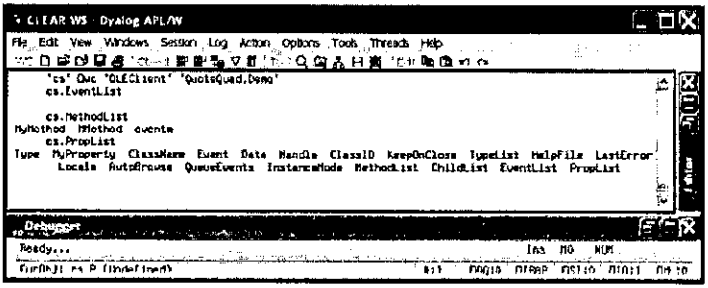


Figure 12. Dyalog

APL+Win and APLX are consistent in the enumeration of the events, methods, and properties of the DLL. However, it is unclear why Dyalog is unable to see the event.

Demonstration

I shall use APL+Win to demonstrate the usage of the DLL. First the properties:

```
      ⎕wi '?MyProperty'
xMyProperty property:
  Value@Long ← ⎕WI 'xMyProperty'
  ⎕WI 'xMyProperty' Value@Long
```

The property `MyProperty` is a read/write property that is numeric.

```
      ⎕wi 'xMyProperty'           ⍝ Read
0
      ⎕wi 'xMyProperty' 100.75    ⍝ Write
      ⎕wi 'xMyProperty'           ⍝ Read
101
```

There appears to be an inconsistency! On closer examination of the code, it is apparent that the DLL coerces the value of the property to integer; hence, the result is `101` instead of `100.75`.

```
public int MyProperty
{
  get { return myVar; }
  set
  {
    MyEvent();
    myVar = value;
  }
}
```

This property will also raise the event `MyEvent` upon the value of the property being changed. However, the event was raised in the previous assignment but I had not specified an event handler. An event handler is specified in the usual manner, thus:

```
      ⎕wi 'onXMyEvent' '0.01×+\10/1' ⍝ Specify event
handler
      ⎕wi 'xMyProperty' 190254 ⍝ Expect handler to run
0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1
      ⎕wi 'xMyProperty' ⍝ Verify
190254
```

The event handler did fire and ran the APL handler; any APL+Win function may be called by the handler.

The method `eventm` also raises the event `MyEvent`.

```
      ⎕wi 'Xeventm' ⍝ Call method & expect MyEvent to
fire
0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1
```

The method `MMethod` simply returns a static result:

```
      ⎕wi 'XMMethod' ⍝ This method returns a string
Ajay Askoolum
```

Finally, the method `MyMethod` takes a single argument and returns it incremented by `10`.

```
      ⎕wi '?MyMethod'
XMyMethod method:
  Result@Long ← ⎕WI 'XMyMethod' numarg@Long
  ⎕wi 'XMyMethod' 190254
190264
```

Although the methods, properties, and events in this demonstration C# COM DLL are simple, it is clear that it is now possible to write COM servers in the Microsoft flagship language, C#, for deployment with APL. Moreover, the template developed in this article for building and deploying such DLLs is minimalist, as would become clear from a study of the subject. For instance, I have completely circumvented the issues relating to strong names, the Global Assembly Cache and the debate related to managed code.

The Next Step

The source code for the DLL is included in qq.zip, as is `quotequad.dll`. The next step is for you to replicate the whole demonstration for yourself, using the source code, and then to write more purposeful methods and properties. In order to accomplish this, you will need a good reference on C# and an understanding of the conventions relating to COM objects. Also, use the `quotequad.dll` to test the deployment technique discussed above.

References

[1] *http://msdn.microsoft.com/vstudio/express/visualcsharp/download/default.aspx*

[2] *http://www.microsoft.com* Search for `dot net framework 2.0 redistributable`

[3] C. Nagel, B. Evjen, J. Glynn, M. Skinner, K. Watson & A. Jones, *Professional C# 2005*, WROX, ISBN-13 978-0-7645-7534-1

[4] Ajay Askoolum, *System Building with APL+Win*, John Wiley, ISBN 0-470-03020-8

Zipping and Unzipping Files in APL+Win

A Basic guide to using Windows XP file compression facilities from APL+Win

by Ajay Askoolum (ajayaskoolum@ntlworld.com)

Verify File

Verify whether a file exists using PathFileExists; this is an API call. If this declaration is not found in your INI file, add it, thus:

```
⎕wcall 'W_Ini' '[Call]PathFileExists=L(*C pszPath) ALIAS
PathFileExistsA LIB shlwapi.dll'
```

The test to verify declaration is:

```
      ρ⎕wcall 'W_Ini' '[Call]PathFileExists' A 0 =
Missing
```

Create Empty Zip File

An empty Zip file can be created using the following function:

```
    ∇ Z←CreateZipFile R
[1]   A Ajay Askoolum
[2]   :if 0=⎕wcall 'PathFileExists' R
[3]      R ⎕xncreate Z←¯1+⌊/0,⎕nnums,⎕xnnums
[4]      ('PK¨←',(18ρ⎕tcnul),⎕tcnl,⎕tclf) ⎕nappend Z
[5]      ⎕nuntie Z
[6]      Z←1
[7]   :else
[8]      Z←0
[9]   :endif
    ∇
```

This function returns `1` if successful and `0` otherwise; the right-hand argument is a fully qualified filename.

```
      CreateZipFile 'c:\ajayAskoolum.zip'
1
```

Signature of Zip File

The first 22 bytes of the file varies, depending on the application that created the Zip file. A file created thus can be used by WinZip. You can write a function to

verify whether a particular file is a ZIP file by matching the first 4 bytes – see line [4].

Add files to an Existing Zip File

The following function will add a list of files to an existing Zip file:

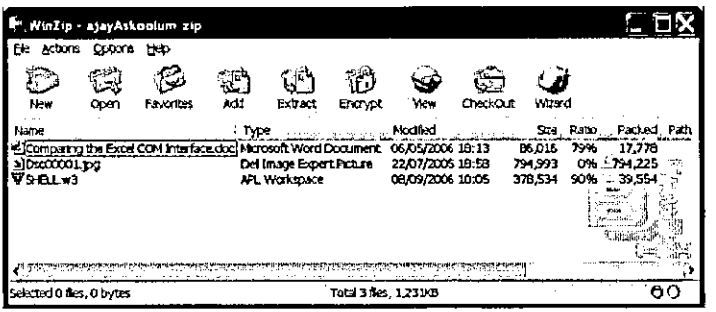
```
    ∇ Z←L AddToZip R;⎕wself
[1]   ⍝ Ajay Askoolum
[2]   ⍝ L = Fully qualified name of an existing and valid Zip file
[3]   ⍝ R is a list of files to add or refresh inside ZIP
[4]   :if 0=⍴'ShApp' ⎕wi 'self'
[5]      ⎕wself←'ShApp' ⎕wi 'Create' 'Shell.Application'
[6]   :else
[7]      ⎕wself←'ShApp'
[8]   :endif
[9]   :while 0≠⍴R←,R
[10]     ⎕wi 'xNamespace().CopyHere' L (⎕io⊃R)
[11]     R←1↓R
[12]  :endwhile
    ∇
```

Note that the function does not delete the instance of the Shell Application object; this is deliberate and intended to save time creating the object when the function is used repeatedly.

```
      ⍝ Add a single file
'c:\ajayaskoolum.zip' AddToZip
   ⊂'c:\Comparing the Excel COM Interface.doc'
      ⍝ Add multiple files
files←'C:\aa(2)\DSC00001.JPG'
'C:\OURFILES\AJAY\APL\SHELL\SHELL.W3'
'c:\ajayaskoolum.zip' AddToZip files
```

Using Winzip to Explore

The file can be opened by WinZip; see below.



List File Names in Zip File

The list of files in a given Zip file can be returned by this function:

```
    ∇ Z←L EnumZip R;⎕wself;i;j
[1]   ⍝ Ajay Askoolum
[2]   ⍝ Return list of files found in Zip file R
[3]   :if 0=⍴'ShApp' ⎕wi 'Self'
[4]      ⎕wself←'ShApp' ⎕wi 'Create' 'Shell.Application'
[5]   :else
[6]      ⎕wself←'ShApp'
[7]   :endif
[8]   Z←0/⊂''
[9]   :if 0≠⎕wi 'xNamespace()' R ⍝ Verify existence
[10]     ⎕wi 'xNamespace().Items>.tr' R
[11]     :if 0≠i←'.tr' ⎕wi 'xCount'
[12]        i←(-⎕io)+⍳i
[13]        j←(⊂'.tr') ⎕wi¨ (⊂⊂'xItem().Path'),¨i ⍝ Contents
[14]        L←(⊂'.tr') ⎕wi¨ (⊂⊂'xItem().IsFolder'),¨i
[15]        Z←Z,(~L)/j                              ⍝ Files
[16]        :if 0≠⍴L←L/j                            ⍝ Folders
[17]           R←(i≥+\'\'=R)/R
[18]           Z←Z,⊃,/EnumZip¨(⊂R,'\'),¨L
[19]        :endif
[20]     :endif
[21]  :endif
[22]  :if 1=+/'EnumZip'^.⍷⎕si[;1 7]
[23]     :if 0≠⍴'.tr' ⎕wi 'self'
[24]        '.tr' ⎕wi 'Delete'
[25]     :endif
[26]  :endif
    ∇
```

Note that this is a recursive function.

```
      EnumZip 'c:\ajayaskoolum.zip'
Comparing the Excel COM Interface.doc  DSC00001.JPG
SHELL.w3

      ⍴EnumZip 'c:\ajayaskoolum.zip'
3

      ⊃EnumZip 'c:\ajayaskoolum.zip'
Comparing the Excel COM Interface.doc
DSC00001.JPG
SHELL.w3
```

Extracting Files from Zip File

The following function will extract the files from an existing Zip file into a specified location:

```
     ∇ L ExtractZip R;⎕wself
[1]    ⍝ Ajay Askoolum
[2]    ⍝ L is the target location for extracted files
[3]    ⍝ R is the fully qualified name of Zip file
[4]    :if 0=⎕nc'ShApp' ⎕wi 'self'
[5]       ⎕wself←'ShApp' ⎕wi 'Create' 'Shell.Application'
[6]    :else
[7]       ⎕wself←'ShApp'
[8]    :endif
[9]       ⎕wi 'NameSpace().CopyHere' L ((⎕wi 'NameSpace().Items' R) ⎕wi 'obj')
     ∇

       'c:\zzz' ExtractZip 'c:\ajayaskoolum.zip'
       ⎕lib 'c:\zzz'
Comparing the Excel COM Interface.doc
DSC00001.JPG
SHELL.W3
```

Note that the target location must exist.

What is Missing?

Lots! If you examine the properties, methods, and events of the Shell object, it is clear that this object does not expose the compression properties or methods. It is possible to do lots more with these facilities; I'll leave you to explore.

Analysing CONTINUE Workspaces

A light-hearted look at the Dyalog CONTINUE workspace, and what to do with it

by Ray Cannon (ray_cannon@compuserve.com)

[This article contains annoying and exuberant references to obscure artefacts of British culture, inappropriate to a serious journal of international reputation. Our first scheme was liposuction: to edit them out as waste tissue. However, investigation revealed them inseparable from the skeleton and sinews of this piece. With the deepest regret, we print them as submitted. *Ed.*]

They sought it with thimbles, they sought it with care;
They pursued it with forks and hope;

I just tried googling for "continue.dws" and came up with a mere "7 English pages", none of which appeared to be related to Dyalog APL. So, not many people out there in the Wild Word World of Googleland appear to know about Dyalog's CONTINUE workspace.

But *we* all know it is what you get when you cross a bug in a workspace with the Dyalog runtime interpreter.

Drunk in Charge

There are only two types of vodka, good vodka and great vodka.
There is no bad vodka, just not enough vodka.

There are only two types of code, Untested Code and Part-Tested Code. There is no Fully-Tested Code, and there is never enough vodka.

Errors in code fall into three types:

1. errors that work, but produce the wrong answer; e.g. two items are added rather than subtracted;
2. errors that cause the code to crash; e.g. dividing by zero;
3. errors that trigger error trapping; e.g. dividing by zero under error trapping.

CONTINUE workspaces only result from the second, and then only ever under runtime Dyalog APL.

Hunting the Snark

He had softly and suddenly vanished away –
For the Snark was a Boojum, you see.

Just like out-of-date comments in the code, I lied. There is also a fourth type, the System Error (SYSERROR). Unlike CONTINUE workspaces, the developer's version of Dyalog supports them.

`aplcore` is the name of the file on disk containing the snapshot of the core memory used by Dyalog APL, at the point the interrupter discovered the system error.

A developer can cause APLCOREs by incorrect calls using `⎕NA` or OLE links etc., to external features. Bugs in the interrupter can also result in a SYSERROR. (Dyalog would like to know about the latter. In particular, they would like to know how to reproduce the SYSERROR with the minimum of code.)

Since it cannot be trapped, or directly recovered from, it is often very hard to discover exactly where in the code the SYSERROR was triggered. It may, however, be possible to extract data from the APLCORE workspace, although sometimes this in turn results in another Boojum.

Fit the First – The Landing

"Just the place for a Snark!" the Bellman cried,
As he landed his crew with care;

The first problem that we (the APL developers) have with the CONTINUE workspace is: How do I get hold of it? Because unlike "real programmers" (the Bellman and crew don't eat quiche or Beaver but do run `dyalog.exe`), the "real users" only ever use `dyalogrt.exe`. So for us, the CONTINUE workspace normally appears on someone else's PC.

I gave up expecting users to send me their CONTINUE workspaces merely because I asked them to. They will do so only if their jobs (not mine) depend on them doing so.

The question becomes "How do we make the CONTINUE workspace into a homing pigeon or boomerang?" (Fade up Charlie Drake singing "My boomerang won't come back".)

The Star Wars Guide to the Galaxy

(In which Darth Vader and the Vogons meet Dr Doolittle and Arthur Dent, and they all get their Blue Peter Badges.)

The APL developer has the basic PushMePullYou choice of the "tractor beam" or the "bulldozer" approach:

- The tractor beam (driven by "Darth Vader, developer from hell") locks onto a CONTINUE workspace floating out there in the network of space but within range, and pulls it into the Death Star.
- The bulldozer (driven by the Vogon user acting as a sub-contractor), pushes the CONTINUE workspace across the network (over Arthur Dent's house) into the arms of the patiently waiting (property) developer Zaphod Beeblebrox.

I prefer the tractor-beam approach. The application workspace need know nothing about the developer, it needs no general error trapping, and it has to do nothing. It is powerless to stop itself being dragged into the clutches of the Evil Empire. It also works on APLCOREs.

Alternatively, you have to build the bulldozer (or Vogon Constructor fleet) into every workspace, setting up complex Snark traps in the hope you won't catch a Boojum.

The main drawback to the tractor-beam approach is that the target must be in range. That is to say, the CONTINUE workspace must be visible (readable) by you, Darth Vader, across the network, and at a known location. Whereas, the Vogons, in their bulldozer, need know only Arthur Dent's home planet address; i.e. your PC's IP address or name on the network.

A third alternative is the "here is one I made earlier" Blue Peter approach. The workspace at start-up, before normal processing, looks for an existing (earlier) CONTINUE workspace and, if found, sends it across the internet back home. This has the advantages of both the tractor beam (no error trapping required) and the bulldozer (can work from an unknown location), but the downside that you only receive the CONTINUE workspace the next time that user runs the system. Like the tractor beam, it also can work with APLCOREs.

A Joint Spanish Inquisition/KGB Production

You may threaten its life with a railway-share;
You may charm it with smiles and the comfy chair.

The developer, having got the poor defenceless CONTINUE workspace into his grubby hands, is now able to `)XLOAD` it into the development APL environment, and, using all the tools of the inquisitor, can extract all the information (who, what, how, when, where, etc.) contained within. (Film: Think *Marathon Man*, Dustin Hoffman, Laurence Olivier. Oh *no* – not the dentist's chair!)

The two most important pieces of information contained in the CONTINUE workspace can be found in `⎕DM` and `⎕SI`. However, if you have used the bulldozer, you may need to defuse the error trapping – before the booby trap blows up in your face. You can then safely cut the stack back to the Snark.

Other useful information about the (real) user and their environment is, available from:

```
⎕AN

⎕WSID

GetCommandLine

⎕WG 'AplVersion'
```

Knowing the IP address, the version of Microsoft Word and the operating system version etc., would also be nice. But from the `)XLOAD`ed workspace can be read only information about the developer's environment on *your* PC, not the user's runtime environment on *his* PC.

So I suggest, again in true Blue Peter fashion, that you get this information using the "Here's one I made earlier" approach...

Go back in your time machine, and either:

- Following the error triggering the error trap, before starting up your bulldozer, save all the information you need;

or do as I do and:

- Get your workspace's start-up function to save *all* the environment information you can lay your hands on into a global variable.

Then, when your tractor beam captures a workspace, it already has this information nicely stored away. (This is also true even if your Boojum has turned into an APLCORE.)

Errorbot

As he wrote with a pen in each hand,
And explained all the while in a popular style

If your users are bent on producing lots of errors you may wish to automate the documentation of them, via an "errorbot" which, without your intervention, quietly extracts the end-user environment and error information, and then (with a pen in each hand) writes it to log file, printer, web page and email.

Under Dyalog APL it is possible, but not straightforward, to create an environment that can automatically extract the ⎕DM and ⎕SI as text from a CONTINUE workspace. I will leave that as an exercise for the advanced user to work out. (I believe it is also now much simpler under Version 11 to extract this info from an APLCORE workspace.)

Alternatively, your bulldozer error trap can attempt to save this information at the point of the error into variables known to your pet errorbot, from where a simple ⎕CY command can extract it.

Code Snippet

N.B. The last (fourth) line of APL is split onto several lines, as shown by ellipses. ⎕ML is assumed to be 0 or 1. The code is ⎕IO-independent.

```
#.quaddm←⍬          ⍝ initialise ⎕DM store
#.quadsi←0⍴⊂''      ⍝ initialise ⎕SI store
continue←''''',(ExpandPath
⎕WSID,'..\..\continue.dws'),''''
⎕TRAP← (0 'E'(
  ... '⎕TRAP←(0 1000) ''S''
  ... ⍝ #.quaddm,←⎕dm
  ... ⍝ #.quadsi←↑⎕XSI,⍝{''['',(⍕⍵),'']''}¨⎕LC
  ... ⍝ ⎕save ',continue,'
  ... ⍝ ⎕off '))
```

Restarting and Resetting

Just the place for a Snark! I have said it twice:
That alone should encourage the crew.
Just the place for a Snark! I have said it thrice:
What I tell you three times is true.

If you)XLOAD a CONTINUE workspace, it may be possible to correct the problem and then restart the workspace to test if the fix has worked.

File ties are independent of saving or loading workspaces. So an)XLOADed CONTINUE workspace will need any files retied before restarting.

I have found that by:

[1] using)RESET to clear the stack;

[2] running the workspace to a stop placed near where the error occurred;

[3] without untying the files,)XLOADing the CONTINUE workspace again

we are almost ready to restart the workspace.

Unfortunately, the CONTINUE workspace does not contain the ⎕PATH at the point of the error. So before restarting, you may need to set the ⎕PATH.

CONTINUE workspaces preserve even **OLE links** to Microsoft Word, etc.!

Now activate **Debug Mode** via Ctl-Enter, and you're ready to roll. (Or at least roll over dead with the same error the user got.) If you can't reproduce the error using his data, start by looking at the differences between his environment and yours.

A pre-Version 11 APLCORE cannot be)XLOADed but data may sometimes be **copied** into the active workspace with)COPY or ⎕CY. (An)XLOAD is preferable because it preserves the stack.)

To determine, without the stack, where the SYSERROR occurred, I find it useful to ⎕CY the whole of the APLCORE into a clear workspace, and then look at all the **local variables.**

```
      )CLEAR
CLEAR WS
    ⍝ use ⎕CY because )COPY omits local variables
    ⍝ APLCOREs don't have .DWS extensions, so use full
pathname
      ⎕CY 'C:\Dyalog\APLCORE.'
```

Since the local variables of functions on the stack at the point of the SYSERROR are accessible, their names and values can be used to determine which functions are in the stack. Distinctive local names help this process; reusing similar names, (e.g. `x, y, r, data, text, mat, name`) make it far more difficult.

Here is one I made earlier

"Two added to one – if that could but be done,"
It said, "with one's fingers and thumbs!"
Recollecting with tears how, in earlier years,
It had taken no pains with its sums.

If you really want to know exactly which functions are active at the time a SYSERROR occurred, try putting into every function a label that identifies the function's name. For example, for a function `Foo`, name the label `Lab_Foo`.) Only the labels of functions in the stack will be `⎕CY` copied from the APLCORE, so listing all variables starting with `Lab_` will list all the functions on the stack at the time of the SYSERROR.

Unfortunately you can't put a label in a D-function, but if the Dfn is *dynamically* created and localised, it is not a problem. Then the very fact that the Dfn has been copied by `⎕CY` proves it was on the stack at the time of the SYSERROR.

Rules of Thumb

1. Untested code always fails in production.
2. Part-tested code that fails in production often does so on "edge conditions".
3. That the code works with 2, 3 or 4 elements, but fails with a scalar, is a mistake typical of a newbie. A more experienced writer's code might fail with *zero* elements. Well-tested code could still fail with twenty million elements (or fewer) due to `WS FULL`.
4. Virtually all code will fail at the extremes, or if the environment is changed.

"Have you stopped beating your wife yet? A simple yes or no answer is all that is required."

The fourth is its fondness for bathing-machines,
Which it constantly carries about,
And believes that they add to the beauty of scenes –
A sentiment open to doubt.

The following examples are all taken from the pensions industry.

A person's sex, (ignoring sex changes, and other medical oddities) can simply be held as a Boolean, e.g. `0` for male and `1` for female. However there is a third possibility, "Not applicable". The sex of someone's spouse is N/A if there is no spouse. Hence the '3-way Boolean': `M` for male, `F` for female and `N` for not applicable.

A frequent fault is caused by common code running off variable data.

One pension system I have worked on handles over 220 pension products from five former insurance companies, now all merged, but still having data held on the five original mainframe systems. Feeds from these differ by product and mainframe.

Failing to reconcile data codes resulted in errors. For example, when the code for a payment frequency (monthly, quarterly, half-yearly etc.) was set to `A` for 'annual' on one mainframe and `Y` for 'yearly' on another, errors were encountered.

Tip: When writing a case statement, list all known cases and let the default `ELSE` be an error. (Rival UK adverts come to mind: "Kills ALL known germs" "Kills 99.9% of all germs".)

```
:Select payment_frequency
:Case 'M'
    …
:Case 'Q'
    …
:CaseList 'A' 'Y'
    …
:Else
    ERROR
:End
```

Do not be tempted to let the `:Else` replace the `:CaseList 'A' 'Y'`. That way, when the code gets a `F` (Fortnightly? Four-weekly?), it is rejected and does not default to the inappropriate Annual case.

"I said it in Hebrew – I said it in Dutch –
I said it in German and Greek:
But I wholly forgot (and it vexes me much)
That APL is what you speak!"

Well that's all I have time for now.

"What's the good of Mercator's North Poles and Equators,
Tropics, Zones, and Meridian Lines?"
So the Bellman would cry: and the crew would reply
"They are merely conventional signs!"

(With apologies to Lewis Carroll, Douglas Adams, George Lucas, and Dyalog. However, I make no apologies whatsoever to the Spanish Inquisition or the comfy chair.)

New Tricks for Old Dogs
Making Sense of Classes and Namespaces

by Adrian Smith (adrian@causeway.co.uk)

Talk given at the FinnAPL Forest Seminar Spring 2007

There is a well-worn proverb in English that "You can't teach an old dog new tricks", which is not entirely true when it comes to APLers. It just takes a little more time and patience as we get older. In my case, I have been avoiding *Classes* in favour of *Namespaces* as long as I can, on the grounds that I understand what a namespace is, and I know how to make it work for me.

Classes looked quite attractive when Morten first showed them in 2005, and they make for great demos. However I have always had a feeling that trying to maintain any reasonable volume of code in a **script** would be anathema to my APL soul. SharpPlot (aka RainPro) has 585 functions and totals around 20,000 lines of APL. I know what most of the functions are called, and I have Shift+F1 set up to `')ed #.SharpPlot.'` at which point autocomplete can help me type stuff. I can start anywhere in the calling tree, and Shift-Enter my way to where I want to be. The very idea of giving all this up in favour of a dumb script makes me want to hide in a corner and whimper.

But I do like the idea of *instances* which show up only the *public functions*, and have assignable *properties* and maybe global variables. I definitely like the idea that I can write some testing code for the SharpPlot namespace that will look just like the code that runs against the true .Net DLL version. This way I can run the same test against both versions, check the speed, and check that I get exactly the same chart back in either case. So what to do. In a weak moment, I decided to read the instructions. This had no effect whatsoever, so I went out and hit a bucket of golf balls. This somehow triggered a mad idea, so I came home and read the instructions again. This time, a couple of things made sense.

Enlightenment – on Reading the Instructions

If you take time to browse through the "Latest Enhancements" help file that came with Dyalog 11 at this year's conference, you will see that you can use `⎕FIX` to turn a *script* (vector of text vectors) into a *class*. What had escaped me on first reading was the fact that if the script omitted a class name, it would create an *unnamed* class that would persist in the workspace only while something was

using it. Of course that *something* could be an instance, created by ⎕NEW. What is more, the script could :Include any number of namespaces, and flags such as :Access Public in functions in included namespaces would be preserved and taken note of. Wow, thought Adrian, let's give this a try...

```
      qq←⎕NEW ⎕FIX':Class' ':Include Tester' ':EndClass'
      qq.Hello
Hello, from Tester
```

We can even use ⎕DF to make it look pretty:

```
      qq
#.[[Unnamed]]
      qq.⎕DF 'myTester'
      qq
myTester
```

This looks very promising – I can use a trivial script to bridge completely over the *class* and effectively just make an instance of my source *namespace*. Oddly, if I copy the namespace in from an earlier version of APL, everything ends up **Public** by default. Refreshing all the functions by refixing them solves the problem, and only functions marked with :Access Public show up – this may be a bug in the workspace upgrader, and it is easy to work around if it turns out to be intentional.

Another big benefit of this approach is that there is only one copy of the source code, and that if I change it, all existing instances see the change immediately:

```
      )ed Tester.Hello
      qq.Hello
Hello, from Tester!
```

... note the extra character on the end of the message! All my old habits (like leaving jots in functions to stop them) just work. Functions called via qq duly stop. I can edit them, fix up the code, save them and continue to the next jot. When I am done, I really did change the working copy of the code back in the original namespace – which makes me very happy indeed. If there are any horrible snags, I have yet to find them!

So what else do I need to do to make this (very useful) idea into a workable utility? Well, it must scan the source namespace for *Properties* and create the correct stuff in the script so that the temporary class knows which Get and Set functions to call. It should also call the *Constructor* function to make sure any data has been set up correctly to default values, and that any arguments to ⎕NEW are passed down when the class is instanced. It is also an excellent idea to have a

default constructor which Dyalog runs for you when you overtake an array of instances. Time to make a start!

Making it Work for Real

Properties

These are really just a comfortable syntax for a pair of Set and Get functions that maintain the value of some state variable. They are often trivial (like `SetHeading` and `GetHeading` in SharpPlot) but may do a little validation (in the Setter case) or formatting (in the Getter case) of the internal value. In SharpPlot, I chose to mark them out with comments like

```
A:PSet Heading The main chart heading
```

which my C# translator uses to make the appropriate markup in the translated class (it also extracts the extra text for the XML that leads to the Visual Studio tips). Alternatively you could simply define a global variable in the namespace that gave the property name, type, description, Getter and Setter for all properties. I quite like the former approach, as maintaining extra data can be a pain, but for SharpPlot I can easily create my property list automatically, so with around two hundred properties to scan for, it will save the script-writing function a lot of work! Use whichever method you prefer, or invent your own.

Constructors

I had half hoped that if I added `:Implements Constructor` into my namespace initialiser, Dyalog would just run it for me – maybe they should and this is a bug, but I can see that you might get some collisions if you included several namespaces, so maybe it is only fair to have to do this myself!

My convention (following the C# rules) is that the constructor always has the same name as the source namespace, so `Tester.Tester` in the above example. I can easily scan for this name, use `⎕AT` to see if it requires an argument, and add the requisite entries into my script. Something like:

```
ss←':Class' ':Include Tester' '⍝ Create arg'
… ':Access Public' ':Implements Constructor'
… 'Tester arg' '⍝ ' ':EndClass'
  qq←⎕NEW (⎕FIX ss) 'Hello'
```

This now correctly calls my own initialiser with the text `Hello` which is fine if I remember to pass an argument to `⎕NEW`, but if I get sloppy:

```
      qq←⎕NEW ⎕FIX ss
LENGTH ERROR
      qq←⎕NEW ⎕FIX ss
     ^
      3↑qq
LENGTH ERROR
      3↑qq
     ^
```

The answer is to provide an extra *Default contructor* which Dyalog can call niladically, and which will call my own initialiser with a sensible default argument:

```
    ∇ CreateDefault
[1]   :Access Public
[2]   :Implements Constructor
[3]   Tester ''
    ∇
```

By adding these lines, I can now have Dyalog make the instance with no data, so it can also overtake the scalar instance to give me a vector of them:

```
      qq←⎕NEW ⎕FIX ss
      (3↑qq).Hello
 Hello, from Tester!  Hello, from Tester!  Hello, from
Tester!
```

At last – I have complete control over how overtake does its prototyping! Just using this for vectors of data items (like name/age/salary triples) could make a bit of sense, and it is not madly slow, even with many thousands of 'records' in the array. If I am prototyping in APL with a view to compiling the finished application, then speed is the last thing I care about.

Of course, all of this is now well buried in a function called `new` which does the messy stuff where I never need to see it again. Copies are likely to be given away to anyone with a keydisk at APL meetings, and will be mailed to anyone who asks me for one. As you might guess, it is still steadily evolving.

Wrap Up

My feeling is that this is quite a breakthrough. I am most of the way through rebuilding NewLeaf for .Net and I am doing it properly using a very object-based approach. I have an object called a `TextBlock` which knows how to handle formatted text (with font changes, superscripts and so on) very nicely. If I make an object called a `Cell` which is just a `TextBlock` with some additional properties like background colour, then a `Table` simply becomes a matrix of `Cells` and much of the complexity of handling multipage tables just falls away.

The big snag is that debugging this in Dyalog 10 is a pain, as you can only pretend to make instances by copying the entire source namespace. I can't face moving the code into true V11 Classes, for all the reasons outlined at the top of this note. But I can believe in the idea of typing things like:

```
lf←new LeafEngine mylayout
```

... and getting a true instance of my `LeafEngine` namespace. I don't even have to enclose `mylayout`, but that's another story...

The Ruler's Edge

by Stephen Taylor (sjt@5jt.com)

This article begins the *In Session* columns, in which working programmers share from their session logs fragments that are useful or instructive (or just plain showing-off, like this piece) but don't warrant an entire article. Please send your contributions to editor@vector.org.uk. *Ed.*

A Zen master famously began each day with the following conversation with himself.

— Master! Master!
— Yes? Yes?
— Wake up! Wake up!

It's morning. Time to work. But are you awake enough? It's not a silly question. To write well you want to be not just eyes-open awake, but *razor*-sharp awake. Coffee and wild dance music can do only so much.

To work, you need your duty sleep.

Sadly, our subjective sense of sleepiness is *entirely* unreliable. Hans van Dongen has shown [1] that as sleep deficit accumulates, cognitive impairment rises steadily; but the subjective sense of sleepiness plateaus after an initial rise. We know we're short of sleep; we feel a touch sleepy; but it seems manageable. And then we find it hard to finish sentences...

Are you ready to *cut code*? Or should you grab a nap – at worst fill in time with less demanding work? Here's my Morning Rule to decide it. If I can write the Ruler's Edge in the session, I'm sharp enough to work.

Dyalog installs with the session log configured to an 80Kb limit. I reset this to 8Mb; I need a log longer than my memory. Searching a long log is easier when it's divided into sections, so I like to rule a line under each day's work; and sometimes between different tasks during the day.

A good ruler shows units of measure. Character widths are less useful since laser printers and proportional fonts replaced line printers; but they're still helpful for visually parsing displayed arrays. So I want a function `rule`. Its right argument will give the length of the line it draws; its left, the intervals at which numbers are displayed. Let's say the left argument defaults to `10`, and that these periods are marked on the line by inverted carets, somewhat like notches on a physical ruler.

We'll write the numbers above the line. So the gross structure of the function is clear:

```
rule←{⍺←10 ⋄ ↑(exp1)(exp2)}
```

where `exp1` returns the superscribed numbers, and `exp2` the notched line.

Cutting the Notches

Let's start with the notched line. This is a pattern ⍺ characters long, reshaped to length ⍵:

```
⍵⍴'¯\/' {something} ⍺
```

For an ⍺ of `10`, we need `something` to return `9 1`; in the general case,

```
(⍺-1)(1)
```

So

```
      10 {⍵⍴'¯\/'⍤(⍺-1),1} 55
---------\---------\---------\---------\---------\-----
```

does it. But the truly awakened mind sees unity in the two `1`s. They could both be, say, `2` or `3` (if one wanted double or triple notches) but they must be the same number. So we factor out the repetition:

```
      10 {⍵⍴'¯\/'⍤(,⍤⊢⍤(⍺⍴-)⍤⊢)1} 55
---------\---------\---------\---------\---------\-----
```

Numbering the Notches

Now for the numbers.

We want to select from the numbers up to ⍵, so `exp1` will be something like

```
{(-⍺)↑⍳⍨⍺ select ⍴⍵}
```

where `select` selects only the elements of its right argument that are multiples of its left; i.e.

```
{⍵/⍤0=⍺|⍵}
```

or:

```
{⍵/⍤~×⍺|⍵}
```

or, avoiding nesting D functions:

```
/⍨∘~⊚×⊚(⍺∘|)⍨
```

Putting that together:

```
10 {⊚,/(-⍺)↑¨⍕¨(/⍨∘~⊚×⊚(⍺∘|))⍨⍳⍵} 55
  10        20        30        40        50
```

and substituting for exp1 and exp2:

```
rule←{⍺←10 ⊚ ↑(⊚,/(-⍺)↑¨⍕¨(/⍨∘~⊚×⊚(⍺∘|))⍨⍳⍵)(⍵⍴'¯v'/⍨(,⍨∘(⍺∘-)⍨)1)}

      rule 55
        10        20        30        40        50
¯¯¯¯¯¯¯¯¯v¯¯¯¯¯¯¯¯¯v¯¯¯¯¯¯¯¯¯v¯¯¯¯¯¯¯¯¯v¯¯¯¯¯¯¯¯¯v¯¯¯¯¯

      15 rule 55
             15             30             45
¯¯¯¯¯¯¯¯¯¯¯¯¯¯v¯¯¯¯¯¯¯¯¯¯¯¯¯¯v¯¯¯¯¯¯¯¯¯¯¯¯¯¯v¯¯¯¯¯¯¯¯¯¯
```

OK: we're awake – time to get to work!

Reference

[1] Hans P.A. Van Dongen, PhD; Greg Maislin, MS, MA; Janet M. Mullington, PhD; David F. Dinges, PhD "The Cumulative Cost of Additional Wakefulness: Dose-Response Effects on Neurobehavioral Functions and Sleep Physiology From Chronic Sleep Restriction and Total Sleep Deprivation" *Sleep* 26.2 pp117-126 http://www.journalsleep.org/citation/sleepdata.asp?citationid=2198

PROFIT

Chance Misunderstandings

by Sylvia Camacho (sylviac@blueyonder.co.uk)

I wrote this article before the Editor pointed out that Devon McCormick had treated the same topic in 21.1 [1]. How could I have missed it? No sign of 21.1 among *my* shelf of Vectors so I went to the obvious hiding place, *Anthony's* shelf of Vectors: Eureka! On reading Devon's piece, I think our take on the story is somewhat different. He gives a very full account of its history and the psychological ramifications of the game itself: I am interested in the psychological response of one of the mathematicians who had such problems with it.

> *Up to this point, I have spoken of numeracy as if it were an all-or-nothing quality. Either you have it or you don't. In truth, there are grades of numeracy and even the most numerate have their bad moments.*
> *A. K. Dewdney*

I first read about the Monty Hall dilemma in Dewdney's book, *200% of Nothing* [2] and I had to think about his argument for a day or two before I could convince myself that I understood how it works. However, when I tried to pass this fascinating insight on to my family, it led to a kitchen-based altercation which came near to burning the supper. It was a marvellous confirmation of the outraged reaction the story so often meets at first hearing.

It all began in 1990 with a column in an American magazine, written by Marilyn vos Savant, who specialised in answering brain-teasers. She was asked about a TV game show and how she would advise the contestant to act. I came across a typical account of the game in the Spanish version of *Scientific American*, which translates as follows [3]:

> Monty Hall is the presenter of a television competition in the United States. In the last part of the competition Monty shows three chests to a long-suffering competitor. In one of them there is a grand prize and the other two are empty. The competitor nervously selects one of the chests. Monty then puts aside the selected chest and looks slowly and theatrically into the inside of the other two. He closes one of them again, takes the other in both hands and upturns it before the eyes of the competitor and the audience: it is empty.

> The competitor sighs with relief although there is no reason to do so. Monty generously shows him the two chests which remain closed and offers him the possibility of changing his initial decision: "You may now choose either of them," he announces, as the spotlights brighten. What ought the competitor to do?

Marilyn's answer, that the competitor is more likely to win if he accepts the offer to abandon his first choice, provoked intense controversy even among mathematicians, some of whom wrote to her in abusive terms.

I was intrigued to come across the story again in Paul Hoffman's delightful book about Paul Erdös, who is known to some at least of the *Vector* community from Eugene McDonnell's article in 17.4 explaining Erdös Numbers. There is a conceit among mathematicians in the style of the 1920s music-hall song, "I danced with a man who danced with a girl who danced with the Prince of Wales." It seems that Roger Hui has an Erdös Number of 2, as he co-authored a paper with a man who had co-authored a paper with the great man himself. Eugene was at first credited with number 3, having written a paper with Roger, but then he was up-graded to 2 by the discovery of a different collaboration. Our Editor tells me that he is an Erdös 4. Never having written a mathematics paper and being wholly unqualified to do so, I am at the bottom of the pecking order with Erdös Number ∞, so reading about his reaction to the Monty Hall dilemma was very reassuring.

It seems that a friend told Erdös of the fuss that the dilemma had caused [4 p237]:

> Vázsonyi told Erdös about the Monty Hall dilemma. "I told Erdös that the answer was to switch," said Vázsonyi, "and fully expected to move to the next subject. But Erdös, to my surprise, said, 'No, that is impossible. It should make no difference.' At this point I was sorry I brought up the problem, because it was my experience that people get excited and emotional about the answer, and I end up with an unpleasant situation. But there was no way to bow out, so I showed him the decision tree solution I used in my undergraduate Quantitive Techniques of Management course."

Vos Savant had used the same technique on her readers but did not succeed in convincing all of them and Erdös was similarly resistant:

> "An hour later he came back to me really irritated. 'You are not telling me why to switch,' he said. 'What is the matter with you?' I said I was sorry, but I didn't really know why and that only the decision tree analysis convinced me. He got even more upset." Vázsonyi had seen this reaction before, in his students, but he hardly expected it from the most prolific mathematician of the twentieth century.
>
> "Physical scientists tend to believe in the idea that probability is attached to things," said Vázsonyi.

The argument turns upon the exact circumstances of the game: "the total experimental situation," as Niels Bohr once said. Since *it is a game* one can assume that the chest which Monty shows to be empty, has been *chosen by him* to be an empty one, or the 'game' would be over. That being so this is a game of two parts. The competitor's initial choice has a probability of one-third of winning but that choice *creates a second probability* for the remaining two chests, which together have a two-thirds chance of including the prize. At this point one of these two is eliminated, investing all the two-thirds chance in the other.

Choose Box 1 & stick

box 1	box 2	box 3	outcome
prize			win
	prize		lose
		prize	lose

Choose Box 1 & switch

box 1	box 2	box 3	outcome
prize			lose
	prize		win
		prize	win

It is the change of probability, consequent upon the opening of the box, which is difficult for some to accept. Unless the contestant sees it chosen by Monty and that it is empty, he will have no reason to alter his original choice. Vos Savant drove this lesson home by pointing out that an outsider who was not present at the game, if asked to assign probabilities for a prize to be in either of the two boxes still closed, could necessarily only assume the odds to be even, 50:50. The dilemma only arises if Monty's intervention is seen to be deliberate and part of his game plan. This makes it a psychological tease, but the lesson that probability is never a property of objects, but only of the changing situations in which they are placed, can cause confusion even when the pertinent situation *does not change*. Eugene Northrop published a little 'book of paradoxes' [5] in 1944, from which the following example is taken.

Imagine three closed purses containing respectively two gold coins, two silver coins and one with one gold and one silver. If we choose a purse at random we can agree that the chance that it holds the mixed coins is one-third. If we take a coin from the selected purse we can agree that it will be either like the second coin or unlike it. So can we say that the chance is one-half that the one we removed differs from its companion? If this is what we think, does that mean that removing a coin has raised the chance of having selected a mixed pair, from one-

third to one-half? Of course not, merely transferring the coin from the purse to the hand changes nothing: the probability of having selected unlike coins, given the total experimental situation, is still one in three. Moreover, if we look at the coin in our hand and find it is gold, we have a one-in-three chance that the second coin in our selected purse is also gold, although there are only two gold but three silver still hidden from us.

In both of these puzzles we are having to consider how objects are grouped. In both we start with three equal possibilities. Monty Hall then creates two groups. The group of one has a one in three chance of holding the prize. In the group of two each starts as a one-in-three chance, making two-thirds in total, which does not change when one is shown to be empty. In the Northrop puzzle there are three groups of two but the possibilities associated with the other two groups must be disregarded after the initial choice. This still leaves three possibilities for the contents of the chosen purse; gold-gold, silver-silver, gold-silver. So in the Monty Hall case the probabilities must be re-calculated after the initial choice but in the Northrop case they cannot be.

Those who know my prejudices will realise that I am still pursuing my objections to the assumption behind Bell's Inequality [6], that probability is a fixed property of the phenomena under test and does not change with the changing experimental situation.

Erdös was unconvinced by the simple decision table but reluctantly accepted that vos Savant was right after Vázsonyi ran a Monte Carlo simulation on his PC. I have never written J, so as an exercise I have appended some code to run a succession of randomised games and collect the statistics. I know that some of my readers will wince and convert my crude attempt into an elegant form more worthy of Erdös: "a proof from the Book", as he would have said. My amateur J code ran an iteration of one hundred thousand simulated Monty Hall games in ten seconds on my 600MHz Pentium III:

```
   play 100000
Total Games Played: 100000
        switch     no switch
win      33379       16516
lose     16613       33492
sum      49992       50008
% wins      66          33
```

```
11=: 40{.' Total Games Played: nnnnnn'
12=: 40{.'           switch     no switch'
13=: 40{.' win       nnnnnn     nnnnnn'
14=: 40{.' lose      nnnnnn     nnnnnn'
15=: 40{.' sum       nnnnnn     nnnnnn'
16=: 40{.' % wins       nn         nn'
play=: 3 : 0
x=: 0 0 0 0
games (^:y.) 0
11=:(6.0":y.) (21+i.6)}11 NB. total games
13=:(6.0":2{x) (10+i.6)}13 NB. switch & win
13=:(6.0":1{x) (21+i.6)}13 NB. no switch & win
14=:(6.0":3{x) (10+i.6)}14 NB. switch & lose
14=:(6.0":0{x) (21+i.6)}14 NB. no switch & lose
15=:(6.0":(2{x)+3{x)(10+i.6)}15 NB. total wins
15=:(6.0":(1{x)+0{x) (21+i.6)}15 NB. total losses
16=:(2.0":<.100*(2{x)%(2{x)+1{x)(14+i.2)}16 NB.% if
switch
16=:(2.0":<.100*(1{x)%(1{x)+0{x)(25+i.2)}16 NB.% if no
switch
stats=: 6 40$11,12,13,14,15,16 NB. display results
)
games=: 3 : 0
p=.1?3 NB. assign prize to box 0,1 or 2
c=.1?3 NB. contestant chooses 0, 1 or 2
s=.1?2 NB. contestant chooses stick=0 or switch=1
n=.#.s,p=c NB. n is outcome 0,1,2,3
x=: x+0 0 0 0 (n}~) 1 NB. accumulate wins & losses
)
```

References

[1] Devon McCormick, "Further Surprises involving a Man and a Goat: The 'Monty Hall' Problem Solved", *Vector*, 21.1 pp85-89

[2] A.K.Dewdney, *200% of Nothing*, 1993, John Wiley & Sons Inc.

[3] *Investigación & Ciencia*, Prensa Científica, Barcelona, Dec 2001, ISSN 0210136X

[4] Paul Hoffman, *The Man Who Loved Only Numbers*, 1998, Fourth Estate Ltd.

[5] Eugene P. Northrop, *Riddles in Mathematics*, 1960, Pelican Books

[6] Sylvia Camacho, "How Wrong Was I?", *Vector* 21.2 pp78-90

3-D Cellular Automata and the Game of Life

by Timothy K. Zirkel (zirkelt@lafayette.edu)

Cellular automata are rules applied to structures. Mathematicians, computer scientists, and theoretical biologists use them to study the behaviour of systems based on local neighbourhoods within the system. The general principle is that a system is modelled by cells in some finite-dimensional space. Each cell can take one of a finite number of states. The next generation of the system is determined by applying some pre-determined rule to the neighbourhood of each cell. Modelling cell growth or crystallography are the most common uses of cellular automata, though they are also applied in other fields, such as cryptography [2].

I first became aware of the Game of Life in an introductory computer science class. Our class was given the assignment of implementing Life using Java. Later, I encountered Life in a course on mathematical visualization. My visualization class implemented Life using J. I found the J version to be much more elegant and intriguing than anything I had seen in other programming languages. As a final project for the class I chose to investigate using J to expand the Game of Life to three dimensions, and that project evolved into this note.

The 3D Game of Life

The Game of Life is a two-dimensional Boolean cellular automaton. It was created by John Conway in 1970. Cells reside in a square grid and can be in one of two states: 'alive' or 'dead'. Each cell is surrounded by eight neighbours. A dead cell has a 'birth' if it has exactly three neighbours. An alive cell stays alive if it has two or three neighbours. All other configurations will result in a dead cell (either a dead cell remains dead, a living cell dies from overcrowding, or a living cell dies from exposure) [1]. For an implementation of the Game of Life in J, see [5] or Section 6.3 of [4].

The concept of Conway's Game of Life can be extended to three dimensions. There are, of course, a variety of possible rules to use. For an initial exploration, try `(4, 5, 6/4)`. That is, a cell will remain alive if it has four, five, or six neighbours and a dead cell will be born if it has exactly 4 neighbours. We will use `1` to represent a living cell and `0` to represent a dead cell. The calculation for the local rule is based on multiplying the 3×3×3 Boolean array of cells by a 3×3×3 mask array. If the mask array has a 27 in the centre and ones everywhere else, then the cell will be alive in the next generation if the sum of the products is 4,

27+4, 27+5, or 27+6. So the local life rule will use `e.` to check if the sum of the products is in the list `4 31 32 33` as illustrated below.

```
   ]L=:3 3$(3 3$1),(1,1 27 1,:1),(3 3$1)  NB. the mask for local life
1  1  1
1  1  1
1  1  1

1  1  1
1  27 1
1  1  1

1  1  1
1  1  1
1  1  1

   llife3d=: +/@:,@(L&*) e. 4 31 32 33"_  NB. local life for a 3D grid
```

Consider a neighbourhood that has a live centre cell with four live neighbours:

```
    ]n0=: (i.3 3 3)e.1 2 10 13 25
0 1 1
0 0 0
0 0 0

0 1 0
0 1 0
0 0 0

0 0 0
0 0 0
0 1 0

   llife3d n0            NB. local life on the neighbourhood
1
```

As expected, local life on neighbourhood `n0` yields a living centre cell for the next generation. Consider another neighbourhood, this time with a dead centre cell and only three live neighbours:

```
   ]n1=: (i.3 3 3)e.2 10 25
0 0 1
0 0 0
0 0 0

0 1 0
0 0 0
0 0 0

0 0 0
0 0 0
0 1 0

   1life3d n1          NB. local life on the neighbourhood
0
```

Local life on neighbourhood `n1` gives a dead centre cell for the next generation, since the cell is dead and does not have the right number of living neighbours for a birth under the (4,5,6/4) rule.

To apply the local rule to tesselations of the array, use `_3 cut`. Before doing this it is necessary to consider the borders of the array. We want all members of the cube to have 26 neighbours, so we will periodically extend the input array in three dimensions using the function `perext` from Section 6.3 of [4].

```
   perext=: {: , ] , {.

   perext3=: perext"1@:perext"2@:perext
   NB. 3-dimensional periodic extension

   life3d=: 3 3 3&(1life3d;._3)@ perext3

   life3d n0
0 1 1
0 0 0
0 0 0

0 1 0
0 1 0
0 0 0

0 0 0
0 0 0
0 1 0
```

Visualizing 3D Life

To produce a graphical representation of the 3D Game of Life we will use POV-Ray 3.6, a ray-tracing program from [3]. Some of the functions we will use are from the `povkit2.ijs` script in [4].

POV-Ray sets viewing parameters based on the idea of having a "camera" with certain orientations relative to the image.

```
  view_pars_sample=: 0 : 0  NB. view parameters; from povkit2.ijs
// set viewing parameters
camera{
  location <20,30,10>                NB. viewpoint
  angle 45                           NB. vertical viewing angle
  up <0,0,1>
  right <0,1,0>
  sky <0,0,1>                        NB. up direction
  look_at<0,0,0>                     NB. centre of interest
  }
//#default{finish{ambient 0.3}}
object{light_source{<200,100,50> color rgb<2,2,2>}}
background {color rgb<1,1,1>}
)
```

The individual cells will be represented as boxes, which can be defined with a utility from `povkit2`. The left argument is an RGB triple. The right argument is two diagonal corners of the box. We can utilise `fmtbox` as follows:

```
  view_pars_sample fwrite povpath,'green_box.pov'
  (0 1 0 fmtbox 1 1 1 2 2 2) fappends povpath,'green_box.pov'
```

The file `green_box.pov` now contains viewing parameters and code for a box:

```
// set viewing parameters
camera{
  location <20,30,10>
      angle 45
      up <0,0,1>
  right <0,1,0>
      sky <0,0,1>
    look_at<0,0,0>
  }
//#default{finish{ambient 0.3}}
object{light_source{<200,100,50> color rgb<2,2,2>}}
background {color rgb<1,1,1>}
object{box{<1,1,1>,<2,2,2>}pigment{rgb<0,1,0>}}
```

Opening `green_box.pov` with POV-Ray and clicking Run will render an image of a green box:

We can gather more information on our 3D Game of Life by colouring the boxes according to the number of neighbours.

To do this, we create unique RGB triples:

```
ct=.1,(],-.,0&*)=i.3
colors=.ct,(0.75*ct),(0.5*ct),(0.25*ct)
```

`colors` is a list of 28 triples with each co-ordinate of each triple in the range from `0` to `1`. Each triple corresponds to the number of live cells that could be in a neighbourhood (from `0` to `27`). The case where there are zero live cells corresponds to the triple `(1,1,1)`, which is white. When zero live cells are in a neighbourhood no cubes will be created, so this colour will never be used, which is desirable since we are rendering cells on top of a white background.

We will need to count the cells in the 3×3×3 neighbourhood of every live cell to determine the colour of the cell. To do this we use:

```
neigh3d=: * 3 3 3"_ +/@,;._3 perext3
```

Writing to the POV-Ray file requires `fwrite` and `fappends`, which are defined in `files.ijs`.

We are now ready to create a verb to make the POV-Ray file. This verb will store 100 iterations of 3D Life in a single file called `life3d.pov` which will be placed in the directory specified by the `povpath`. Some understanding of POV-Ray language directives is helpful. POV-Ray uses the float identifier `clock` to control animations. The value of `clock` ranges from `0` for the start frame to `1` for the end frame, with the other frames evenly spaced to values between `0` and `1`. Since we want our frames to be rendered in a specific order, we can use the clock value along with the fact that `0` is false and any non-zero value is true for the `#if` directive in POV-Ray to define the order of frame rendering.

```
  life3d_pov=: 3 : 0
k=.0                                    NB. index for the for loop
dx=.0.8                                 NB. side length of a cube
pfn=: povpath,'life3d.pov'              NB. the output file
ct=.1,(],-.,0&*)=i.3                    NB. 7 different RGB triples
NB. 28 different RGB triples
colors=.ct,(0.75*ct),(0.5*ct),(0.25*ct)
pos=.{<@i."0 $ y.                       NB. boxed list of xyz coords in the
array
view_pars_sample fwrite pfn             NB. some view parameters
'#declare I=-1;' fappends pfn
for_k. i. 100 do.                       NB. does 100 itns of life3d
  '#declare I = I + 1;' fappends pfn    NB. each itn has an index

  NB. POV-Ray will choose which iteration
  NB. to create based on the index
  '#if(I-100*clock)#else' fappends pfn

  mask=.,y.                             NB. mask for the array
  c=.(mask#,neigh3d y.){colors          NB. gets colors for each cell
according to
                                        NB. number of neighbours

  xyz=.>mask#,pos                       NB. positions of live cells
  (c fmtbox xyz,"1 xyz+dx) fappends pfn   NB. draw the boxes
  y.=. life3d y.                        NB. life3d on input array
  '#end' fappends pfn
end.
)
```

We can create the `*.pov` file by running `life3d_pov` on a random Boolean array. Don't forget to initialize `povpath`.

```
rand=: 1=?10 10 10$3
life3d_pov rand
```

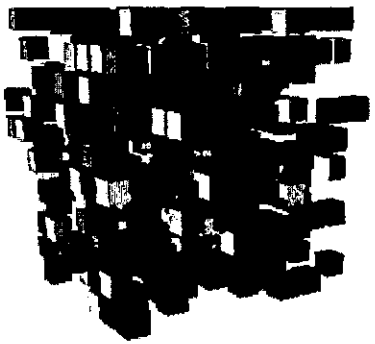
In order to create a sequence of images of the configurations in `life3d.pov` we need to create a `*.ini` file.

```
  life3d_ini=: 0 : 0
Initial_Frame=0
Final_Frame=100
Subset_Start_Frame=0
Subset_End_Frame=99
Input_File_Name=life3d.pov

Cyclic_animation=Off
Pause_when_Done=Off
Output_File_Type=S

Sampling_Method=1
Antialias_Threshold=0.5
Antialias_Depth=2
Jitter=Off
Test_Abort_Count=100

  life3d_ini fwrite povpath,'life3d.ini'
```

Running the `*.ini` file creates 100 frames numbered `0` to `99`. With the help of the Image3 addon we can assemble these into a QuickTime movie as follows:

```
load '~addons/image3/movie3.ijs'
$fns=:'life3d*.png' files_in povpath
4 fseq_to_png_mov fns; povpath,'life3d.mov'
```

Experimenting with different initial setups can give interesting results.

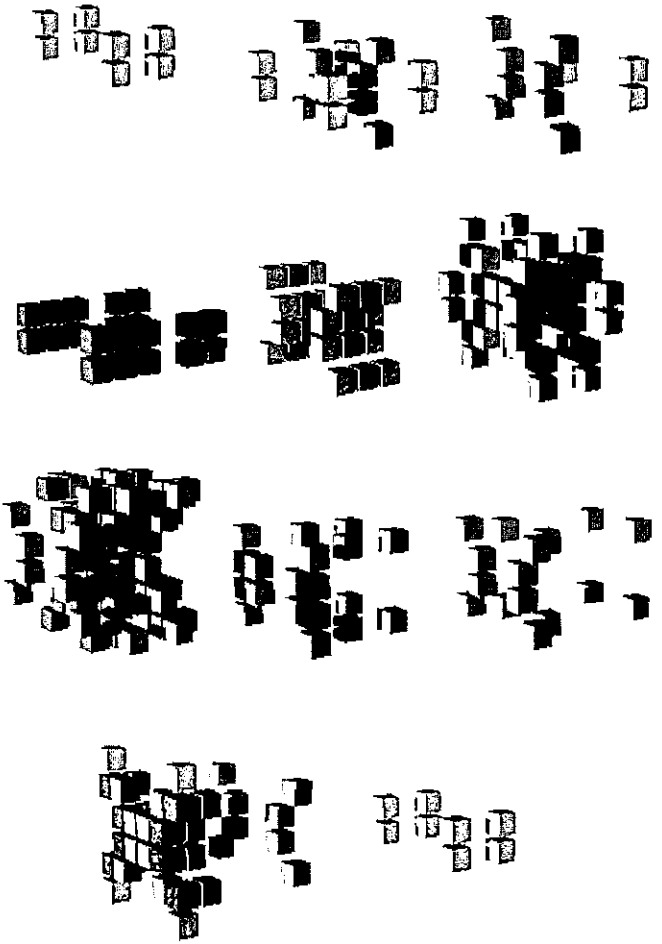


A random 10×10×10 configuration:
`rand=: 1=?10 10 10$i.3`

The same configuration
after 100 iterations

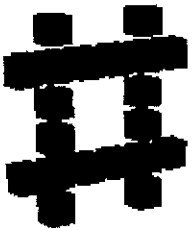
This configuration is off-centre in a 6×6×6 array:

This will form a periodic pattern that repeats every ten iterations:



A similar configuration centred in a 6×6×7 array:

will yield this stable formation:



QuickTime movies of the evolution of these configurations are available from [6].

The `life3d` function can easily be altered to experiment with different rules. As in the original Game of Life, it is possible to develop a variety of stable patterns and repeating cycles. J and POV-Ray allow easy study of these situations.

References

[1] Paul Callahan, "What is the Game of Life?", math.com, Online, 8 Dec 2005.

[2] Cellular Automaton, Wikipedia, Online, 8 Dec 2005.

[3] POV-Ray – The Persistence of Vision Raytracer, *http://www.povray.org*

[4] C.A. Reiter, *Fractals, Visualization and J*, Autumn 2005 update to 2nd Edition, *http://ww2.lafayette.edu/~reiterc/j/fvj2/index.html*

[5] Cliff Reiter, "Time(r) for the Game of Life", *Vector*, 21:3 (2005) 88-98.

[6] T. Zirkel, Auxiliary Materials for 3D Cellular Automata and the Game of Life, *http://ww2.lafayette.edu/~reiterc/mvq/zirkel/index.html*

Cumulative Normal Distributions Black-Scholes and Error Functions

by Ralph Selfridge (selfrid@cns.ufl.edu)

Recent comments in the J Forum about normal probabilities and distributions have led to a revival of interest in Black-Scholes algorithms. Black-Scholes is a set of equations that provide premiums for 'call' and 'put' options, and there has been an interest in providing appropriate algorithms in J. That, in turn, opens the question of an algorithm that will run in APL. This study was an attempt to create such an algorithm, which may already exist in some APL library but was not available to this author. We use APL2 as the APL medium. A serendipitous result is also a speed-up in computing cumulative normal distributions and an error function for APL.

We start here with Eugene McDonnell's article "At Play with J: Beware Scholes" [1], which provides Black-Scholes equations in reasonable mathematical form and then considers some algorithms in J. This article has several algorithms created, among others, by Hu Zhe and Oleg Kobchenko. McDonnell uses the `cnd` of Zhe and his own verb `bs` for the algorithm of page 141, which we label A1, followed on page 142 by an algorithm, which we label A2, using a formula from Ewart Shaw. All algorithms, using essentially the same verb `bs`, are below in one of two appendices.

We know that A2 is much faster in execution than A1, but it has the problem that it uses `H.` (an included verb in J) not available in APL2. Thus we must create a replacement for `H.`, or go back to A1 and use the `cnd` verb from Zhe. This relies only on a verb `normalprob` from `statdist` which can be converted so that we have Black-Scholes in APL2.

However if we examine `cnd` in A1 there is the noun `__` (negative infinity), which results in a domain error if we replace `__` with a large negative number for use in APL2. That, in turn, forces examination of `normalprob` to find, and work around, `__`. With that covered we have a working Black-Scholes in APL2. Since fairly simple APL is used these algorithms should all work in any APL.

The conversion from the verb `cnd` (J) to the function `CND` (APL) allows for considerable simplification, and even further if `CND` is only for use in Black-Scholes. We make these changes, and any other improvements we can think of, and obtain `CND` and `BS` in Appendix 2. Now if we consider A2 we have a formula

for the error function but that in turn allows us to come back from CND to get an easy APL function for the error function of a variable (it is entirely possible that such a function exists in an APL library elsewhere). We add this function to Appendix 2.

Now if we consider the changes for APL2, which shorten CND considerably, we can ask if similar changes can be made in CND of A1. These changes produce A3 in Appendix 1.

S = Current stock price
X = Option strike price
R = Risk-free interest rate
T = time in years until strike date
V = volatility, or standard deviation of asset price

All four algorithms are entered with `BS S,X,T,R,V` and return two numbers: the first is the 'call' premium, the second the 'put' premium, with an example

```
      BS 60 65 0.25 0.08 0.3
2.13337  5.84628
```

We need to compare execution times for the differing algorithms, both for BS and CND/ERF. In the case of J we use

```
(10000)6!:2'BS 60 65 0.25 0.08 0.3'
```

For APL we must create a timing function that runs 10000 executions of the desired function. In the following table we set w=1.2 and ww a vector of length 500 of numbers in the range 0 to 5 ((i.500)%100). All results are normalized so that A2 values are 100.

	BS yc	CND w	CND ww	ERF w	ERF ww
A1	288	1490	1514		
A2	100	100	100	100	100
A3	87	154	4.8	289	13.9
APL	75	130	27	230	69

When comparing results numeric output across the algorithms agree to several significant places. Since we must use system clocks, without knowing precisely

how they run (nor do we know the resolution of the clocks), we use a repetition of 10000. Even so, timing comparisons should not be taken too seriously.

We find that A2 and A3 are substantially the same for Black-Scholes, and the benefit of avoiding H. is debatable. It is also interesting that APL appears faster for Black-Scholes. Both CND and ERF are faster in A2 for a scalar argument but considerably slower as the argument length grows; the fact that BS has an internal 2 by 2 is probably the explanation for speed-ups in A3 and APL. In the case of APL, avoiding H. becomes crucial, unless an H function is written either in APL (very slow) or some better choice, say Fortran and a DLL. There is little choice in APL, but it would appear A3 is better in J, unless the arguments for cnd and erf are scalars.

If we create a Fortran subroutine to replace H. specifically for Black-Scholes, and hook across to APL2 by way of a DLL the resulting timing for BS is slightly slower than the given APL2 version (though much faster than using a Fortran version of H. with the attendant use of H"0).

References

[1] Eugene McDonnell, "At Play with J: Beware Scholes" *Vector* 19.3, pp137-142

Appendix 1

```
A1
load 'statdist'
cnd=: 3 : 'normalprob 0 1 __, y' "0       NB. Hu Zhe
BS=: 3 : 0                                NB. Eugene McDonnell
'S X T R V'=.y
d=. ((^.S%X) + T*R(+,-)-:*:V)%V*%:T
|(S,X*^-R*T)-/ .*cnd d*/1 _1
)

A2
erf=:1 H. 1.5@*: * 2p_0.5 &* %^@:*:       NB. Ewart Shaw
cnd=: -:@ >:@erf %&(%:2)
BS=: 3 : 0
'S X T R V'=.y
d=. ((^.S%X) + T*R(+,-)-:*:V)%V*%:T
|(S,X*^-R*T)-/ .*cnd d*/1 _1
)
```

```
A3  (modified A1)
cnd=: 3 : 0
b=. 0 0.31938153 _0.356563782 1.781477937 _1.821255978 1.330274429
1-|(0.398942* ^_0.5**:y)* b p. %>:y*0.2316419
)
BS=: 3 : 0
'S X T R V'=.y
d=. ((^.S%X) + T*R(+,-)-:*:V)%V*%:T
|(S,X*^-R*T)-/ .*cnd d*/1 _1
)
```

Appendix 2

CND provides for cumulative normal distributions, but only for use in Black-Scholes. It can be modified for any shape argument by simple changes in line 2 (made, of necessity, for timing comparisons). ERF returns the error function value for its argument, and must also be modified for timing comparisons (again in line 2) or multiple arguments.

```
    ∇ Z←CND Y
[1]   Z←6 1⍴1.330274 ¯1.821256 1.78148 ¯0.356565 0.31938153 0
[2]   Z←1-|(0.398942××¯0.5×Y*2)×2 2⍴(4 1⍴÷1+Y×0.2316419)⊥Z
    ∇

    ∇ Z←BS Y;S;X;T;R;V
[1]   (S X T R V)←Y
[2]   Z←|(S,X××-R×T)-.×CND(((⍟S÷X)+T×R+Z,-Z←0.5×V*2)÷V×T×0.5)∘.×1 ¯1
    ∇

    ∇ Z←ERF Y
[1]   Z←6 1⍴1.330274 ¯1.821256 1.78148 ¯0.356565 0.31938153 0
[2]   Z←1-2×|(0.398942××-Y*2)×(÷1+Y×0.3275911166)⊥Z
    ∇
```

(Readers might prefer the following slightly shorter version in Dyalog, using the direct-definition form implemented by John Scholes. *Ed.*)

```
      CONS←6 1⍴1.330274 ¯1.821256 1.78148 ¯0.356565 0.31938153 0
      CND←CONS⍎{1-|(0.398942××¯0.5×⍵*2)×2 2⍴(4 1⍴÷1+⍵×0.2316419)⊥⍺}
      ERF←CONS⍎{1-2×|(0.398942××-Y*2)×(÷1+⍵×0.3275911166)⊥⍺}

    ⍎ Z←BS (S X T R V)
[1]   Z←|(S,X××-R×T)-.×CND(((⍟S÷X)+T×R+1 ¯1×0.5×V*2)÷V×T×0.5)∘.×1 ¯1
    ⍎
```

Digitalising the Vector Archive

by Ian Clark (earthspot2000@hotmail.com)

Two years ago the decision was taken to make all future articles available on the *Vector* website (*www.vector.org.uk*). It was a natural decision to convert back-issue articles to the same format. The *Vector* production team reached the conclusion that HTML was a good definitive format for maintaining documents which needed to be published in a variety of forms (e.g. on the Web and in printed version). This set the pattern for the web-published archive.

It was considered that HTML versions of articles to be published on the *Vector* website ought to include minimal mark-up, consisting of no more than what is necessary to identify different sections of text, to be handled in different ways (e.g. narrative text, APL code, tables and placement of diagrams). As many options as possible ought to be deferred for controlling the appearance of the article on the screen.

This is a deceptively straightforward requirement. Most of it can be delivered by using a cascaded style sheet (CSS file), but it is also important to avoid browser features which act in such a way as to reduce our options in the future. It takes a lot of experience to know what these features are, plus the need to debate what represents good practice. Personal favourite constructs have to be sacrificed on the altar of compliance to the HTML 4 standard recommendation [1], because at the time of writing this is considered the way to go for future XML migration.

The History of Vector Production

Camera copy for *Vector* was originally pasted up from panels of printed text and art-work produced in a variety of ways. Gradually page layouts became computer generated in increasingly standardised ways. By volume 4.2 (October 1987), substantially complete camera copy is extant in the form of Microsoft Word files. This is the sort of material to be found in the APL Madrid CD [2], of which more below.

Nowadays camera copy takes the form of one or more PDF files, which are what is delivered to the printer. Assuming we have done our job properly, it would in principle be possible to republish back editions of *Vector* by regenerating PDF files from the web archive, to the extent of course that all the articles are there. Which they aren't, and probably never will be, because much of *Vector* consists of ephemera. However what defines ephemera is a debatable question and nobody has debated it yet.

The author considers that there are many fine editorials going back to Vol 1.1 (1984) which can still be read with pleasure and profit – indeed whose content is extremely provocative in the light of subsequent developments. On the other hand when choosing which articles to varch next, i.e. to process for publication in the archive, some sort of priority asserts itself: articles of timeless application coming before articles focussing on obsolete technology.

The APL Madrid CD

The APL Madrid CD was the first serious attempt to publish the whole *Vector* Archive for the benefit of members at large. The CD was produced in quantity and given to all delegates at APL Madrid 2002. The material on it however is incomplete, conforms to various different standards and is of variable quality. A variety of different APL fonts are in use (I-APL, APL2, APL*PLUS, Dyalog APL, etc), these all having different ⎕AV layouts. The DOC files which comprise the bulk of the material on the CD require a range of obsolete versions of Microsoft Word in order to display them, and even then this cannot be relied upon to work correctly (see below). It is only when you attempt to do so that you realise how badly Microsoft Word supports upward-compatibility, or historically has done so since 1984. Diagrams and mathematical formulae do not reproduce reliably. If you install a series of obsolete versions of Microsoft Word in order to display these files, you discover that these software antiques no longer perform on modern machines in quite the same way as they did when they were first released, not least because the author of a typical article dealing with APL has until very recently been pushing his word-processor to the limit. The result might be considered (by the vendor) good enough to import an old document for editing and re-issue, but in the author's opinion it is not dependable enough to view a published document.

It soon becomes clear that to publish even the content of the APL Madrid CD on the Web is a shade more complicated than what the vendor would like you to believe, i.e. that you merely have to feed each DOC file into the latest release of Microsoft Word and select menu: "Save as Web Page...". Migrating each paper in the archive to the web is an act of creative originality and support for the task cannot be purchased off-the-peg. Indeed the task has to be properly designed if it is to be consistent and economical to perform.

Task Design Strategy

The following task design strategy has been arrived at by trial and error:

1. Extract the plain text of the narrative from the original DOC file (or whatever format the source file is in).

2. Insert appropriate HTML mark-up, in such a way as to repeat the action in a different way without having to repeat all the manual effort.

3. Scan anything "awkward" (i.e. awkward to reproduce in HTML) from the original printed page, in the form of a JPEG or GIF

4. Generate a test page and proof-read it, repeating the previous steps until the page looks satisfactory.

5. Upload it to the correct folder on the *Vector* server.

Fortunately Microsoft Word doesn't make step 1 too difficult: if you import a DOC file into an APL variable you see a clearly discernible header and trailer which can be lopped to yield more-or-less acceptable plain-text.

Microsoft Word also helps with step 3 in certain limited but useful cases, especially where mathematical notation has been used. As mentioned, Word 2000 offers the option to save a DOC file "as a Web Page". The code generated as a result is reminiscent of the early days of third-generation language compilers before optimisation came along. It is far from lean and mean, the vendors' developers having felt obliged to emulate the most piffling features of the original document. However, where the source document defeats even their inspired ingenuity, Word generates a neat GIF which you can pick out and use in place of scanned artwork. This includes all mathematical notation.

Handling APL code

Another place where Microsoft Word is relatively obliging is in the handling of APL code sections. All the information is there (usually) to restore the original code – though of course the characters which first appear bear little resemblance to the ones intended. But the APL primitives are generally 1-1 and so can be handled by a `⎕AV` conversion table. It is merely a question of deciding which table to use. Provided of course someone can deliver you a usable table in the first place. (They can't).

A crude approach is found to work: simply build up a collection of tables as you go along. There must be 10 or so different layouts to be found in the APL Madrid CD, of which two predominate. So the task of building the table by eye gets easier and easier: for each fresh article you get to recognise the code layout required, and when you apply it, progressively more characters are coded correctly. The VARCH workspace (described below) warns you of APL characters hitherto unencountered in the chosen conversion table and invites you to tell it what APL characters they are meant to be.

The Joys of Microsoft Word

In all other cases Microsoft Word is the enemy, to be confronted, outmanoeuvred and finally defeated. One irritating trick it has is to force a newline inside a block of specially formatted code by means of a character indistinguishable from APL: ⍝. Another trick (now thankfully obsolete) is the way in which Word once represented a non-standard character. Unlike the handling of italics and other such text formatting, which gets stripped out by the lopping process described above, a complex inline mark-up construct was used. The VARCH workspace user (the *varcher*) must recognise this by eye and replace it by the corresponding VARCH construct.

For example: how the paper loads into VARCH:

```
If ad ⍱symbol 186 \f "Symbol" \s 13ùò 1 mod n then exit
```

...how the VARCH regeneration function `sullivan123_70` converts it:

```
<p>If <i>a<sup>d</sup></i> &#x2261; 1 mod <i>n</i> then exit</p>
```

...the end appearance in Firefox (faithfully mirroring the hardcopy back-edition, p72 [3]):

If $a^d \bullet 1 \bmod n$ then exit

Old papers employing mathematical notation don't always display under later versions of Microsoft Word as intended. Here's how Word 2000 corrupts some formulae in a DOC file dated 1994 from the APL Madrid CD:

Advantages and Disadvantages of the VARCH Approach

The advantages of the above task design strategy, or rather of the VARCH support for it, are:

1. There is no need to depend on Microsoft Word to do correctly what it claims to do, which is a major worry off one's mind, not to mention a major time-waster circumvented.

2. Multifarious obsolete standards for representing APL fonts in print are replaced by one single standard based on Unicode. "One font to rule them all", as Adrian Smith has put it [4]. A useful by-product is that any code sample in the whole archive can be copied and pasted into the session of any modern APL, yielding identical behaviour (which is hopefully the one you want). You could say that APL has at last reached the happy state which ASCII-based languages have been in since the 1980s.

3. The mark-up process for generating and regenerating an article can be done in any order. Successful editing steps don't have to be repeated.

4. An article can be regenerated one-touch with a different basic template, with additional mark-up, or a different ⎕AV layout (or a corrected one).

5. Work done on the VARCH system to enable it to handle a given article benefits the processing of all subsequent articles.

6. There are no intermediate versions of articles stored anywhere, nor any intermediate cribs or tables.

VARCH takes the source material exactly as it appears on the APL Madrid CD and converts it to HTML in the currently approved way, storing the output of the manual mark-up task as a single APL function called a `paperfn`. The information a `paperfn` contains is an abstract description of the source text: it does not assume that any particular given HTML construct is going to be used. The actual HTML that gets generated is determined by the version of the VARCH workspace used to execute the `paperfn`.

Example of a `paperfn: langlet62_23`

The following is a short but sweet example of a `paperfn`, that of an archive article by the late Gérard Langlet [5]:

```
    ∇ langlet62_23;selection;REPLAY
[1]   ⍝∇paper: 430 created: 18 December 2006, 22:44 using 1 VARCH44
[2]   ensure
[3]   fetch myname
[4]   ⍝AUTHOR←'G&eacute;rard Langlet'
[5]   AUTHOR←'Gérard Langlet'
[6]   ⍝PTITLE←APL "RISC Programming Style"'
[7]   PTITLE←APL ',2 qu 'RISC Programming Style'
[8]   CODETYPE←1
[9]   REPLAY←1
[10]
[11]   slx 536 3     ⋄ is_code 1       ⍝ ⎕IO
[12]   slx 1492 3    ⋄ is_code 1       ⍝ ⎕SS
[13]   slx 2610 3    ⋄ is_code 1       ⍝ ⎕IO
[14]
[15]   slx 1228 3    ⋄ is_code 1       ⍝ 100=⍴X
[16]   slx 1873 2    ⋄ is_code 1       ⍝ ^⌿
[17]   slx 1877 2    ⋄ is_code 1       ⍝ +⌿
[18]
[19]   slx 0 0       ⋄ is_para         ⍝ In general
[20]   slx 333 0     ⋄ is_code 0       ⍝       ∇RÉC
[21]   slx 473 0     ⋄ is_para         ⍝ It works p
[22]   slx 650 0     ⋄ is_code 0       ⍝       ∇RÉC
[23]   slx 765 0     ⋄ is_code 0       ⍝       ⎕NSI
[24]   slx 868 0     ⋄ is_para         ⍝ COUNTALLV
[25]   slx 1236 0    ⋄ is_para         ⍝ Why use go
[26]   slx 1356 0    ⋄ is_para         ⍝ I have wri
[27]   slx 2156 0    ⋄ is_para         ⍝ The "RISC"
[28]   slx 2196 245 ⋄ is_list 'a'      ⍝  a) Simple a...
[29]   slx 2441 0    ⋄ is_para         ⍝ I even str
[30]   slx 3068 0    ⋄ is_para         ⍝ P.S. A com
[31]   slx 3500 0    ⋄ is_code 0       ⍝ (If youc
[32]   slx 3597 0    ⋄ is_para         ⍝ It might b
[33]  substws
[34]  proc 1 ⍝--use the appropriate variant
[35]  writeout
[36]  see
    ∇
```

Note that VARCH generates this APL fn after the first editing session of the article concerned. Subsequent sessions generate additional work lines in the session log, but do not tinker with the `paperfn` itself. This is left to the varcher to do, by copy/paste from the session log. The `paperfn` is not hard to hand-edit.

Notes on the listing

Let's go briefly down the listing, commenting on the code highlights:

```
    ∇ langlet62_263;selection;REPLAY
```

This function, when executed, will regenerate the paper by Langlet, Vol 6.2, page 23. VARCH uses a standing global: `INDEX` to get the author's name plus title, and

the existence of a varched paper and its Madrid source-file are written back into INDEX, which therefore serves as a work-schedule.

```
[2]   ensure
```

This fn checks whether Init has been run and if not runs it. Init sets up globals containing frequently used constants, especially paths to work folders. The varcher edits Init on installation to provide his/her own folder names, then forgets it.

```
[4]   ⍝AUTHOR←'G&eacute;rard Langlet'
[5]   AUTHOR←'Gérard Langlet'
```

Heritage code is left in-place for forensic purposes. In this case earlier versions of VARCH did not handle e-acute correctly if it was the APL+Win character (ASCII: 130) and not the Unicode one: #233. Now it does. (It also handles APL characters in titles, e-acute being here an honorary APL character, or more correctly a character from the atomic vector of APL+Win.)

Commented-out line 4 was a fudge to force the browser to employ the so-called HTML entity: é. This HTML feature is recognised by both Microsoft Internet Explorer (IE) and Mozilla Firefox, but maybe it's one of those features best avoided. It does at least say clearly what it is when you come across it, which é or é don't. In the working code proper, VARCH doesn't use specifications which are entangled with how they are implemented.

```
[6]   ⍝PTITLE←APL "RISC Programming Style"'
[7]   PTITLE←APL ',2 qu 'RISC Programming Style'
```

A similar consideration applies to the use of quotes in titles. Commented-out line 6 employed dumb-quotes, and since these too are honorary APL characters, VARCH doesn't presume to smarten them up. However the tool-function qu surrounds a string with smart-quotes for embedding in HTML. Subsequent versions of VARCH are at liberty to implement smart quotes however they want (including quotes in titles), by altering the implementation of qu, which governs quotes throughout VARCH. This is an example of how VARCH typically defers a decision.

```
[8]   CODETYPE←1
```

This controls the behaviour of function coded, which generates embedded HTML for all types of code, whether J or a flavour of APL. Global CODETYPE controls a :Select//:EndSelect block inside coded. The default value is 1, so line 8 is redundant. It is generated nonetheless because you might want to finesse the handling of code in this function at some future date. The experience of VARCH

is that each new back-issue to be varched shows what you thought was the standard treatment to be the exception rather than the rule.

```
[9]   REPLAY←1
```

The behaviour of VARCH fns needs to differ when the given article is first edited by hand (`REPLAY←0`) and subsequently regenerated (`REPLAY←1`). Some fns are only valid on replay. In particular when `REPLAY←0` the function `selection` gets data from the editing panel using

```
PANEL ⎕wi 'selection'
```

whereas when `REPLAY←1` the editing panel isn't there and instead `selection` becomes a localised variable assigned by `slx` (see below)

```
[11]   slx 536 3    ⋄ is_code 1      A ⎕IO
```

This is the first *work line*. All previous lines are generated from a function template and differ little between paperfns. By dragging the cursor, the varcher has selected 3 chars of text in the editing panel starting at character `536` (viz. `⎕IO`) and pressed the button (or selected the menu) to run function `is_code 1`. (Incidentally all interactions with the editing panel are equivalent to entering some `htmfn` in the session log.) This action not only marks up `⎕IO` with the HTML construct to do the trick, but also generates a work line which, when re-executed at `REPLAY←1` time, will repeat the original editing action. Notice however that the work line is careful not to prejudice the actual HTML mark-up originally used, or to be used in the future. In fact (now line 4 has been superseded) there's no literal HTML mark-up in the entire `paperfn`.

```
[19]   slx 0 0      ⋄ is_para        A In general
[20]   slx 333 0    ⋄ is_code 0      A       ∇RÉC
[21]   slx 473 0    ⋄ is_para        A It works p
```

The argument of function `slx` is called a selection. Its form is always an integer 2-vec (`start len`), being determined by the GUI interface of APL+Win. The GUI numbers the first char in an Edit control as `0` (whatever the setting of `⎕IO`). If you either select it or place the cursor in front of it, the result is a selection commencing `0` (i.e. with `start=0`), as exemplified by line 19.

If the second number (`len`) is `0`, this means the cursor is a winking line and not a smeared-out strip. However, by convention, VARCH recognises this as a request to seek the next newline character (ASCII: `13`) and take that as the span of the selection. So the varcher can specify a paragraph by simply placing the cursor at the start of the line and running function `is_para`. A logical paragraph invariably starts a line in the edit window, but the converse is not always true.

This `len=0` trick also works with most blocks of code you encounter. Function `is_code 0` designates a pre-formatted block of code, generally to be marked-up: `<pre class="aplu">...</pre>`, whereas `is_code 1` designates a string of in-line code, to be marked-up (e.g.) thus: `<tt class="aplu">...</tt>`.

As a visual cue VARCH lifts the selected text at generating time and appends the first few characters as a trailing comment to the work line, white-spacing anything non-legible. This helps a lot when hand-editing the `paperfn` (not to mention debugging VARCH!). So, for instance, if the `htmfn` you called at generation time was the wrong one to use, or you need to craft a new one as a variant of some existing one, then you can simply overtype the fn name in the work line without needing to bring up the editing panel again. In fact as a varcher, faced with a section I don't know how to handle, I often find myself clicking the button "placeholder" (to run `is_placeholder`). This is a no-operation in the editing panel but generates a work line I can subsequently hand-edit.

Notice too that the `start`-arguments of `slx` do not need to ascend. The work lines happen to be grouped into three sections, representing the three separate edit sessions which were needed before the HTML generated satisfactorily. However (unless selection spans overlap) the order of execution of work lines is immaterial.

```
[33]  substws
[34]  proc 1 A--use the appropriate variant
```

The global: `TEMPLATE` is read from a given HTML file, which can be adjusted standalone to give the right appearance under both IE and Firefox. The current `TEMPLATE` uses the same CSS (cascading style sheet) as the most recent papers in the archive, hence changes to this CSS should alter the appearance of all archived articles in step. Function `substws` sets `PAGE←TEMPLATE` and replaces the tags: `{WS}`, `{VERSION}`, `{WHEN}`, `{PTITLE}`, `{AUTHOR}`, etc.

Function `proc` is largely heritage, there once being a supposed need for custom pre/post-processing. It still replaces special characters globally with the appropriate HTML entity. Also if the article contains frequent references to a given APL identifier (such as `vx` above) it can apply mark-up to these words wherever they occur, provided it is not inside designated code. As the final operation before writing to disk, `proc` does not need to maintain `ORIG`, which makes it somewhat easier to implement.

```
[35]  writeout
[36]  see
```

These fns write PAGE as a HTML file to the correct folder in the local website image and call the browser to show the latest HTML file generated.

Text Selection and Mark-up

The hand-editing task is one of selecting sections of text and specifying how they are to be marked-up. As already remarked, editing steps can be carried out in any order and the work lines executed in any order. That's because the selection in the argument of `slx` is in *orig* numbering, i.e. it is the selection you'd see if this was the first editing step to be performed.

VARCH converts actual selections at hand-editing time to orig selections in generated work lines, even though each deletion, insertion and mark-up operation shifts all the subsequent characters. On replay, the actual current selection is reproduced, however the situation stands.

The way VARCH does this is to set up a global `ORIG←⍳⍴VX` and maintain it in-step with `VX`, the buffer of marked-up text. This is sheer Homer Simpson programming and I'm embarrassed not to have developed a reliable `orig` conversion fn which works from a history of edit selections. But it's a rock-hard implementation and I'm loath to replace it: failure of the `orig` function potentially wastes hours of varching work (not to mention embroiling you in hours of debugging) because the HTML page will then fail to regenerate properly from the `paperfn`. It goes without saying that the first attempt at generating HTML never does quite what you hope it will – or it does different things in IE and Firefox!

At each edit step using the editing panel, the appropriate `htmfn` is called and causes mark-up tags to appear in the edit window. This mark-up is however purely illustrative, since the sole purpose of the task is to generate a `paperfn`, since it is only the execution of a `paperfn` that saves the HTML file of a varched paper. As stated earlier, it is a design objective of VARCH to see that no explicit mark-up creeps into the `paperfn` itself. All mark-up is governed by the set of htmfns.

These include:

```
is_APL          is_bullets      is_numlist      is_subscript
is_BLOB         is_c            is_omega        is_superscript
is_Eacute       is_caption      is_para         is_symb
is_J            is_code         is_para_b       is_tab
is_OBLOB        is_dash         is_para_n       is_table
is_addr         is_eacute       is_para_q       is_tabspec
is_addrSP       is_entity       is_placeholder  is_tagged
is_aelig        is_fig          is_refs         is_txt
is_alpha        is_figure       is_rule         is_uml
is_block        is_italic       is_safelytagged is_verse
is_bold         is_last_action  is_short_caption
is_boxed        is_list         is_special
is_break        is_mxtab        is_subpara
```

They have tended to proliferate. Many are aliases of each other or straightforward variants. The reason is that it has been deemed safer to write a new `htmfn` where there is no clear existing one rather than go retrospectively generalising them, with the possible consequence that an old `paperfn` will no longer regenerate correctly. Also (in the case of aliases) a distinctive name holds out the possibility of a potentially different treatment in the future.

For example: `is_refs` currently runs `is_list 1`. However we may in time want a block of text spanned by `is_refs` to be marked-up as a table (HTML: `<table>...</table>`), allowing for finer control over its appearance than the current crude numbered list provides.

The overriding consideration governing htmfns is that they should specify the (varcher's) intention, not the (current) implementation.

The Current State of VARCH

There are many valuable papers hidden in back-issues of *Vector*. Each time I encounter one, it stiffens my resolve to see that the archive gets substantially, if not wholly, Web-published before I kick the bucket. Some 90% (around 950 articles) of the *Vector* archive as listed in `INDEX` remains to be varched. At this rate it will take me 10 years, but I must confess it hasn't been my sole activity during 2006, nor at times my top priority. However it's too much for one man and we need volunteers.

VARCH has been well-honed for the tasks it handles, so productivity cannot be improved much for an experienced varcher. However there are a number of unskilled, time-consuming tasks for which volunteers will speed the process:

- Identifying figures, or "awkward" tabulations, and scanning them as JPEGs

- Hand-marking the hardcopy originals of *Vector* to identify in-line code, identifiers and italic text
- Proof-reading draft HTML and reporting errors

The latest version of the VARCH workspace is available for download [6]. If you have the APL Madrid CD, some or all back-copies of *Vector*, and can run an APL+Win 3.6 workspace, then you can varch a few articles yourself, maybe in time becoming one of the anonymous yet blessed copyists of the sacred texts underpinning every major world religion 5,000 years hence. On a careful reading of history that's no joke. Contact the author, or the editor of *Vector*.

References

[1] W3C, *HTML 4.01 Specification*, http://www.w3.org/TR/REC-html40/

[2] British APL Association, The APL Madrid CD, APL2002, Madrid

[3] John Sullivan, "Multiprecision Arithmetic – Part III" *Vector* 12.3, 70

[4] Adrian Smith, "One Font to Rule Them All", *Vector* 11.2, 105

[5] Gérard Langlet, "APL 'RISC Programming Style'", *Vector* 6.2, 23

[6] *Vector* Archive Project, latest VARCH workspace

Subscribing to Vector

Your *Vector* subscription includes membership of the British APL Association, which is open to anyone interested in APL or related languages. The membership year runs from 1st May to 30th April. The British APL Association is a Specialist Group of the British Computer Society, Reg. Charity No. 292,786

Title: _____ Surname: _____

Other Names : _____

Home Address: _____

_____

Postcode/ Country: _____

Telephone: _____ Mobile: _____

Email Address: _____ Date of Birth: _____

UK private membership _____ £20 ❑

Overseas private membership _____ £22 ❑

Airmail supplement (outside Europe) _____ £4 ❑

Non-voting UK member (student/OAP/unemployed only) _____ £10 ❑

PAYMENT – in Sterling or by Visa/Mastercard/American Express or Switch
Payment should be enclosed with membership applications in the form of a UK Sterling cheque to "BCS", or you may quote your credit-card number. To pay by Direct Debit – please download the registration form from *www.vector.org.uk*.

I authorise you to debit my credit-card or Switch account:

_____ Expiry: ___ / ___ Start: ___ / ___

Name on card: _____ Issue number if applicable: _____

for the membership category indicated above,

❑ annually, at the prevailing rate, until further notice
❑ one year's subscription only

Cheque: I enclose a cheque for £ _____

Signature: _____ **Date:** _____

Please send your completed form and payment to:

Specialist Groups' Officer, BCS, 1st Floor, Block D, North Star House, North Star Avenue, Swindon, SN2 1FA, UK Fax: +44 (0) 1793-417-444.

Data Protection Act:
The information supplied may be stored on computer and processed in accordance with the registration of the British Computer Society.

I agree to this information being processed for administration by BCS

I agree to the above information being used to contact me regarding BCS or third party events, promotions, products and services
(Please delete as necessary)

The British APL Association

The British APL Association is a Specialist Group of the British Computer Society. It is administered by a Committee of officers who are elected by a postal ballot of Association members prior to the Annual General Meeting. Working groups are also established in areas such as activity planning and journal production. Offers of assistance and involvement with any Association matters are welcomed and should be addressed in the first instance to the Secretary.

Committee

Chairman	Paul Grosvenor 01293-562700 paul@optima-systems.co.uk	Optima House, Mill Court Spindle Way, CRAWLEY West Sussex, RH10 1TT
Secretary	Anthony Camacho 0117-9730036 acam@blueyonder.co.uk	11 Auburn Road Redland BRISTOL, BS6 6LS
Treasurer	Nicholas Small 01223-570850 treas.apl@bcs.org.uk	12 Cambridge Road, Waterbeach, Cambridge CB25 9NJ
Journal Editor	Stephen Taylor editor@vector.org.uk	81 South Hill Park LONDON NW3 2SS
Education	Dr Alan Mayer 01792-295779 a.d.mayer@swansea.ac.uk	European Business Management School, Swansea University, Singleton Park, SWANSEA SA2 8PP
Activities	Ray Cannon 01252-874697 ray_cannon@compuserve.com	21 Woodbridge Rd, Blackwater Camberley, Surrey GU17 0BS
Projects	Ian Clark earthspotty@googlemail.com	
Membership	Specialist Groups' Officer, BCS,	1st Floor, Block D, North Star House, North Star Avenue, Swindon, SN2 1FA
Enquiries	Nicholas Small 01223-570850 treas.apl@bcs.org.uk	12 Cambridge Road, Waterbeach, Cambridge CB25 9NJ

Journal Working Group

Editor:	Stephen Taylor	see above
Production:	Adrian & Gill Smith	Brook House, Gilling East, YORK (01439-788385)
Advertising:	Gill Smith	Brook House, Gilling East, YORK (01439-788385)

Support Team: Anthony & Sylvia Camacho (0117-9730036), Tomas Gustafsson
Ray Cannon (01252-874697), Stephen Taylor (077-1340 0852), Ian Clark

Typeset by APL-385 with MS Word

Printed in England by Short-Run Press Ltd, Exeter

VECTOR

VECTOR is the quarterly Journal of the British APL Association and is distributed to Association members in the UK and overseas. The British APL Association is a Specialist Group of the British Computer Society. APL stands for "A Programming Language" – an interactive computer language noted for its elegance, conciseness and fast development speed. It is supported on most mainframes, workstations and personal computers.

SUSTAINING MEMBERS

The Committee of the British APL Association wish to acknowledge the generous financial support of the following Association Sustaining Members. In many cases these organisations also provide manpower and administrative assistance to the Association at their own cost.