Contents

Quick reference diary		2
Editorial	Stephen Taylor	3
News		
FinnAPL In Tallinn 2009	Adrian Smith	5
Discover		
A newbie's discovery of APL	Rebecca Burriesci	14
Rediscovering APL's history	Catherine Lathwell	21
Array languages for Lisp programmers	Slobodan Blazeski	26
APLX interface to R statistics	Simon Marsden	33
Inner product - an old/new problem	Roger Hui	58
Learn		
Financial math in q 1: Graduation of mortality	Jan Karman	67
Generating combinations in J efficiently	R.E. Boss	75
Ten divisions to Easter	Ray Polivka	89
All but one	Norman Thomson	93
The Snap SALT command	Daniel Baronet	103
Coding for a shared world	A Reader	109
On average	Roger Hui	122
Profit		
Backgammon tools in J 1: Bearoff expected rolls	Howard A. Peelle	128
A genetic algorithm primer in APL	Romilly Cocking	135
Functional calculation DNA identification technology and APL	Charles Brenner	145
APL: The next generation	Ajay Askoolum	153
Subscribing to Vector		192

Quick reference diary

13-16 Sep	Berlin	APL2010 Conference
16-20 Oct	Minnowbrook, USA	APL Implementers Workshop
13-13 Sep	London	q training courses
27 Sep - 4 Oct	Singapore	q training courses
18-28 Oct	New York	q training courses
8-18 Nov	London	q training courses
8-20 Dec	New York	q training courses

Dates for future issues

Vector articles are now published online as soon as they are ready. Issues go to the printers at the end of each quarter – as near as we can manage!

If you have an idea for an article, or would like to place an advertisement in the printed issue, please write to editor@vector.org.uk.

EDITORIAL

It has been a long time since were last able to print *Vector*, so here is a fat issue. It has been a busy time.

Ian Clark and his collaborators in the J community have published the second edition of *At Play With J*. Where the first edition carefully collected Eugene McDonnell's *Vector* columns, the second edition revises all the examples for the J language as it is now, and so becomes an invaluable J textbook. Vector Books is proud to have published this.

Earlier this year Dyalog published Bernard Legrand's compendious *Mastering Dyalog APL*. At nearly 800 pages, generously illustrated, this is probably the best modern introduction to APL. This is a landmark publication that is doing much to make the language accessible to 21st-century programmers.

New faces appear, some of them drawn to Dyalog's worldwide programming contest. This year we congratulate Ryan Tarpine, from Browns University in the US for winning the Grand Prize, which includes entry and travel to APL2010 in Berlin next month.

Eventually we lose others. We are sad to record the loss of Donald McIntyre and Eugene McDonnell, both long-standing contributors. Donald, writes Roger Hui, "was an eminent geologist who pioneered the use of of computers in geology. He was a gifted and inspiring teacher, an early and long-time APL and J user, and a friend and colleague of Ken Iverson." Gene was "a family man, utterly decent, generous, kindhearted; also erudite and witty. His contributions to APL were graceful as well as useful." (Larry Breed) Before his *Vector* columns, Gene wrote for *APL Quote-Quad*. "In my youth, when I was just starting in APL, on receiving an issue of the *APL Quote-Quad* I would inevitably and eagerly first turn to Eugene McDonnell's 'Recreational APL' column. Through these columns I learned that it was possible for technical writing to be erudite, educational, and entertaining, and through them I learned a lot of APL." (Roger Hui)

Catherine Lathwell is drawing attention with her documentary film project and its blog, *Chasing The Men Who Stare At Arrays*. Slobodan Blazeski put a spike in our website log with his article "Array languages for Lisp programmers", which appears in this issue. Also in this issue, Graeme Robertson's *A Practical Introduction to APL*, gets reviewed by a Stanford CompSci graduate student. Ajay Askoolum conducts a tour of Visual APL. Dan Baronet has more to tell about using Dyalog's SALT for managing source code. We report meetings in Estonia and California.

Romilly Cocking reunites two old loves: APL and genetic algorithms. Simon Marsden calls on R for more power with statistics. Roger Hui shows that more performance can be wrung out of inner products, and that even the 'unimprovable' idiom for the arithmetic mean can be improved. Ray Polivka revisits the algorithm for finding Easter. An anonymous reader offers a problem of converting between representations to sharpen you up for writing .Net assemblies.

We start two new series. Howard Peelle uses J to play backgammon better. Jan Karman adds to the little published on K and q with a series *Financial math in q*.

Stephen Taylor

N E W S

MEETING

FinnAPL in Tallinn, April 2009

reported by Adrian Smith (adrian@apl385.com)

Location and other social stuff

Tallinn is a bit like York (you can walk everywhere) only the walls are bigger, the sun seems to shine most days, and it can still be seriously cold. The Finns (and Jim Brown) came in by ferry, the rest of us by the cheapest airline we could find. We all enjoyed an excellent guided tour of the old town (pictured below with Jim on the left) followed by a most friendly banquet at a mediaeval-ish restaurant on the evening of the first day.

Our tour group in the centre of Tallinn

The general balance of the seminar was excellent, the real bonus being the presentations in J from two doctoral students at the local technical university. There is a serious hotbed of array programming expertise (and internationally renowned ability to win on internet gambling sites) in Tallinn – maybe the two things go together? One final thing to mention was the collection of *very annoying* puzzles emblazoned with the FinnAPL logo and handed around at the start. I was treated (along with JD) to a level-5 challenge, which I have so far failed to make progress with. Gill has now mastered hers, which will be a great relief to the family when we get home.

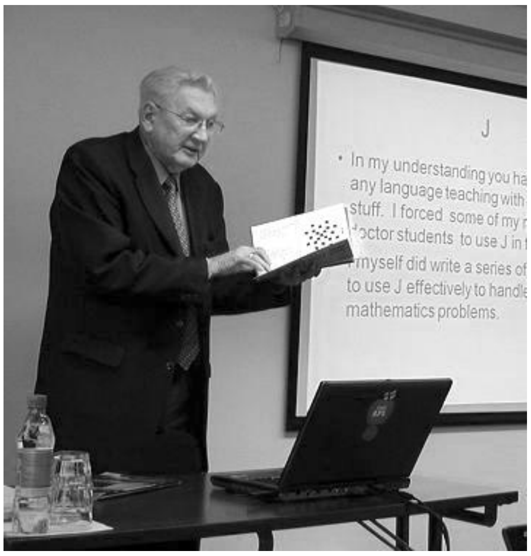
Day 1 – mostly applications

Does APL interest people?
Veli-Matti Jantunen

I think this was just a good excuse to give out the puzzles! Veli-Matti introduced the meeting with the idea that one of the best ways to get people interested in APL was to have solutions to puzzles and brain teasers ready to hand. Programmers are always willing to fill in the quiet times with a bit of puzzle-solving, and students are regularly set data-analysis challenges as part of their training or project work.

So we should be ready with our grab-bag of solutions, whether Sudoku solver, RainPro charting, or just a bit of trivial arithmetic to turn a big pile of numbers into information. He showed some nice examples, for example how easy is is to make a grid with all the negative values coloured red. Even basics like `+/?1000000⍴6` – to check the average of a million dice throws – are impressive to someone new to an array language.

Graph cliques in J
Leo Võhandu, Professor Emeritus in Informatics, Tallinn Technical University


J
In my understanding you ha
any language teaching with
stuff. I forced some of my r
doctor students to use J in t
myself did write a series of
to use J effectively to handle
mathematics problems.
Professor Võhandu on winning at Poker

For me, this was the highlight of the meeting – Leo was a keen follower of Langlet and Michael Zaus back in the 1990s, and has published extensively on experimental mathematics in Russian journals that very few of us in the West ever saw, let alone attempted to read. He is a renowned expert on gaming and lotteries, and also writes regular columns in the popular Estonian press on good algorithms for winning in casinos. Nearly all this work is founded in J, and of the 15 languages he has used:

> "I like J the best – why? Because I am old, fat and lazy!"

So he encouraged many of his masters students to use J in their doctorates, and we saw the result in two excellent talks at this seminar. He thinks there are around 10 active J programmers in Estonia as a result, and I have high hopes we may now see much more of them in *Vector* in the future!

The bulk of the talk was about “how not to look dumb in other fields, from childbirth to space. In space you *always* have enough money!” – a really general approach when faced with a mountain of data is required here, and Leo uses graph theory to search for patterns by progressive clustering of groups and elimination of variables. J proves ideal for handling large boolean matrices, and operations like closure express very nicely and run quite quickly even on huge graphs, although when you get up to 35m vertices, you really need to look towards parallel computing. Sometimes he turns up recording errors in the data, for example in a study of around 5,000 Baltic Herring over 10 generations, the experimenters were expecting 10 groups and the analysis showed 11. It turned out that on one survey trip the trawl was broken and they sampled the wrong shoal!

“Because I am the main gambling expert in Estonia, they read my stuff!”

Which sums up his message – getting published in the popular press is what more of us need to do to keep the language alive. Oh, and it help if you can turn 250€ into 750€ since New Year, with no more that 30min per day on the internet.

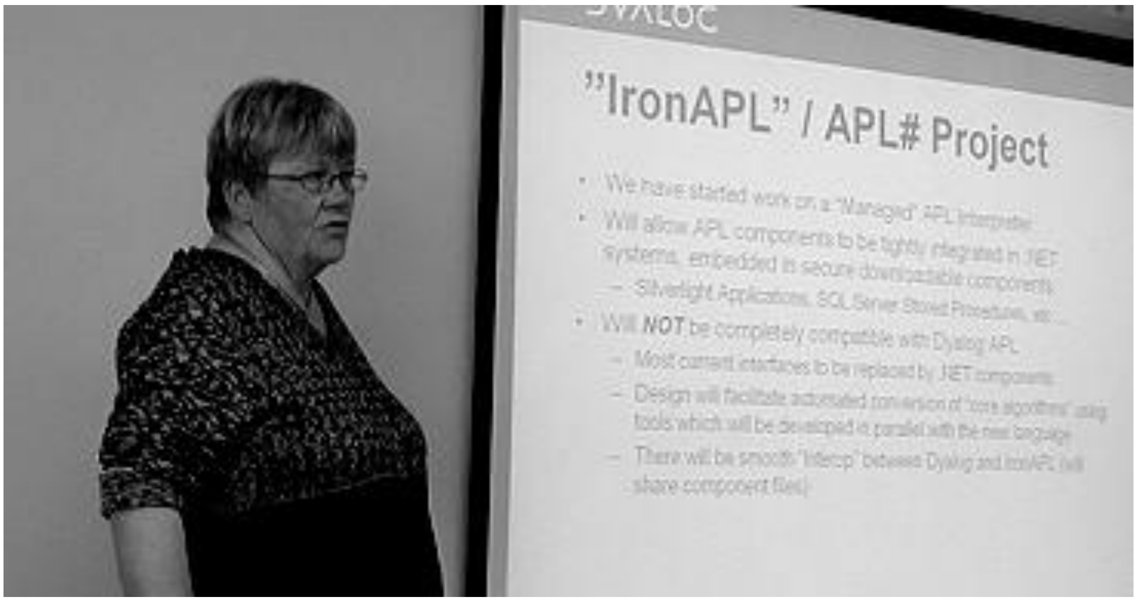
Objects and XML files
Kimmo Linna, FinnAir

Kimmo must have one of the best jobs in the world – flying commercial airliners 2 days a week and maintaining Dyalog APL software the rest of the time. He has been experimenting with trees of namespaces as a way of mapping XML structures into objects, and then in using Twitter as a support infrastructure for an easily accessed and queried logging tool.

The interface to Twitter made good use of Dyalog’s Conga libraries and also Conrad’s MD5 hashing code from the APL Wiki (which you need to encrypt passwords for the login). The result is a stream of XML which can be broken out into a tree of nested namespaces. Every time the application starts, or encounters a problem, it *tweets* a short message with some well-coded supporting data. The restructured XML is easily explored and summarized to provide an efficient support mechanism.

Incidentally, the work in Dyalog 12.1 to provide `⎕XML` complements this nicely, as it takes away much of the pain of parsing the incoming text, reformatting it into a nice matrix layout which Kimmo can easily transform into his preferred object structure. See later!

Dyalog news from *Gitte Christensen*


DYALOG
"IronAPL" / APL# Project
We have started work on a "Managed" APL Interpreter
Will allow APL components to be tightly integrated in .NET systems, embedded in secure downloadable components
Silverlight Applications, SQL Server Stored Procedures, etc.
Will NOT be completely compatible with Dyalog APL
Most current interfaces to be replaced by .NET components
Design will facilitate automated conversion of "core algorithms" using tools which will be developed in parallel with the new language
There will be smooth "Interop" between Dyalog and IronAPL (will share component files)

Gitte outlines the Iron APL project

I think you can read all of this on Dyalog's new pages, apart from the new material on the Iron APL project which caused a lot of interest. This builds on Dyalog's experience with the core library used to support the APL-C# translator to propose a fully managed .Net APL interpreter which could be run as part of a SQL-server stored procedure, called from Microsoft SSRS (their attempt to kill off Crystal Reports with an embedded reporting tool), run as a SilverLight component in a browser, and so on.

This requires it to be very lightweight, and to support the .Net array types very directly so that there is minimal conversion overhead in accessing large arrays of .Net objects. A consequence will be that it can never quite be a direct migration from Dyalog APL, or that there will be some places where arrays are handled a little differently, for example the action of *enclose* does not map exactly, as .Net has no way to represent an enclosed array as a scalar. It will also take multi-argument function calls (probably with optional arguments) as a requirement of the design, so that it will be very easy to work with the .Net libraries. We should see a fleshed-out specification at Princeton in September, which will be extremely interesting.

Analysing text with J
Kairit Sirto, masters student in Linguistics

This talk was a nice example of some relatively straightforward J used in support of a master's thesis in Linguistics. Essentially the task involved a basic text cleanup on a 'cuttings file' of Estonian newspapers, cutting the text into syllables (calling out to a standard DLL) and analysing the n×n matrix of syllable pairs (sparse array handling helps a lot here as the matrix can get rather large).

It was interesting to compare the rather exotic regular expression that could have been used as the first step of the cleanup process with the very simple use of the J *cut* primitive which has a certain amount of hidden power when you dig into it. There were a couple of places where 'old hands' pointed out library functions that could have saved here a bit of time, but (as always) it turned out that Kairit's re-invented wheels did exactly the same job and it probably took her less time to re-invent them that it would have to find the library copies!

Iqx – a database colouring guide
Adrian Smith, Causeway

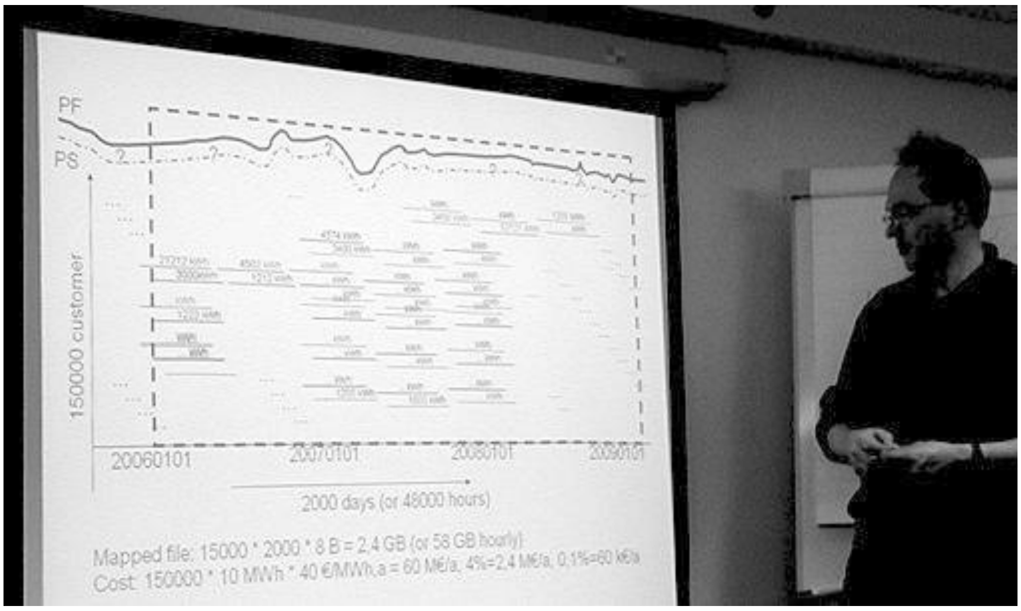
This was the second iteration of the talk I gave in Denmark last autumn, so I tried to show much more of the nuts and bolts of the system, by simply starting with a clear workspace and making a database from scratch. I see the future of this as providing APL newbies with a simple data-editing tool, APL professionals with a rather handy access tool for reasonable size tables, and possibly as a first-cut delivery platform for a completed application (currently being pushed along by the Minster Stoneyard project).

I think a crucial difference from standard SQL-based tools is that it understands nested arrays and matrices (I leaned quite hard on the idea that a matrix can simply sit at the intersection of 2 tables), and that a selection is really just a table in disguise. Yes, it is single-user, it resides entirely in the workspace, and it has no fancy transaction stuff or rollback. Then again, so is kdb and no-one seems to be too bothered now you can have terabytes of main memory and run the engine behind a simple socket library. I will keep chipping away at this as I need it, so expect updates from time to time. Thanks again to Arthur Whitney and Paul Mansour for most of the ideas.

Day 2 – mostly technical

Where did that 4% go – a short challenge from *Anssi Seppälä*

This was a 10-second filler to challenge us to code up the search strategy to find the 4% of the total Finnish electricity consumption that no-one gets charged for.

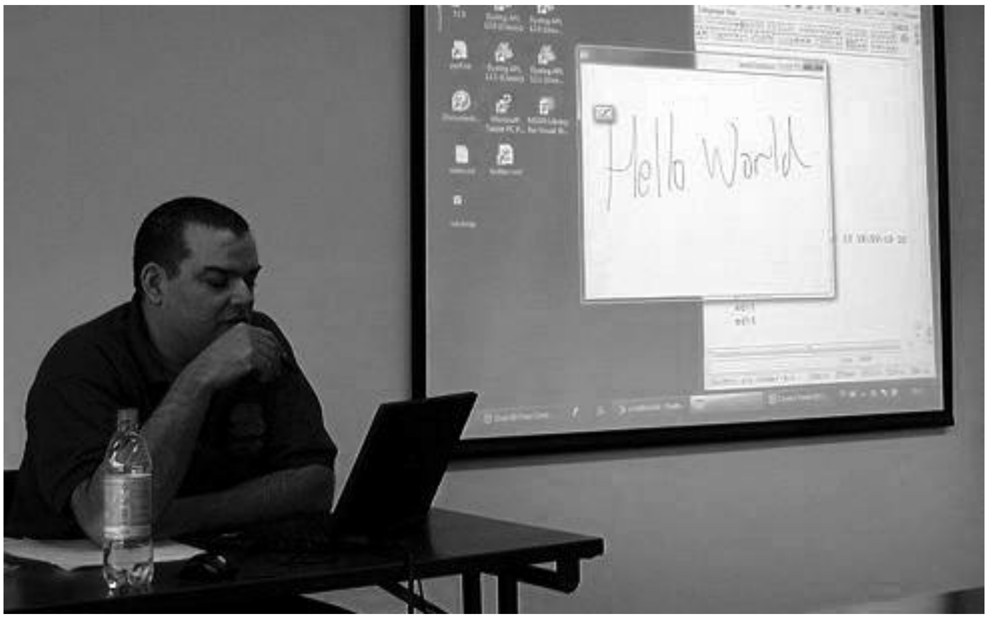

PF
PS
150000 customer
20060101
20070101
20080101
20090101
2000 days (or 48000 hours)
Mapped file: 15000 * 2000 * 8 B = 2,4 GB (or 58 GB hourly)
Cost 150000 * 10 MWh * 40 €/MWh,a = 60 M€/a, 4%=2,4 M€/a, 0,1%=60 k€/a

Anssi Seppälä looking for the missing 4% of Finnish electricity

The dataset is very large – hourly data is available by consumer for several years. The problem is that when you add it up, the sum is around 4% less than the amount you know went through the grid (allowing for known losses). For anyone who can find it, there are several million Euros on offer.

New features in Dyalog 12.1
John Daintree, Dyalog Ltd

John covered a lot of stuff in a short time, with several rather natty demos along the way, so this will inevitably be a rather patchy summary. Version 12.1 is rapidly nearing code-freeze, so we may get our hands on a test copy quite soon.



John Daintree demonstrating handwriting recognition in Dyalog 12.1

Dyalog have been working with MicroAPL to make a standard implementation of `⎕XML` to do all the heavy lifting and parse up a stream of incoming XML-coded data into an easy-to-access format. This looks like a nice old-style matrix, with tags classified by depth, and columns for the attributes and values – you can display a big chunk of XML (such as Kimmo's Twitter logs) in a tree-control almost trivially. Of course the process is reversible, so to save (for example) an IQX database as a stream of XML would become a much less tedious programming task, and re-constituting it would be almost as easy. I will definitely need this to augment my current approach of simply dumping the `⎕OR` of the namespace on a component file, so this little addition definitely gets a hug or two from me!

Next up were a bunch of editor enhancements that make working with 'real' (as opposed to 'toy') classes a practical proposition. Both line-numbering and syntax-colouring currently die if you have more than 10,000 lines in the object being edited (this editor was designed for functions after all, and we all start to feel uneasy if we have more that 100 lines). Now we have a rewritten syntax-colouring engine, a proper navigation tree on the left, tagged sections with an outliner, double-click can start a new function (or jump to an existing one) and stops can be set in the traditional way. Even the tracer has got a bit colourful.

The class remembers how it was when you left it, so you can run with nearly all your code collapsed, and you can dive straight to the function you want with the traditional `)ed myclass.foo` from the session. I'm sure there are a few bugs left to shake out, but I can think of enough places where this will help that I am willing to give it a run out and tolerate the inevitable annoyances!

Next up – the *really cool* stuff!

You used to have a choice between old-style forms made with `⎕WC` and the new Windows Forms library you get from Microsoft. Now you can just add one of the standard .Net controls (or any number of controls that you can download for free) to your form using the new `NetControl` type which mirrors `OCXClass` in putting a lightweight wrapper around any control you like. John showed some of the 60 free controls from DevExpress – hence that natty handwriting control working on his tablet PC – and illustrated a grid with a handy drop-down calculator implemented with a .Net control as its `input` property. This opens a lot of doors to the application designer, and frees up Dyalog from trying to keep up with all the rest of the GUI world. Applause please!

Finally (but worth the wait) we can now package up the workspace, runtime interpreter, and an icon into a single executable which just makes the installation game that much simpler. I still have a bad habit of updating my DSS copy, resaving the workspace, recreating the .exe, mailing it to the user, and then getting a puzzled message back about incompatible versions as soon as she tries to start it. Not any more – thanks John!

Roundup

All in all, a most satisfactory outing. The post-conference feedback was all positive, and the attendance was a little up on last year (30+ people at most of the talks). There is definitely a little J bushfire going in Estonia – we need to do all we can to fan the flames and get this message spread more widely before the enthusiasm fades. *Vector* can offer its pages for the masters students to publish in a 'well-respected' English-language journal, which can only help to encourage a community which still feels the residue of Soviet occupation, and where getting your message heard has been very tough in the quite recent past.

D I S C O V E R

Review

A newbie's discovery of APL

Rebecca Burriesci (rburriesci@gmail.com)

A Practical Introduction to APL 1&2
A Practical Introduction to APL 3&4
56pp & 196pp, Graeme Robertson, Basingstoke, 2008

Meant for newbies, Graeme Robertson's *A Practical Introduction to APL (1&2 and 3&4)* seemed perfect for me because I've only just heard about APL. My background is in C++/C# .net/JAVA, although I've dabbled in C, PBasic, Ruby, PHP, JavaScript, and what I think will help me the most with APL, Matlab. I started programming when I was 10, which has impressed on me that all good introductory books are written so as to be appropriate for a pre-teen. As I go through the books, I'll let you know whether Robertson can clear that high bar while introducing a new user to the subject.

Getting started

The structure of the first book is split into day-long modules, each Day into a couple of sessions, and each session into six or seven lessons. The lessons generally start right away with some examples, then provide some notes or explanation, and occasionally end with exercises. The book is designed to complement a class on APL, complete with interspersed questions to ask a tutor and a certificate of achievement on the back cover.

There is a very brief introduction, but it doesn't answer the who-what-why question of most introductions to programming books. The introduction also doesn't explain the style conventions of the book – on the first page of lesson 1, I saw eight different font styles. The introduction does include Robertson's claim that APL is simple and easy to learn, and encourages easy experimentation. We'll see if that claim is right.

The first problem I ran into is that I'm not taking the class. Lesson 0 began by telling me the APL session should be visible and ready to interact with, which of course it's not. I don't see a reference or appendix that tells me where to go to get whatever software Robertson is using.

Fortunately, I was able to get APLX, an APL interpreter, from MicroAPL [3]. The install was extremely simple and worked with only a few small hitches (which had more to do with me being too lazy to read the instructions and forgetting to install as root).

Symbols and more symbols

After going through the first few sections, I can easily see why Robertson started with us exploring the keyboard and filling in a page with a blank keyboard with where our symbols are. There are over 200 symbols in this nutty language! An index in the book with a summary per symbol would have been nice, as during the examples I found myself constantly flipping back to review what a symbol does.

The first surprising thing Robertson calls out is the lack of order of operations. I guess with over 200 symbols for mathematical notation it probably is easier just to work right to left.

Flexibility of lists, arrays and matrices

Starting with lists, I began to understand why APL is powerful and easy to experiment with. Robertson does a great job in showing, rather than just telling, the awesomeness of lists. To start with, the list notation of spacing makes working with lists and matrices easier than with any other language I've seen.

The best part is that the interpretation of the list of numbers is up to you – sometimes it is a list as a whole, sometimes an array of individual elements.

For example, here we generate a list of numbers from 1 to a scalar

```
      ⍳3
```

we can act on each element individually

```
      - 1 2 3
¯1 ¯2 ¯3
```

or sum the elements in the list

```
      + / 1 2 3
```

```
6
```

or multiply each matching element on each list

```
      1 2 3 × 1 2 3
1 4 9
```

Here we can use the reshape function `ρ`. We make the list as a whole have 7 numbers:

```
      7 ρ1 2 3
1 2 3 1 2 3 1
      7 ρ ι3
1 2 3 1 2 3 1
```

and now we are going to catenate that with another list

```
      (7 ρ ι3),5 6
1 2 3 1 2 3 1 5 6
```

and now turn it into a matrix

```
      3 3 ρ (7 ρ ι3),5 6
1 2 3
1 2 3
1 5 6
```

Have you ever seen anything so elegant?

Most languages fumble through this and make you constantly redefine the data explicitly. However, APL just lets you very naturally use lists to represent a grouping or a sequence of numbers as needed.

Robertson does an excellent job of showing us all the different operations you can perform on lists, and how easily to generate lists from scalars, matrices from lists, and lists and scalars from matrices, and round and round it goes.

Fun with functions

A language can't be complete without providing the user with a way to specify functions. The syntax for APL is a little awkward, starting and ending the function with a special symbol. Reminiscent of Basic, we can branch to a line number or a label in the function.

The problem is that Robertson doesn't tell us the rules for defining functions, just gives a few sentences and a few examples (one of which has a bug in it). I had to go through the examples carefully before I found that branching to line 0 exits the function.

As I've now come to expect, there is another level to functions – the function fixer and execute commands. You can turn a string into a function, or just execute a string as an expression. This continues APL's flexibility with lists in treating data as data and interpreting it as you want at will. Is it a string? A function? An array? A matrix? You decide.

But the coolness factor isn't just in executing strings – you can write programs that write programs. This is so interesting that I wish Robertson had given it more than a single line of explanation and short example.

Therefore, I decided to honour the coders before me and do something steeped in tradition – write a really inefficient factorial function. With only Robertson's book at my side, can I write a program that writes the factorial function for me?

I started by writing the simple recursive factorial function. This proved harder than I thought because Robertson doesn't explain how to do conditional branching, and I had to go over the examples carefully before I found what I needed in the looping sum example.

Recursive function:

```
[0] A←FAC N
[1] →(N=0)ρ4
[2] A←N×(FAC N-1)
[3] →0
[4] A←1
```

Here is my favorite factorial function demonstrating the power of the built in symbols:

```
[0] A←FAC2 N
[1] A←× / ⍳N
```

And lastly, here is a function that creates a function to just do the one factorial problem, and then executes it:

```
[0] A←FAC3 N
[1] ⎕FX 2 9 ρ 'A←FAC_',⍕N,'  A←× / ⍳N'
[2] ⍎'FAC_',⍕N
```

It took a bit of playing around with the symbols and syntax to be able to write the factorial functions, but I was eventually able to figure it out from Robertson's descriptions of other functions and symbols.

Reduce and Scan

The reduce operator deserves a little time and attention for making operations that are really messy to express in most languages, really easy to write in APL. It applies a function left argument to a list right argument.

In the second factorial function above (`A←× / ⍳N`), I generated a list of numbers from 1 to N (`⍳N`), and then used reduce `/` to apply the multiply function to all the elements of the list and multiply them together. This lets us do very sophisticated things very easily. In one of Robertson's exercises, he asks us to output the sum of the squares of the first 10 positive numbers. If I were using another language, I might write something like

```
int total = 0;
for (int i=1; i<11; i++) {
   total = (i*i) + total;
}
```

But in APL, I can now do

```
      +/(⍳10) × (⍳10)
```

Again, we can see how easy it is to go from a scalar `10`, to a list `⍳10`, operate on it like a list of separate items (`(⍳10) × ⍳10`), and then operate on that list like a stream of arguments (`+/`). This flexibility built into APL is what validates Robertson's initial claim of APL being intuitive and powerful.

Summary of Days 1-2

That is about all there is to Days 1 and 2. Robertson interjects some nice APL history and philosophy into the lessons, but I'll let you read that yourself. For myself, I feel like Robertson introduced me to enough APL that I could write basic functions and experiment with simple math operations and matrix manipulation quickly and easily.

Days 3 and 4

Days 3 and 4 cover Dyalog and graphical user interfaces. They follow the same format as Days 1 and 2, except with modules instead of lessons.

The book doesn't start too smoothly, however. Again, Robertson doesn't ground us – no introduction, no transition, no software setup. He just launches straight into a dense definition/example list for the next… well, two days. It would take me as long as his book to go over the material, so here are just the highlights:

Robertson introduces us to Dyalog, an object oriented language with its roots in APL. The awesome power of Dyalog is its seamless integration with other languages and Windows programming. However, Robertson's level of detail seems a little light: what it is, an example, and then onward. Robertson rushes through data types, more control flow, and GUI objects in Dyalog. If you've ever tried to do Windows programming in C++, then you'll share my enjoyment of how easy manipulating forms is in Dyalog. However, if I hadn't already understood object oriented programming, I would have been confused by Robertson's fast pace and lack of explanation.

The integration with other languages is also a neat feature. Robertson shows us the range of Dyalog with integrations with C, Microsoft .Net, TCP/IP and remote applications, and more.

Conclusion

Overall, Robertson delivers on his promise: a practical introduction to APL. Robertson spends almost the entire first book translating mathematical notation into APL for us, and ends up being a little light on the explanation, examples, exercise solutions, and future direction. I wouldn't recommend it for a 10-year-old first time programmer, as Robertson assumes knowledge of things like boolean arithmetic and object-oriented programming, and doesn't provide any flashy long-running examples that teach you good principles of coding, and what to do with all the random symbols you've just learned. The second book is so densely packed that you can't do much more than skim and remember where to return to when you want to use some of the functionality. However, by taking the APL tools out of the box one by one and describing each with an example, Robertson clearly demonstrates the flexibility, the power, and the intuitiveness of APL.

References

1. *A Practical Introduction to APL 1&2*, Graeme Robertson, Basingstoke, 2008, 56pp
2. *A Practical Introduction to APL 3&4*, Graeme Robertson, Basingstoke, 2008, 196pp
3. APLX. MicroAPL. http://www.microapl.co.uk/apl/aplx_downloads.html

Rediscovering APL’s history

by Catherine Lathwell

Catherine Lathwell’s talk to the APL Bay Area Users’ Group on 11 May 2009, reported by Curtis A. Jones

In March Catherine Lathwell’s blog [1] on making a movie about the history of APL mentioned a trip to the San Francisco Bay area. She quickly accepted an invitation to speak to the APL Bay Area Users’ Group (APL BUG or The Northern California SIGAPL of the ACM) on the trip. Thirteen people assembled at a pizzeria before the meeting on the 11th of May. Gene McDonnell showed a proof copy of *At Play With J* [2]. Ed Cherlin showed a computer for the One Laptop per Child [3] project and said two people are working on two APLs for it. Catherine may have gotten something to eat, but mostly she took advantage of the chance to introduce herself to everyone who came.

Joey Tuttle

The meeting itself was at the Computer History Museum [4] in Mountain View, California. After the more or less annual election of officers under Chuck Kennedy’s chairmanship, Joey Tuttle introduced Catherine. He recalled walking up to Catherine Lathwell’s desk at Reuters in Toronto and addressing her as “Cathy”. Her coworkers asked how Joey could get away with “Cathy” when they’d get an earful if they attempted such familiarity. Her answer: “Grandfather rights.” Catherine was born when her father, Richard Lathwell, worked with Larry Breed and Roger Moore to take Iverson notation from an elegant way of writing algorithms to APL\360 that could actually be run on a computer. Joey, Larry Breed, Eugene McDonnell, David Allen and Charles Brenner were the people in attendance who might claim grandfather rights.

Catherine saw first hand the rapid growth of APL from the late 1960s through the 1970s, and even worked herself as an APL programmer at I.P. Sharp Associates and Reuters. Her academic area has been fine arts and English literature. Now she's reflecting on her experience of APL and, as an independent producer, starting to organise a documentary film on the history of APL.

A first step is her blog to scope out interest in the project. The Toronto APL SIG has provided seed money, and she is out looking for more support. She encouraged us to check the blog frequently since readership indicates interest in the project to potential funders.

Getting the history of APL into a finite-length documentary requires serious attention to the scope of the topic! The working question around which Catherine started organizing the movie is *What is APL's contribution to science?* and it is creeping to *What is APL's contribution to computer science?* and *What is APL's contribution to international finance?*.

Movie? She made one for the APL BUG with greetings from Dick Lathwell. This video, with some additional editing, is on YouTube, and may be found through Catherine's blog. The title *Dad & My APL Box* describes the beginning in which Dick shows some memorabilia. There's the issue of the *IBM Systems Journal* celebrating the 25th Anniversary of APL [5], a record of "APL Blossom Time" composed by J.C.L. Guest (Mike Montalbano – "A wonderful person"), some I.P. Sharp Associates' newsletters and a picture of the 'original six'. These are Dick Lathwell, Ken Iverson, Roger Moore, Adin Falkoff, Phil Abrams and Larry Breed. [6] Dick noted that not everyone in the picture worked on the same program at the same time. Phil Abrams, in fact, never worked directly for IBM. Phil worked for SRI with Larry Breed to write IVSYS which computed with input in Iverson notation. Later Dick Lathwell converted IVSYS to run on the IBM 7040 computer at the University of Alberta.

Here are some exchanges in the video (with some editing).

> Catherine Lathwell: *Tell me about the first time you met Ken [Iverson].*
>
> Richard Lathwell: Ken had come to the University of Alberta around 1964 to lecture on describing hardware architecture using his notation. I didn't meet Ken personally until I worked on converting IVSYS to run on the 7040.
>
> Dick's thesis advisor left the University of Alberta and suggested that better research was being done in industry. Dick went to Toronto to interview Honeywell. On the day he was to accept Honeywell's job offer a

call came from Ken Iverson: “I hear you’re leaving. Come and talk to us.” Dick’s advisor must have called Ken.

CL: *Do you remember the first time you met Adin?*

RL: No. Probably when I went to be interviewed. I felt overawed. New York and its suburbs were overwhelming.

CL: *I always remember being a little bit afraid of Ken but I remember Adin as being kind of gentle.*

RL: Yes. Our first two cars came from Adin. The first was a 1956 Cadillac. Adin’s view was that a ten-year-old luxury car, because it was built so well, gave a better deal than a cheap new car. He sold his 1956 Cadillac to us for $50 in 1966. Then later he sold a 1957 Cadillac for about $100. We later sold it to Mike Jenkins who ended up taking it to Queens University and spending a fortune to have it certified in Canada.

CL: *Do you have any comments to the people who will be watching at the museum?*

RL: I think the thing that’s important – what I always come back to – I have downstairs somewhere Roger’s yellow pad with his design of the APL\360 supervisor – but the thing that we did always was to do our design in APL and so we would write whatever it was we were going to do first in APL and use that to prove it out and check it and then transcribe that into assembler language which is what we programmed in in those days. By doing that – by using that tool – we were able to produce really good code significantly faster than any other way and pretty much faster and more accurately than what anybody else was doing at the time.

And IBM never got it. In fact at one point I remember a vice president trying to convince me to do something in a different way and I kept trying to convince him that the reason we were successful was because we used our own tool and the IBM vice president said no, it was because I was smart. I remember mentioning that to Ken later, and Ken said “You should have asked him ‘If you think I’m so smart, why don’t you take my advice?’”

CL: *That sounds so much like Ken. Did I tell you what Ken said when I told him about doing this?*

RL: No.

CL: *"OK. Maybe all that art school is good for something after all."*

Here are a few highlights of the question-and-answer session after the video.

Larry M. Breed: Mentioned Phil Abrams and Dick Lathwell working with FORTRAN on IVSYS before APL\360. [7]

Curtis A. Jones: Would Ken Iverson have wanted APL's contribution to science to be through education?

LMB: Ken's dream was to improve communication of mathematical ideas. [8] Howard Aiken's advice to Ken: "Don't worry about people stealing your ideas. If they're any good you'll have to stuff them down their throats."

LMB: Lack of declarations may have limited APL.

Joey K. Tuttle: Recalled meeting with Frito Lay in 1970s – their most important application was modelling a potato in APL.

CAJ: How Important was language to APLers?

LMB: Everyone had strong view on correct dictionary. Ken favored the American Heritage Dictionary.

JKT: The last book from Ken was *Eats, Shoots and Leaves*.

Catherine played a video montage created from "APL Blossom Time" [9] and Peter McDonnell's much-loved caricatures of the APL veterans that is also the closing sequence of her video of Ken Iverson's Toronto Memorial, which can be found in Myspace [10].

Thanks to the Lathwells for the talk. Thanks to The Computer History Museum and the head of its board of trustees, Len Shustek, who once worked with Dick Lathwell, for hosting the event.

Notes

1. http://aprogramminglanguage.com

2. *At Play With J*, Eugene E. McDonnell, Vector Books, UK 2009, ISBN 978-1-898728-16-0
3. One Laptop per Child (OLPC) http://laptop.org/
4. Computer History Museum http://www.computerhistory.org/
5. *IBM Systems Journal*, Volume 30, Number 4, 1991
6. The Computer History Museum has a large print of this picture donated by Eugene McDonnell, who received it from Don McIntyre. CAJ
7. See Mike Montalbano's "A Personal History of APL". http://www.ed-thelen.org/comp-hist/APL-hist.html
8. "Programming may be our middle name but language is our end." CAJ
9. The audio recording of "APL Blossom Time" may be found at http://smartarrays.com/downloads/aplblossomtime/aplblossomtime.html "Even today it is sold out in most record stores." or http://cosy.com/language/APLBlos/APLBlos.htm
   "APL Blossom Time" is annotated in Mike Montalbano's "A Personal History of APL" at http://www.ed-thelen.org/comp-hist/APL-hist.html
10. http://vids.myspace.com/index.cfm?fuseaction=vids.individual&videoid=49489049

Array languages for Lisp programmers

by Slobodan Blazeski (slobodan.blazeski@gmail.com)

A language that doesn’t affect the way you think about programming is not worth knowing. [1]

You can post comments on this article at the *Vector* blog. *Ed.*

During my pilgrimage into the land of programming languages I’ve stumbled upon quite a lot of them. Most languages are just poorly-integrated duct-tape-and-wire rehashes of principles discovered by others. Few of them are true paradigm-shifting jewels. APL and its offspring are definitely among those. But what’s their value for spoiled Lisper brats? What’s so great about them that Lisp doesn’t already have? During my whirlwind presentation of examples I will try to answer this question.

The first thing that comes to mind is that array-languages such as APL, J and q operate on whole arrays instead of munching elements one by one.

By using array functions you can work on whole containers the same way as you work on scalars. It takes only a little tinkering with CLOS[2] dispatch to make a set of operators that work roughly the same. (Adding ranks will take more time than I currently have.) [3]

J	Common Lisp
`   2 + 2 3 4`	`(j+ 2 #(2 3 4))`
`4 5 6`	`4 5 6`
`   2 3 4 + 1 2 3`	`(j+ #(2 3 4) #(1 2 3))`
`3 5 7`	`3 5 7`

Then having a set of operators for making arrays is very handy, so I wrote those quickly in Lisp.

J **Common Lisp**

```
   2 3 $ 'abcde'     (shape '(2 3) "abcde")
abc                 abc
dea                 dea

   i. 2 3            (enum 2 3)
0 1 2               0 1 2
3 4 5               3 4 5
```

Adverbs give great power to J, but Lisp has had higher-order functions since forever. Things like built-in `reduce` already work on vectors and it doesn't take much work to create a version of `reduce` such as `fold` that handles multidimensional arrays.

J **Common Lisp**

```
   +/ 1 2 3 4     (reduce #'+ #(1 2 3 4))
10             10

   +/ i. 2 3      (fold #'+ #(1 2 3 4))
3 5 7          3 5 7
```

And once you have them it's too easy to write factorial of 6 in array style:

J **Common Lisp**

```
   */ 1+ i. 5  (reduce #'+ (j+ 1(enum 5)) or
120            (fold #'j+ (enum :start 1))
```

One of q's very cool features allows omitting the first three parameters of a function. That's a feature that I immediately ported to Lisp with a little macro.

q	Common Lisp
`{x+y+z}⇔{[x;y;z] x+y+z}`	`(f + x y z) ⇔` `(lambda (x y z)(+ x y z))`

But maybe it's not about single features but the ways to combine them into non-trivial programs and problem solutions.

Just take a look at the Lisp code answering the task below: [4]

> We have a list of elements, some are duplicates. We're trying to figure out how to find the duplicate elements and increase a counter value by 1 for each instance of the element found. The list consists of lists with two elements, the first being the incremental counter, the second being the string sought. Example:
>
> ```
> ((1 "one") (1 "two") (1 "three") (1 "one") (1 "four") (1
> "two"))
> ```
>
> The result should be:
>
> ```
> ((2 "one") (2 "two") (1 "three") (1 "four"))
> ```

q

```
count:flip(count each group v[;1];unique v[;1])
```

Lisp

```
(defun count (list)
  (let ((hash (make-hash-table)))
    (dolist (el list)
      (incf (gethash (cadr el) hash 0) (car el)))
    (let (result)
      (maphash (lambda (key val)
        (push (list val key) result))
```

```
          hash)
        result)))
```

The above seems an exemplary case where Lisp loses its magic. But that’s only because q has `group` built in. If I write it, or being a lazy person like myself ask c.l.l [5] denizens to write it for me, the edge is lost.

q

```
brunberg:flip(count each group v[;1];unique v[;1])
```

Lisp

```
(defun brunberg (l)
  (mapcar (f list (length x) (cadar x))  (group l #'cadr
#'string=)))
```

The q solution is still shorter but only due to a Lisp culture of very long and descriptive names. Using a golfing [6] library with operators’ bindings pointing to shorter names will make it look even shorter.

Another fine example is the problem of parsing identifiers:

> This piece of [Haskell] code goes through a parse tree of Haskell source code, locates every reference to an identifier that ends with `Widget`, puts it on a list, and removes duplicates so every identifier is represented in the list only once. [7]

Haskell

```
extractWidgets :: (Data a) => a -> [String]
extractWidgets = nub . map (\(HsIdent a)->a) . listify
isWidget
  where isWidget (HsIdent actionName)
      | "Widget" `isSuffixOf` actionName = True
    isWidget _ = False
```

q

```
a:distinct raze over x
a where a like "*Widget"
```

Lisp

```
(defun widgets (l)
  (unique (keep (f like x "*Widget") (flatten l))))
```

This time the difference is even smaller. I have the same number of tokens with a surplus of parentheses.

It seems it's always the same old story. As soon as Lisp integrates the needed utility or invests into a domain-specific language, the advantage of the other language is lost. The big ball of mud is like a warlock, able to steal opponents' powers while still staying the same.

However the features above were easy compared to the work needed to implement my beloved tacit [8] style. While point-free style isn't a complete stranger to the Lisp community, it remains a rarity [9]:

Lisp

```
(reduce  (flip #'cons) #(1 2 3 4) :initial-value '())
(4 3 2 1)
```

Writing a macro that understands forks and hooks is an exercise for a beginner – if the operators covered are strictly monadic and/or dyadic as are those in J. However in Lisp all those optional auxiliary and keyword arguments immensely complicate matters. In order to make it operational within a month or so of effort a tacit macro could be implemented to work just on a subset of Lisp operators.

A library consisting of operators with integrated looping, adverbs that understand verb rank, and a macro able to operate on hooks and forks would bridge the gap in Lisp array-processing facilities. But it's not enough to bridge the gap between technology and need. Somebody needs to make humans want to cross that bridge or else it will follow the fate of Richard C. Waters' series [10].

So why should Lispers study array programming languages?

Learning anything of APL, J or q makes programmers aware of opportunities opened by thinking at the higher level of container abstraction. This knowledge pays off well in Lisp, with or without the array sublanguage. Keep in mind too that the difference between an array sublanguage and an array language is like the difference between learning a foreign language at home or among its native speakers. You *can* learn just the language while remaining clueless about its culture!

loop

```
(loop for i from 0 to 10
  collect ( loop for j from 0 to 10
    collect (+ i j)))
```

functional

```
(flatten (outer #'+ (range 10) (range 10)))
```

loop

```
(defun separate (list)
  (loop for element in list
    if (plusp element)
    collect element into positives
    else if (minusp element)
    collect (abs element) into minuses
    else collect element into zeroes
    finally (return (list minuses zeroes positives))))
```

functional

```
(defun separate (lst)
  (mapcar  (f mapcar #'abs (remove-if-not x lst)))
    (list #'plusp #'zerop #'minusp)))
```

Notes and references

4. Alan J. Perlis, *Epigrams in Programming*
5. http://en.wikipedia.org/wiki/Common_Lisp_Object_System
6. The Lisp return values were presented in J style for increased readability.
7. http://www.nsl.com/papers/kisntlisp.htm

8. comp.lang.lisp
   http://groups.google.com/group/comp.lang.lisp/
9. Golfing:
   http://en.wikipedia.org/wiki/Perl#Perl_golf
   Some of the functional solutions were developed in the c.l.l. golfing sessions with feedback from many c.l.l. denizens.
10. http://www.defmacro.org/ramblings/haskell-productivity.html
11. http://en.wikipedia.org/wiki/Tacit_programming
12. Vassil Nikolov – “Could loop loop backward across vector?”
13. http://series.sourceforge.net/
14. The first three solutions in the last block were written by Kenny Tilton, Matt Pillsbury and Rob St Amant respectively. Though Tilton’s and Amant’s solutions are far longer they’re also more efficient.

APLX interface to R statistics

by Simon Marsden, MicroAPL Ltd

This article is based upon a talk originally presented at the BAPLA09 Conference in June 2009.

APLX Version 5 includes a new interface to the R statistics package.

R is a computer language in its own right, but it also comes with a large collection of library functions. Most are concerned with statistics but there are many others, including excellent support for producing graphs.

There have been attempts at implementing a statistics library in APL, but it's difficult to approach the comprehensive coverage that R brings. There are functions for linear and non-linear modelling, classical statistical tests, time-series analysis, classification, clustering, and many others.

I'm no statistician – I'm not even qualified to pronounce the names of some of the statistical tests that R offers, let alone explain them – but the whole of the huge, well-tested R library can be called easily from APL.

R is a GNU project, based on the S language and environment developed at Bell Labs. It's open-source, and free under the GNU General Public Licence. It's available on many major platforms, including Windows, Macintosh and Linux.

A quick introduction to the APLX-to-R interface

Let's look at a quick example to get a feel for what's possible when APL is combined with R. (This is in part based on material in the *Introduction to R* manual[1]. Don't worry too much about the details yet: we'll come to those later.

We need to start by telling APLX to load the R plug-in. We'll assign a reference to a variable, also called `r`

```
      r←'r' ⎕NEW 'r'
      r
[r:R]
```

```
      r.⎕DS
R version 2.9.0 (2009-04-17)
```

I mentioned that R has rather a lot of functions. We can get a listing of them from APLX:

```
      ⍴r.⎕DESC 3
2051 117
      r.⎕DESC 3
AIC (object, …, k = 2)
ARMAacf (ar = numeric(0), ma = numeric(0), lag.max = r, pacf =
FALSE)
ARMAtoMA (ar = numeric(0), ma = numeric(0), lag.max)
Arg
Arith
Axis (x = NULL, at = NULL, …, side, labels = NULL)
Box.test (x, lag = 1, type = c("Box-Pierce", "Ljung-Box"), fitdf =
0)
C (object, contr, how.many, …)
CIDFont (family, cmap, cmapEncoding, pdfresource = "")
…etc
```

As you can see, there are well over 2,000 functions in the base package alone, and that's before we start loading additional libraries.

R comes with a number of sample data sets for use in teaching statistics. For example there is a data set giving eruption details for the Old Faithful geyser in the Yellowstone National Park, famous for the regularity of its eruptions. The data is stored in an R data type called a `data.frame`:

```
      r.faithful
[r:frame]

      18O↑r.faithful.⎕DS
    eruptions waiting
```

```
1        3.600       79

2        1.800       54

3        3.333       74

4        2.283       62

5        4.533       85

6        2.883       55

7        4.700       88
```

Let’s get the data for the eruptions column (durations of eruptions in minutes) into an APL variable:

```
      eruptions←(r.faithful).$$ 'eruptions'
      10↑eruptions
3.6 1.8 3.333 2.283 4.533 2.883 4.7 3.6 1.95 4.35
```

We can perform some simple statistical tests using R methods like `mean` and `summary`:

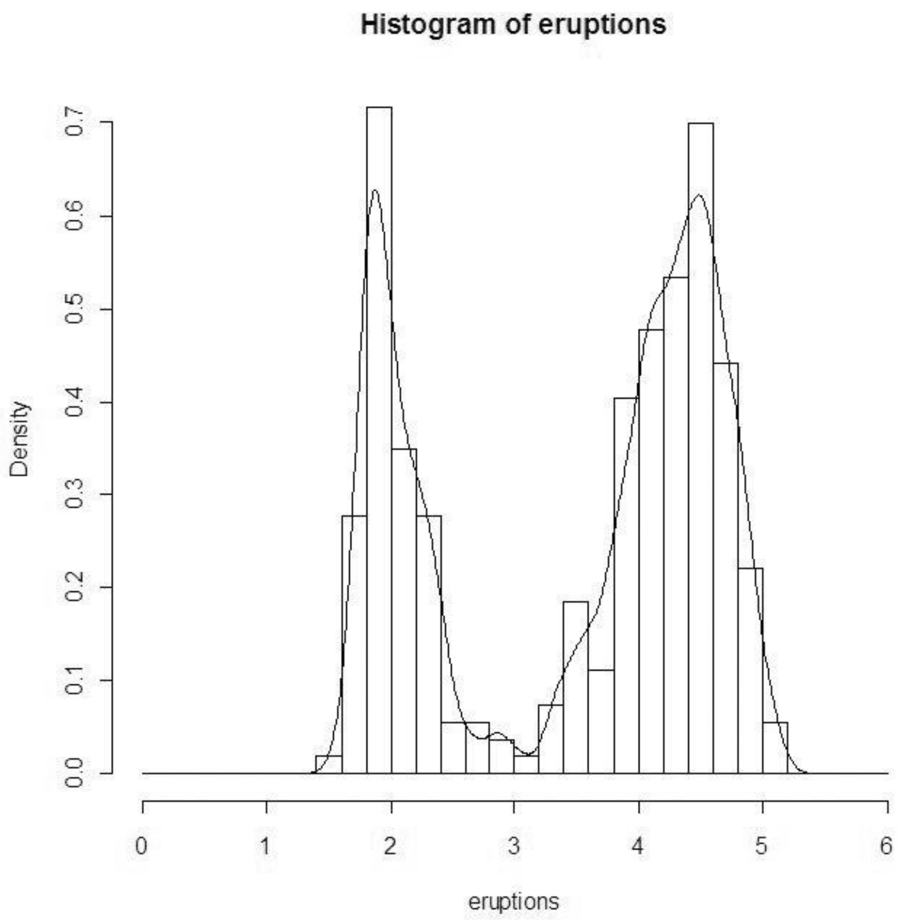
```
      r.mean (⊂eruptions)
3.487783088
      r.summary (⊂eruptions)
1.6 2.163 4 3.488 4.454 5.1
```

The summary is clearer when presented in text form like this:

```
      (r.summary.⎕REF (⊂eruptions)).⎕DS
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.600   2.163   4.000   3.488   4.454   5.100
```

Let’s get R to plot the APL data as a histogram:

```
      r.eruptions←eruptions
      r.⎕EVAL 'hist(eruptions,seq(0,6,0.2),prob=TRUE)'
      r.⎕EVAL 'lines(density(eruptions,bw=0.1))'
```



Looking at the histogram, we can see that Old Faithful sometimes stops after 3 minutes. If it keeps going beyond 3 minutes, the eruption length data appears to conform roughly to a Normal distribution. How can we test this?

The Kolmogorov-Smirnov test allows us to test the null hypothesis that the observed data fits a given theoretical distribution:

```
      r.long←(eruptions > 3)/eruptions

      ks←r.⎕EVAL 'ks.test (long, "pnorm", mean=mean(long),
sd=sd(long))'

      ks
[r:htest]

      ks.⎕DS


 One-sample Kolmogorov-Smirnov test
```

```
data:  long

D = 0.0661, p-value = 0.4284

alternative hypothesis: two-sided


      ks.⎕VAL

 0.06613335935  0.4283591902  two-sided One-sample Kolmogorov-
Smirnov test
```

This brief example gives a flavour of what's possible when using R from APL : easy access to a huge library of statistical and other functions, and rich graphics.

A practical introduction to R

Let's leave APL out of the picture for a while, and take a look at some of the features of R by typing expressions at the R console window's `>` prompt.

First of all, you'll notice that R is an interpreted language:

```
> 2+2

[1] 4
```

It's also an array language. Here's the equivalent of `⍳10`:

```
> 1:10

 [1]  1  2  3  4  5  6  7  8  9 10
```

You can assign a value to a variable. Notice that the assignment arrow is similar to APL, but of course we don't have APL's beautiful symbols:

```
> x<-1:10

> x+x

 [1]  2  4  6  8 10 12 14 16 18 20
```

There are a limited number of symbols like `+` and `*`. For everything else we must use a function:

```
> sum(1:10)
[1] 55
> length(1:10)
[1] 10
```

R is similar to APL in that it has the concept of a workspace. If we try quitting, we get prompted whether to save the session. If we say ‘yes’ and then quit, all the variables will be there next time we launch R.

Like APL, higher order arrays are possible in R:

```
> x<-matrix (1:20, nrow=4)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

Notice that R uses column-major order, something that the APL interface must take care of transparently.

Entering a vector is a bit clumsy. You might expect that you can type:

```
> x<-1 2 3 4 5
Error: unexpected numeric constant in "x<-1 2"
```

…but you need to use a function called `c` and make each item an argument:

```
> x<-c(1,2,3,4,5)
```

Rather strangely, R has no concept of scalars

```
> length(1:10)
[1] 10
> length(123)
[1] 1
```

123 is a single-element vector!

You can also have mixed arrays:

```
> x<-list(1,2,'hello',4,5)
```

R also includes complex number support:

```
> 2*(3+4i)
[1] 6+8i
```

…but it’s not very consistently done:

```
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
```

To compute the square root of `-1` you must specify it as `-1+0i`:

```
> sqrt(-1+0i)
[1] 0+1i
```

Interestingly, R does include support for NaNs, or *Not-a-Number*s, as well as infinity. It also has the concept of a data item whose value is *Not Available*, using the special `NA` symbol:

```
> NA
```

```
[1] NA
> 10*NA
[1] NA
> x<-c(1,2,NA,4,5)
> x
[1]  1  2 NA  4  5
> sum(x)
[1] NA
```

Another important concept is that R supports optional parameters to functions, specified by keyword. In this example we supply a parameter `na.rm` with a value `TRUE` specifying that we want to remove the `NA` values before calculating the sum:

```
> sum(x,na.rm=TRUE)
[1] 12
```

So far, we've only seen simple arrays. R data items can also be arrays with attributes. Let's create an array called a *data frame*:

```
> x<-1:5
> y<-sin(x)
> frame<-data.frame(x,y)
> frame
  x          y
1 1  0.8414710
2 2  0.9092974
3 3  0.1411200
4 4 -0.7568025
5 5 -0.9589243
```

Notice how the rows and columns are labelled. If we look at the attributes of the data frame, we can see how this is done:

```
> attributes(frame)
$names
[1] "x" "y"


$row.names
[1] 1 2 3 4 5


$class
[1] "data.frame"
```

There's nothing magic about attributes. They are just extra bits of data associated with the main array.

Notice in particular the attribute `class` in the frame. It's not a class in a normal object-oriented programming sense, just a character vector. However, many generic R functions use the `class` attribute to decide how to handle the data.

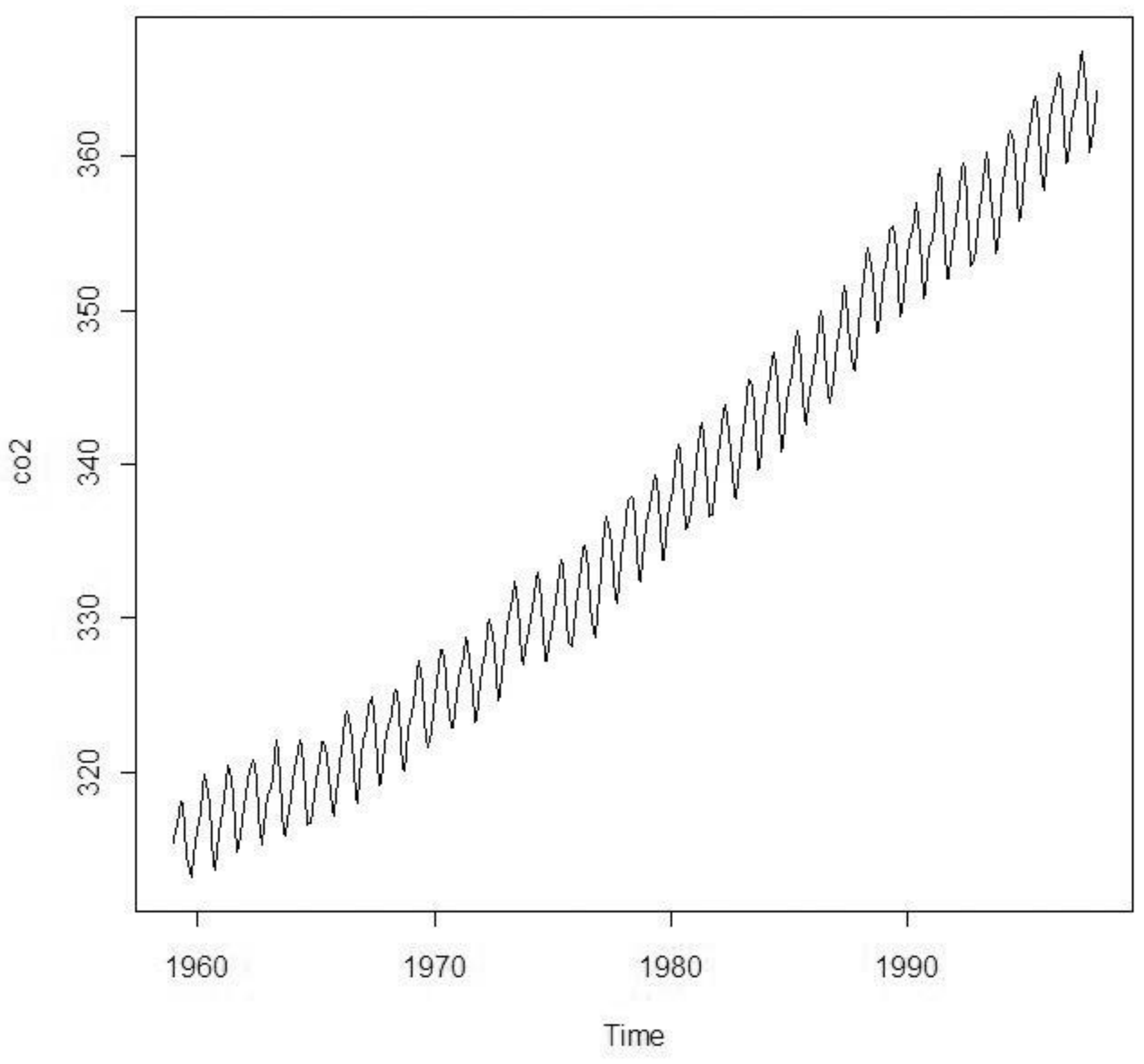
R comes with a number of built-in sample data sets for use in the teaching of statistics. If you want the results of the classic Michaelson-Morley experiment to measure the speed of light, or the survival rates of passengers on the Titanic, it's all here.

Here is some data for monthly atmospheric CO_2 concentrations between 1959 and 1997:

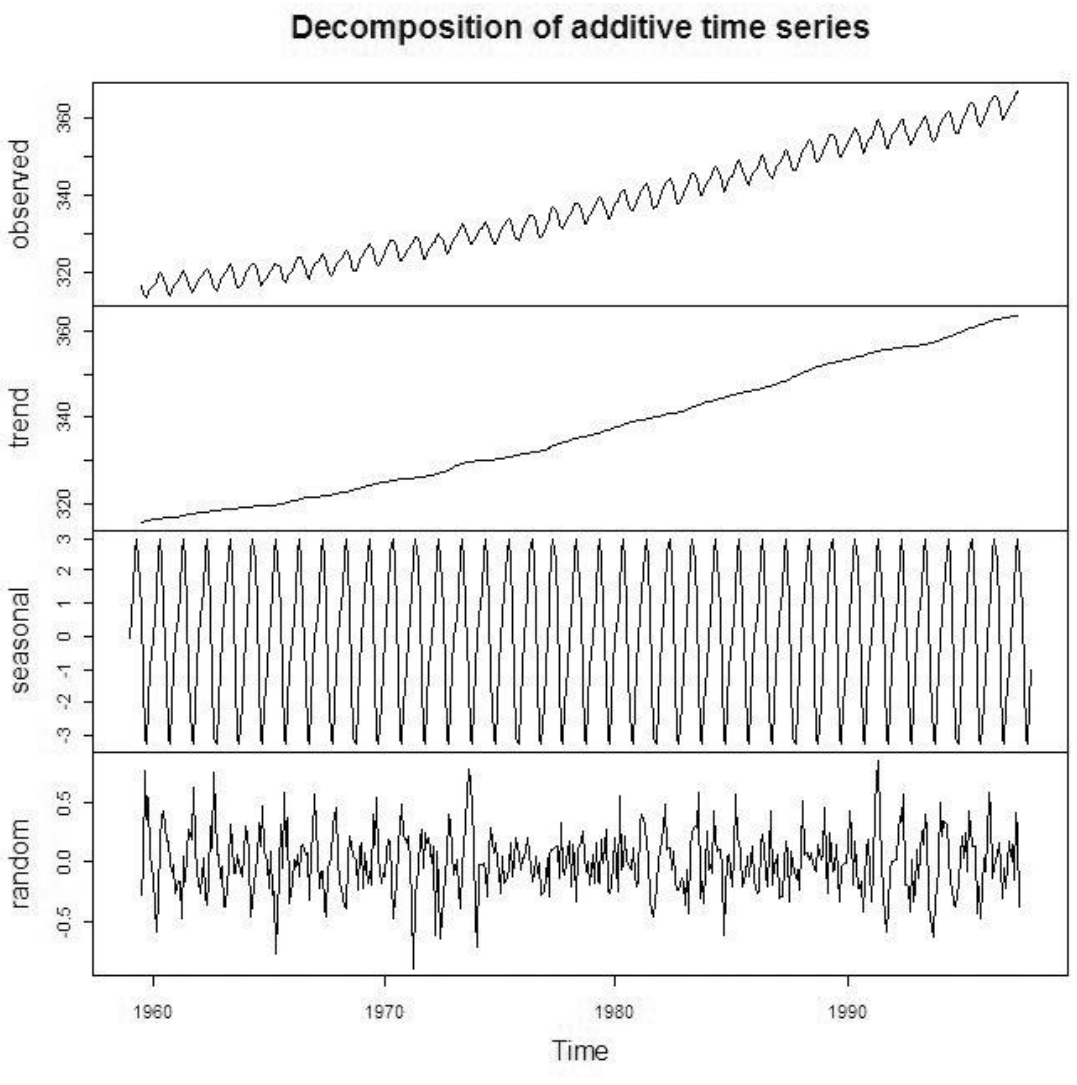
```
> co2
        Jan    Feb    Mar    Apr    May    Jun    Jul    Aug   …
1959 315.42 316.31 316.50 317.56 318.13 318.00 316.39 314.65  …
1960 316.27 316.81 317.42 318.87 319.87 319.43 318.01 315.74  …
…etc

> summary(co2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  313.2   323.5   335.2   337.1   350.3   366.8
```

```
> plot (co2)
```



```
> plot (decompose(co2))
```



Notice that the two plots we just did produced very different results (the second one is a seasonal decomposition of the `co2` data). It's because the `plot` function behaves differently based on the class attribute:

```
> class(co2)

[1] "ts"

> class(decompose(co2))

[1] "decomposed.ts"
```

Earlier work on an R to APL interface

Many of you will have read articles by Professor Alexander Skomorokhov (better known as 'Sasha') describing an earlier interface between R and APL running on Windows, achieved via the COM interface. The method works for Dyalog APL and APLX, and I'll quickly show it here.

What Sasha's work does is to make a bridge between APL and R by using four functions. Here's a short example (we're back in APL now):

```
      RINIT
      'x' RPUT 10?10
      RGET 'x'
7 1 8 3 9 2 6 10 4 5
      1 REXEC 'mean(x)'
5.5
      0 REXEC 'plot(x)'
```

What we've tried to do in APLX version 5 is to build on this work, but make the interface more elegant.

Using R from APLX Version 5

APLX includes the ability to interface to a number of other languages, including .NET, Java, Ruby and now R. This is achieved through a plug-in architecture.

Most of the interface between APLX and R is done using a single external class, named `r`, which represents the R session that you are running. (Note that this is different from most of the other external class interfaces, where objects of many different classes can be created separately from APLX). You create a single instance of this class using `⎕NEW`. R functions (either built-in or loaded from packages) then appear as methods of this object, and R variables as properties of the object.

```
      r←'r' ⎕NEW 'r'
```

We can assign to a variable in the R world, roughly equivalent to Sasha's `RPUT` and `RGET` functions:

```
      r.x←10?10
      r.x
7 1 8 3 9 2 6 10 4 5
```

Next, we also have `⎕EVAL`, similar to Sasha's `REXEC`:

```
      r.⎕EVAL 'mean(x)'
5.5
      r.⎕EVAL 'plot(x)'
[NULL OBJECT]
```

But it's not necessary to keep using an assignment to copy the data across to the R side before we can use it:

```
      r.mean (⊂10?10)
5.5
      r.plot (⊂10?10)
```

Notice that we need to enclose the data to tell R that it's a single argument. We can also do something like:

```
      r.outer (⍳3) (⍳4) '-'
0 ¯1 ¯2 ¯3
1  0 ¯1 ¯2
2  1  0 ¯1
```

…where we're supplying the `outer` function with three separate arguments.

Boxing and un-boxing data

For simple data types like vectors and arrays returned to APL, the data is automatically 'un-boxed'. In other words it's converted to an APL array with the same values.

For more complex cases, the un-boxing is not automatic. Here's the CO_2 data again:

```
      r.co2
[r:ts]
```

What we have here is a *reference* to an item of class `ts`: an R Time-Series.

We can force the reference to be un-boxed:

```
r.co2.⎕VAL
```

…but it’s often more useful to keep it boxed up.

```
      co2←r.co2
      co2.summary.⎕DS
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  313.2   323.5   335.2   337.1   350.3   366.8
```

This last example works because the APLX-R interface treats an expression like

```
       obj.function args
```

…as a synonym for

```
       r.function obj, args
```

…so `co2.summary` is the same as `r.summary co2`. It’s just a bit of syntactic sugar that makes things more readable.

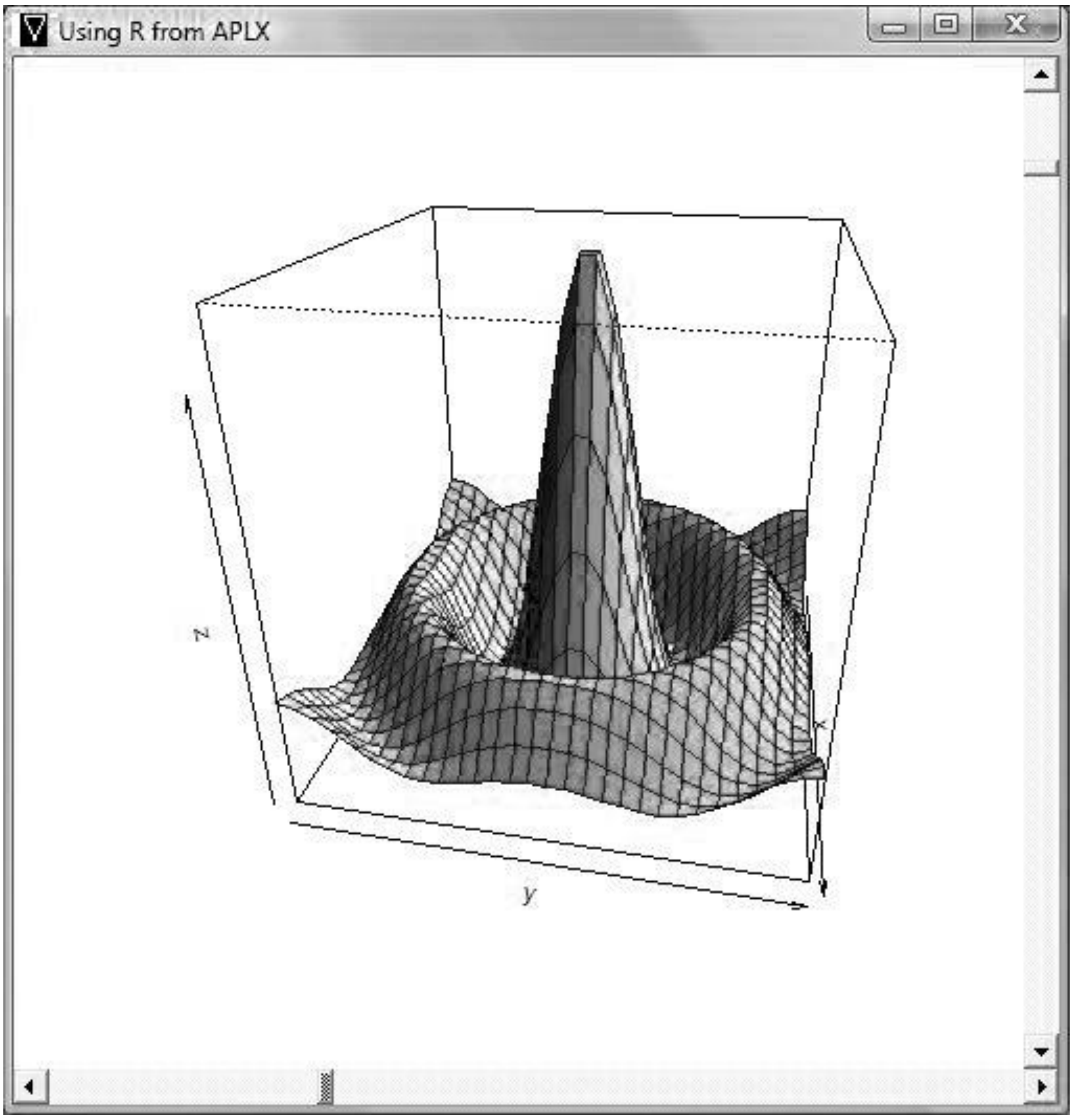
Similarly, we can tell APL not to un-box a value even when it would normally do so:

```
      r.x←⍳10
      r.x
1 2 3 4 5 6 7 8 9 10
      r.x.⎕REF
```

```
[r:integer]
      x←r.x.⎕REF
      x.mean
5.5
```

Graphical example

Now for something a little more interesting. Here’s a little interactive APL program which calls R to plot a 3-D surface and displays the result.



What we’re doing here is using APLX to create the window and manage the scroll bars. It’s calling R to do the 3-D plot and store the result in a bitmap, which APL then displays. As the user drags the scroll bars to change the viewing angle, the graph is rotated.

```
∇ Demo3D;w
⍝ Interactive 3-D charting using R from APLX


⍝ Create the 3D data to plot
r.x←r.⎕EVAL 'seq(-10,10,length=30)'
r.y←r.x
r.z←r.y ∘.Sinc r.x


⍝ Create a window
'⎕' ⎕WI 'scale' 5
w←'⎕' ⎕NEW 'window'
w.title←'Using R from APLX'
w.size←500 500


⍝ Create vertical scroll bar
w.phi.New 'scroll'
w.phi.style←0
w.phi.align←4
w.phi.value←0 360
w.phi.onChange←'Plot3D'
```

```
⍝ Create horizontal scroll bar

w.theta.New 'scroll'

w.theta.style←1

w.theta.align←3

w.theta.value←0 360

w.theta.onChange←'Plot3D'


⍝ Create the picture object

w.pic.New 'picture'

w.pic.align←¯1


⍝ Draw the initial plot

Plot3D


⍝ Handle events until user closes window

:Try

  0 0⍴⎕WE w

:CatchAll

:EndTry

∇


∇ Plot3D;sink
```

```
⍝ Helper function for Demo3D



⍝ Read the scroll bar values to get the rotation theta/phi

r.theta←1↑w.theta.value

r.phi←1↑w.phi.value

⍝

⍝ Tell R to do its plotting to a bitmap, not a graphics window

sink←r.bmp 'C:\\Users\\simon\\Desktop\\persp.bmp'

⍝

⍝ Plot the data

sink←r.⎕EVAL
'persp(x,y,z,theta=theta,phi=phi,shade=0.5,col="green3")'



⍝ Close the bitmap file and display in APL window

sink←r.dev.off

w.pic.bitmap←'C:\Users\simon\Desktop\persp.bmp'

∇



∇ R←X Sinc Y

⍝ Helper function for Demo3D

⍝

⍝ Compute Sinc function

R←(+/(X Y)*2)*0.5
```

```
R←(10×1○R)÷R

∇
```

More details of the R interface in APLX

We can now use R to support complex numbers – something which APLX currently lacks:

```
      comp←r.⎕EVAL '3+4i'
      comp
[r:complex]
      comp.⎕DS
[1] 3+4i

      comp.real
3
      comp.imag
4
```

We can also create new complex numbers with the following syntax…

```
      x←'r' ⎕NEW 'complex' 3 4
      x.⎕DS
[1] 3+4i
```

…and even create arrays of complex numbers:

```
      x← 'r' ⎕NEW 'complex' (2 2⍴(2 3)(6 8)(1 7)(¯1 9))
      x.⎕DS
     [,1]  [,2]
[1,] 2+3i  6+8i
[2,] 1+7i -1+9i
```

```
      x.real
2  6
1 ¯1
      x.imag
3 8
7 9
      x.sqrt.⎕DS
                   [,1]                 [,2]
[1,] 1.674149+0.895977i 2.828427+1.414214i
[2,] 2.008864+1.742278i 2.006911+2.242252i
```

You will remember that R also supports a ‘Not Available’ value. We can also specify this from APL:

```
      NA←'r' ⎕NEW 'NA'
      NA
[r:NA]
      r.sum (⊂1 2 3 NA 5 6)
[r:NA]
```

Sometimes it is necessary to explicitly assign APL data to an R variable so that we can get at it using ⎕EVAL – something which is necessary if we want to call functions which take keyword parameters:

```
      r.x←1 2 3 NA 5 6
      r.⎕EVAL 'sum(x,na.rm=TRUE)'
17
```

Attributes

The APLX interface also includes support for R attributes. Any use of ∆XXX is interpreted as an attribute access.

Here we create a time series and change the `tsp` attribute to say that the values run monthly from January 2005 to May 2009:

```
      series←r.ts (⊂(5+4×12)?1000)
      series.attributes.⎕DS
$tsp
[1]  1 53  1

$class
[1] "ts"

      series.Δtsp←2005 (2009+4÷12) 12
      series.⎕ds
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2005 826 835 727 332 803 634  52  49 161 770  72 525
2006 930 231 216 103 705 712 504 276 718 392 219 251
2007 450 483 865 978 461  63 539  62 762 606 829 354
2008 260 554 453 427 988 969 857 874 342 187 998 657
2009 494 936 707 825 566
```

Climate example

Let me repeat my opening remarks that I'm no statistician, and neither am I a climatologist. However, today is World Ocean Day (*the talk was given on 8 June 2009*), so I thought just for fun we could conclude by looking at some statistics involving sea temperatures.

Let's use some data from the University of East Anglia's Climate Research Unit, quite widely used in the literature. It's available from their web site[2].

As a demonstration of the APLX-R interface, we'll try to reproduce (approximately) the graph of global temperature change.

In particular we'll use a file called `HadCRUT3`, which contains monthly land and sea-surface temperature records since 1850. The surface of the earth is divided into 5-degree-latitude by 5-degree-longitude squares, so the underlying data is a three-dimensional array.

The data is in NetCDF format, a text format for representing scientific data. We could parse it by writing some APL code, but we can get R to do it for us. We load the `ncdf` package and then read the file:

```
       r.library 'ncdf'
 ncdf stats graphics grDevices utils datasets methods base

      nc←r.open.ncdf 'C:\Users\simon\Downloads\HadCRUT3.nc'
      tempdata←nc.get.var.ncdf.⎕REF
      nc.close.ncdf
      tempdata
[r:array]
```

The data contains a lot of Not Available values, particularly for the earlier years, so we'll get R to find the mean temperature for each month by averaging over all the 5×5-degree squares before getting the data into APL. (Note that this might be statistically a bit dubious!)

```
      r.tempdata←tempdata
      averageTemp←r.⎕EVAL 'colMeans(tempdata,na.rm=TRUE,dim=2)'
      ⍴averageTemp
1912
```

We'll create an R Time Series, specifying that the data ranges from January 1850 to March 2009:

```
      tmonth←r.ts (⊂averageTemp)
      tmonth.tsp←1850 (2009+3÷12) 12
```

How many complete years of data do we have?

```
      ⌊tmonth.length÷12
159
```

Let’s get APL to find the mean temperature in each year:

```
      years←159 12⍴ tmonth.⎕VAL
      averagesForYears←(+/years)÷12
```

What was the hottest year?

```
     (1849+⍳159)[⍒averagesForYears]
1998 2005 2003 2002 2007 2004 2006 2001 2008 1997 1999 1995 2000
1990 1991
1988 1981 1944 1994 1983 1996 1989 1987 1993 1980 1973 1943 1938
1939 1992
1953 1977 1979 1962 1986 1878 1963 1941 1940 1937 1961 1958 1984
1945 1985
1982 1942 1957 1959 1969 1978 1970 1952 1967 1960 1934 1931 1975
1972 1932
1936 1930 1951 1877 1948 1968 1947 1926 1946 1971 1935 1949 1966
1954 1974
1889 1965…
```

We can create an R time series of yearly temperatures:

```
      tyear←r.ts (⊂averagesForYears)
      tyear.tsp←1850 2008 1
      tyear.summary
¯0.5757 ¯0.3768 ¯0.2342 ¯0.1655 0.004088 0.5483
      tyear.summary.⎕DS
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-0.575700 -0.376800 -0.234200 -0.165500  0.004088  0.548300
```

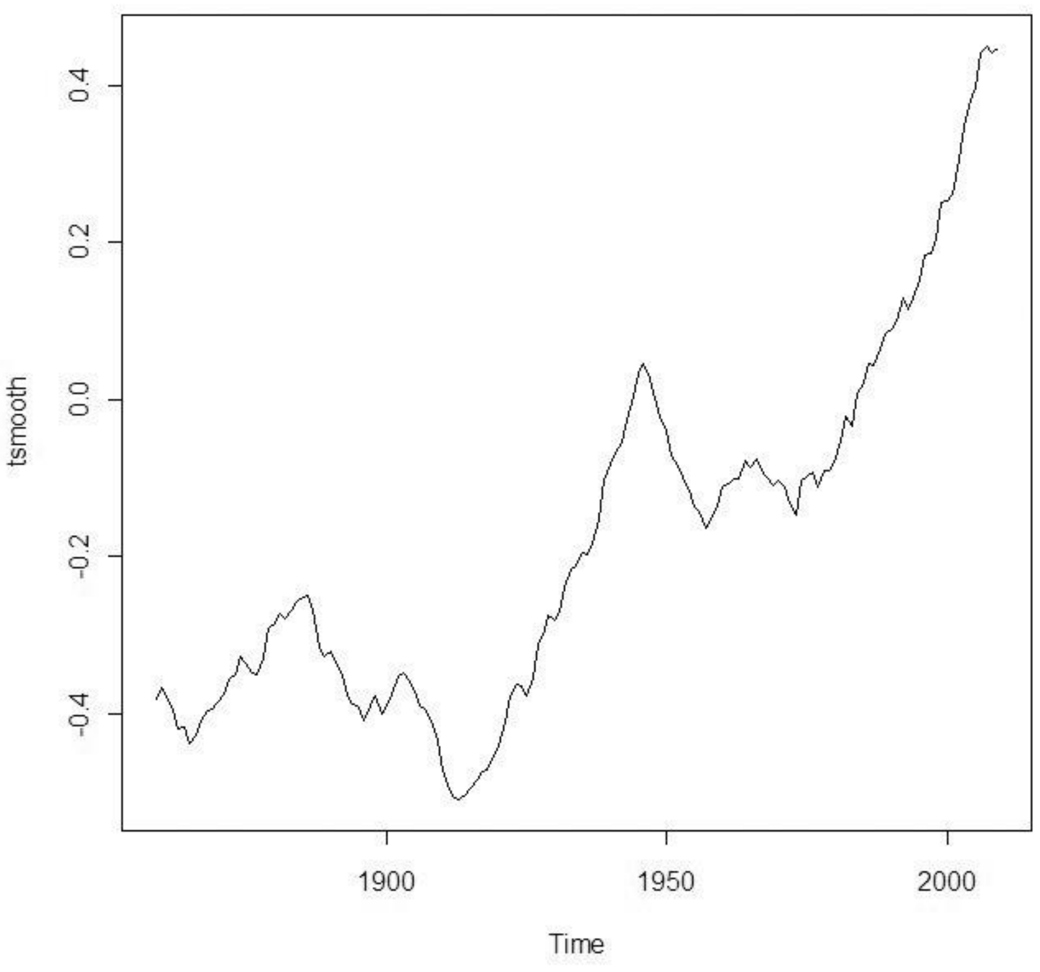
Now let’s use APL to compute a 9-year moving average of the data:

```
      tsmooth←r.ts (⊂9+/tyear.⎕VAL÷9)

      tsmooth.‵tsp←1859 2009 1

      r.tsmooth←tsmooth

      r.⎕EVAL 'plot(tsmooth)'
```

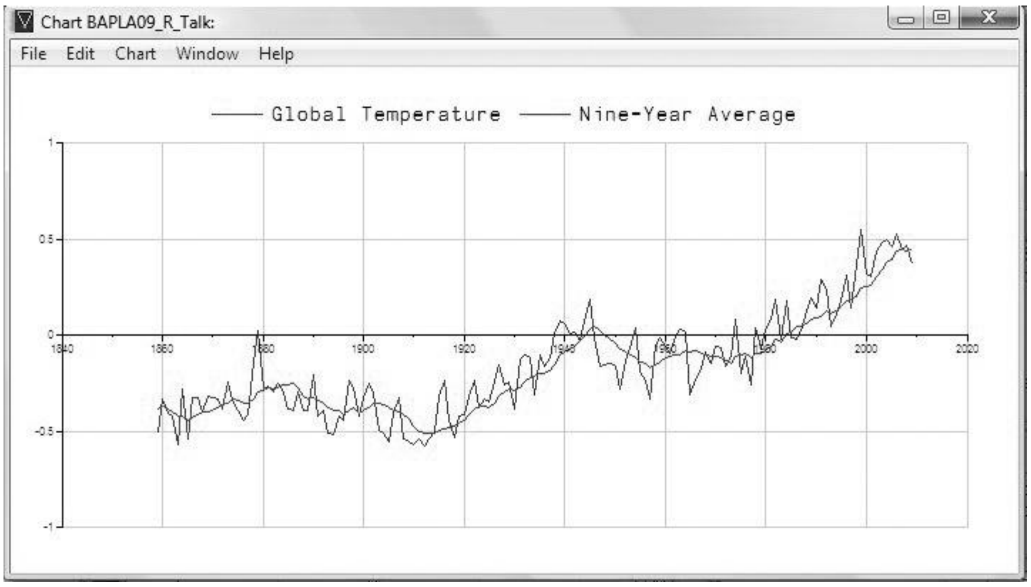


Of course we shouldn't forget that APLX can do charting too:

```
      titles←'seriesname=Global Temperature' 'seriesname=Nine-Year
Average'

      vals←(8↓tyear.⎕val) (tsmooth.⎕VAL) (1858+⍳tsmooth.length)

      titles ⎕CHART ⊃vals
```



Conclusion

There is a huge amount of high-quality software, much of it open-source, much of it free, written in languages like .NET, Java, Ruby and R.

Much of the recent work MicroAPL has done with APLX makes it easy to call code written in these other languages, so all of that software becomes available to the APL programmer.

References

1. *Introduction to R*
   http://cran.r-project.org/doc/manuals/R-intro.pdf
2. University of East Anglia's Climate Research Unit
   http://www.cru.uea.ac.uk/cru/data/temperature/

Inner product – an old/new problem

by Roger Hui (rhui000@shaw.ca)

Abstract

Exploitation of an old algorithm for inner product leads to a greater than 10-fold improvement in speed to `+.×` in Dyalog v12.1. In the process dfns and dops were found to be excellent tools of thought for thinking about the problem. We also demonstrate the product of some large sparse matrices in J.

1. Row-by-column

The conventional method for inner product produces its result one element at a time, operating on one row of the left argument against one column of the right argument. That is, if `z ← x f.g y`, then

```
   z[i;j] = f/ x[i;] g y[;j]
```

Question: why is inner product defined in this computationally inconvenient way?

`A` is a linear transformation from `n` (dimensional) space to `m` space, where `m n←⍴A`, and `B` is a linear transformation from `p` space to `n` space (`n p←⍴B`). The transformations are effected by `A+.×y` and `B+.×y1`. For example:

```
   A←¯3+?3 3⍴10
   B←¯3+?3 3⍴10
   y←¯3+?3⍴10

   B+.×y
17 12 ¯12
   A+.×(B+.×y)
2 58 15
```

The last transformation is the result of transforming `y` first by `B` and then by `A`. The inner product `A+.×B` computes the matrix for this transformation, `A` composed with `B` .

```
   (A+.×B) +.× y
2 58 15
```

```
   A+.×B
¯9 ¯19 ¯2
¯6  16  8
 9 ¯21 21
```

So the answer to the question is, inner product is defined this way to effect the composition of two linear transformations.

2. Row-at-a-time

It turns out that there is an algorithm more efficient than 'row-by-column', namely 'row-at-a-time'. For `z←x f.g y`:

```
   z[i;] = f⌿ x[i;] g[0] y
```

The phrase `x[i;] g[0] y` applies `g` to an element of `x[i;]` against the corresponding row of `y`. For example:

```
   10 100 1000 ×[0] 3 4⍴⍳12
   0   10    20    30
 400  500   600   700
8000 9000 10000 11000
```

In practice, the cumulation `f⌿` is composed and interleaved with the dyadic function `g[0]` so that no additional temporary space is needed. The result for row-at-a-time is mathematically and numerically identical to that for the conventional row-by-column method.

The above description is stated in terms of matrices, but the arguments are applicable to arguments of any rank.

3. Advantages

Row-at-a-time has some efficiency advantages over row-by-column:

- APL arrays are stored in row major order, so that in a column `y[;j]` successive elements are separated from each other and `y[0;j]` can be quite far from `y[n-1;j]`. On modern CPUs with on-chip caches, on large matrices, it is possible for row-by-column to miss the cache on every column, that is, on every element of the inner product result. In contrast, row-at-a-time operates on adjacent memory locations and is 'cache friendly'.

  On 600-by-600 matrices, row-at-a-time is faster than row-by-column by a factor of 10.

  I believe that there is an additional advantage if the right argument is boolean (bits), as indexing a column `y[;j]` of boolean `y` is not trivial.

- Row-at-time makes it practical to exploit zeros in the left argument. ('Zero' in the general sense of the zero in a field with operations `f` and `g`.) In row-at-a-time, in `x[i;] g[0] y`, if `x[i;j]` is zero, there is no need to do `x[i;j] g y[j;]`. In row-by-column, in `x[i;] g y[;j]`, if `x[i;k]` is zero, only `x[i;k] g y[k;j]` can be elided.

  What it means is that you can not afford to spend the time to check `x[i;k]` for zero, because it would slow down the general case. Checking `x[i;j]` for zero in row-at-a-time also slows down the general case, but the cost is amortized over the entire row `y[j;]`.

  On 600-by-600 matrices with a density of 0.5 (i.e. half zeros), row-at-a-time with zero exploitation is faster than row-by-column by a factor of 24.

- Item `j` of the phrase `x[i;] g[0] y` is `x[i;j] g y[j;]`. If the left argument is boolean, this can have only two possible results: `0 g y[j;]` or `1 g y[j;]`. Implementation of this idea in J provides a factor of 5 improvement for boolean inner products.

4. Models

The current row-by-column implementation in Dyalog v12.0 starts by transposing the right argument to move the leading axis to the back, followed by an outer product of the rows. The process can be modelled as follows:

```
   dotc ←{↑ (↓⍺) ∘.(⍺⍺{⍺⍺/⍺ ⍵⍵ ⍵}⍵⍵) ↓(¯1⌽⍳⍴⍴⍵)⍉⍵}

   x←¯500+?13 19 11⍴1000
   y←¯500+?11 23⍴1000

   (x+.×y) ≡ x +dotc× y
1
   (x-.≥y) ≡ x -dotc≥ y
1
```

The expression can be simplified and shortened with judicious use of components:

```
   dotc    ← {↑ (↓⍺) ∘.(⍺⍺{⍺⍺/⍺ ⍵⍵ ⍵}⍵⍵) ↓(¯1⌽⍳⍴⍴⍵)⍉⍵}
   dotc1   ← {↑ (↓⍺) ∘.(⍺⍺{⍺⍺/⍺ ⍵⍵ ⍵}⍵⍵) ↓flip ⍵}
   dotc2   ← {↑ (↓⍺) ∘.(⍺⍺/at ⍵⍵) ↓flip ⍵}
   dotc3   ← {↑ ⍺ ∘.(⍺⍺/at ⍵⍵)compose↓ flip ⍵}
   dotc4   ← {⍺ ∘.(⍺⍺/at ⍵⍵)under↓ flip ⍵}   ⍝ nonce error
   dotc5   ← {⍺ ∘.(⍺⍺/at ⍵⍵)under↓∘flip ⍵}
   dotc6   ← {∘.(⍺⍺/at ⍵⍵)under↓∘flip}       ⍝ nonce error

   flip    ← {(¯1⌽⍳⍴⍴⍵)⍉⍵}
   at      ← {⍺⍺(⍺ ⍵⍵ ⍵)}
   compose← {(⍵⍵ ⍺)⍺⍺(⍵⍵ ⍵)}
   under   ← {(⍵⍵⍣¯1)(⍵⍵ ⍺)⍺⍺(⍵⍵ ⍵)}
```

The transformation from `dotc3` to `dotc4` and from `dotc5` to `dotc6` do not work, but should. The later definitions are shorter than the earlier ones if the metric is word count; alternatively, they are shorter if the compositions were denoted by symbols such as ⍤ and ⍥ instead of `at` and `compose`.

Row-at-a-time can be modelled as follows:

```
dotr  ← {↑ (↓⍺) ⍺⍺{⍺⍺/⍺ ⍵⍵[0] ⍵}⍵⍵¨ ⊂⍵}
dotr1 ← {↑ (↓⍺) ⍺⍺/at(⍵⍵[0])¨ ⊂⍵}
```

In a C implementation of the algorithm, the functions `↑ ↓ ⊂` and the operator `¨` would not performed explicitly. Rather, they would be embodied as `for` loops and other control structures to mediate access to the arguments `x` and `y`.

In J, the `dotc` (row-by-column) and `dotr` (row-at-a-time) operators can be modelled as follows:

```
flip  =: 0&|:
dotcj =: 2 : 'u/@:v"1/ flip'

lr    =: 1 : '1{u b. 0'  NB. left rank
dotrj =: 2 : 'u/@(v"(1+(v lr),_))'
```

In the important special case of rank-0 (scalar) functions, where the left rank is 0, `dotrj` simplifies to:

```
dotrj0 =: 2 : 'u/@(v"1 _)'
dotr1  ← {↑ (↓⍺) ⍺⍺/at(⍵⍵[0])¨ ⊂⍵}
```

`dotr1` is repeated from above. The expression `(↓⍺) f¨ ⊂⍵` means that rows of `⍺` are applied against `⍵` *in toto*, and that is what rank `1 _` does in J. The leading mix `↑` is not needed in J because, not having done the explicit split `↓`, there is no need to undo it.

5. Inner products on sparse arrays

The sparse datatypes in J represent an array by the indices and values of non-'zero' entries. For example:

```
   ] x=: (?.3 4$10) * ?.3 4$2
0 5 9 0
0 9 0 7
0 0 0 0
   ] y=: (?.4 5$10) * ?.4 5$2
0 5 9 0 0
9 0 7 0 0
0 0 3 0 1
0 0 0 0 0

   ] xs=: $. x  NB. convert to sparse
0 1 | 5
0 2 | 9
1 1 | 9
1 3 | 7
   ] ys=: $. y
0 1 | 5
0 2 | 9
1 0 | 9
1 2 | 7
2 2 | 3
2 4 | 1

   x -: xs      NB. match
1
   y -: ys
1
```

Functions apply to sparse arrays, including inner product.

```
   x +/ .* y
```

```
45 0 62 0 9
81 0 63 0 0
 0 0  0 0 0

   xs +/ .* ys
0 0 | 45
0 2 | 62
0 4 |  9
1 0 | 81
1 2 | 63

   (x +/ .* y) -: xs +/ .* ys
1
```

Sparse arrays make possible large inner products which are impractical in the dense domain. In J7.01 beta:

```
   load '\ijs\sparsemm.ijs'

   NB. d sa s - random sparse array with density d and shape s

   x=: 0.0001 sa 2$1e5
   y=: 0.0001 sa 2$1e5

   $ x             NB. shape of x
100000 100000
   */ $ x          NB. # of elements in x
1e10

   z=: x +/ .*y    NB. inner product of x and y
   $ z             NB. shape of z
```

```
100000 100000


   +/ +./"1 x~:0   NB. # of nonzero rows    in x
99995
   +/ +./   y~:0   NB. # of nonzero columns in y
99995
   +/@, z~:0       NB. # of nonzero entries in z
9989822
```

The preceding expressions show that the conventional row-by-column approach would be problematic. There are 99995 non-zero rows in `x` and 99995 non-zero columns in `y`. Consequently, there are roughly 1e10 row-by-column combinations (of which only 1e7 end up being non-zero). No matter how fast each combination is, there are still a lot of them.

Sparse inner product in J uses a row-at-a-time algorithm. The 1e5 by 1e5 inner product above took 3.6 seconds.

L E A R N

Financial math in q

1: Graduation of mortality

by Jan Karman (jkarman@planet.nl)

When I first read about the release of q (kdb+) I was reminded of an article in the Dutch insurance press (1979) by Dr J.A. van der Pool, Actuary A.G., then with IBM, called “Een belegen notitie in een nieuw licht” (A stale note in a new light). He discussed a new approach to the technique in life-insurance companies for changing their base interest rate used for premiums and valuation: APL. And I saw the Light! (Changing the base interest rate for valuation was not a sinecure in those days – the doors were closed for a fortnight, so to speak.)

Over the years around the millennium change I composed a couple of applications in k on the financial and actuarial territory. Reading recently about the free release of q, I thought it an opportunity to revisit my code and write it up for publication.

K4, the underlying code of q, differs from k3, the language I had been using. The principles are similar but the languages are not compatible. And q is designed for large databases – i.e. kdb+. *

Attila Vrabecz helped translate the k3 version into q. I have shown the k3 code here because k3 has GUI features which neatly display the execution of the code.

Introduction

Graduation of mortality is a complex activity, which has had and still has the interest of many demographers and actuaries. Its importance for insurance offices and pension funds is obvious, and may be even greater for scientific research as well. Several methods of graduation have been developed. Here, we focus on graduation by a mathematical formula, i.e. Makeham’s Law (1879).

In all ages people have sought for system in human mortality. Even from the Romans a primitive form of annuity is known. The Dutch Raadspensionaris (Prime Minister and Minister of Finance) Johan de Witt (1625-1672) may be considered the first actuary, writing out life-annuity sheets, based on mathematics. He ‘convinced’ the States General by presenting loudly his “Waerdye van Lyfrenten” (Valuation of Life Annuities) – a famous but unreadable mathematical document – when financing his war against

Britain. Benjamin Gompertz found a “law of mortality” (1825), which served for at least five decades:

$$\mu_x = Bc^x$$

Gompertz’ formula had only to do with the deterioration of life, or rather the resistance to that. Then, in 1879, Makeham added a term reflecting bad chances in life. He found for the force of mortality

$$\mu_x = A + Bc^x$$

From this by integration we get Makeham’s graduation formula:

$$l_x = k.s^x.g^{cx}$$

(in all these formulas x is age and l represents the number of lives exposed to risk at age x).

The objective is to get a smooth series of values. They should be strictly increasing, because it is undesirable for a one-year term insurance to be cheeper for an older person. (But believe me – there’s more to it).

Our task is to find the values of s, g and c from the crude material of the life table, that is right from the observed values. (k is a normalisation factor used to get an initial number in the graduated table of, say, 10,000,000.)

Solution of parameters

Since we have three parameters to solve we take three distinct points in the ‘vector’ of the observed values, at distance h from each other. King-Hardy’s system of equations is a transformation of Makeham’s law for the force of mortality, and reads:

$$\ln \frac{l_{u+h}}{l_u} = \sum_{i=0}^{h-1} \ln p_{u+1} = hA + Bc^u . \frac{c^{h-1}}{c-1} = H_j$$

for u = x, x+h, x+2h and j = 1, 2, 3. After some reduction we find:

$$c^k = \frac{H_3 - H_2}{H_2 - H_1}$$

$$B = \frac{(H_2 - H_1)(c-1)}{c^x(c^k-1)^2}$$

$$A = \frac{1}{h}\left[H_1 - \frac{bc^x(c^k-1)}{c-1}\right]$$

It may be clear that we prefer h as large as possible, in order to get the last part of the table valid. Also, we cannot start at age zero, because child mortality follows a different pattern; but we might try to take this segment as small as possible. Finally it will turn out that these days old-old-age mortality deviates more and more from Makeham's law. It is in fact considerably lower.

This means that we need to provide controls for new factors. Makeham's Law is not timeless, but it served well until about 1965. Then, because of new diseases (e.g cardiovascular) major deviations began in the mid 1960s. These could be repaired by minor adjustments. (See "blending" below.)

Solution in k3, k4 and q

I shall start immediately with the function I designed for it, because it will be clear then, that you can usually type the formulas into the code, whether APL or K, almost literally from the text:

```
f:{[H;x;qx]
   h:+/' _log 1-qx[x+(0 1 2*H)+\:!H]
   c::((h[2]-h[1])%h[1]-h[0])^%H
   A:(-1 _ h) _lsq +(1.0*H),'(c^x,x+H)*((c^H)-1)%c-1
   s::_exp A[0]
   g::_exp A[1]%c-1
   1-s*g^(c^!#qx)*c-1}  / finding parameters i.e. points in curve
                        / by means of King-Hardy's equation system
                        / and calculating qx from s, g, c
```

We need to save c, s and g for later use – the display – so, they have double assignments, in order to make them global. (If one prefers, one can use the parameters even on the Casio).

Attila Vrabecz has been so kind to translate the k3 code into k4, with some comments:

```
f:{[H;x;qx]
    h:+/' log 1-qx[x+(0 1 2*H)+\:!H]
                        /log instead of _log
    c::((h[2]-h[1])%h[1]-h[0])xexp%H
                        /xexp instead of ^, ^ is fill in k4
    A:(-1 _ h) ! +(1.0*H),'(c xexp x,x+H)*((c xexp H)-1)%c-1
                        /! is _lsq, but it only accepts float
matrices
    s::exp A[0]
                        /exp instead of _exp
    g::exp A[1]%c-1
    1-s*g xexp(c xexp!#qx)*c-1}
```

and after that also into q. (Q translation: monadic verbs are exchanged for words, `lsq` instead of `!` – though `!` would still work, as it is a dyad. Also, semicolons at line ends.)

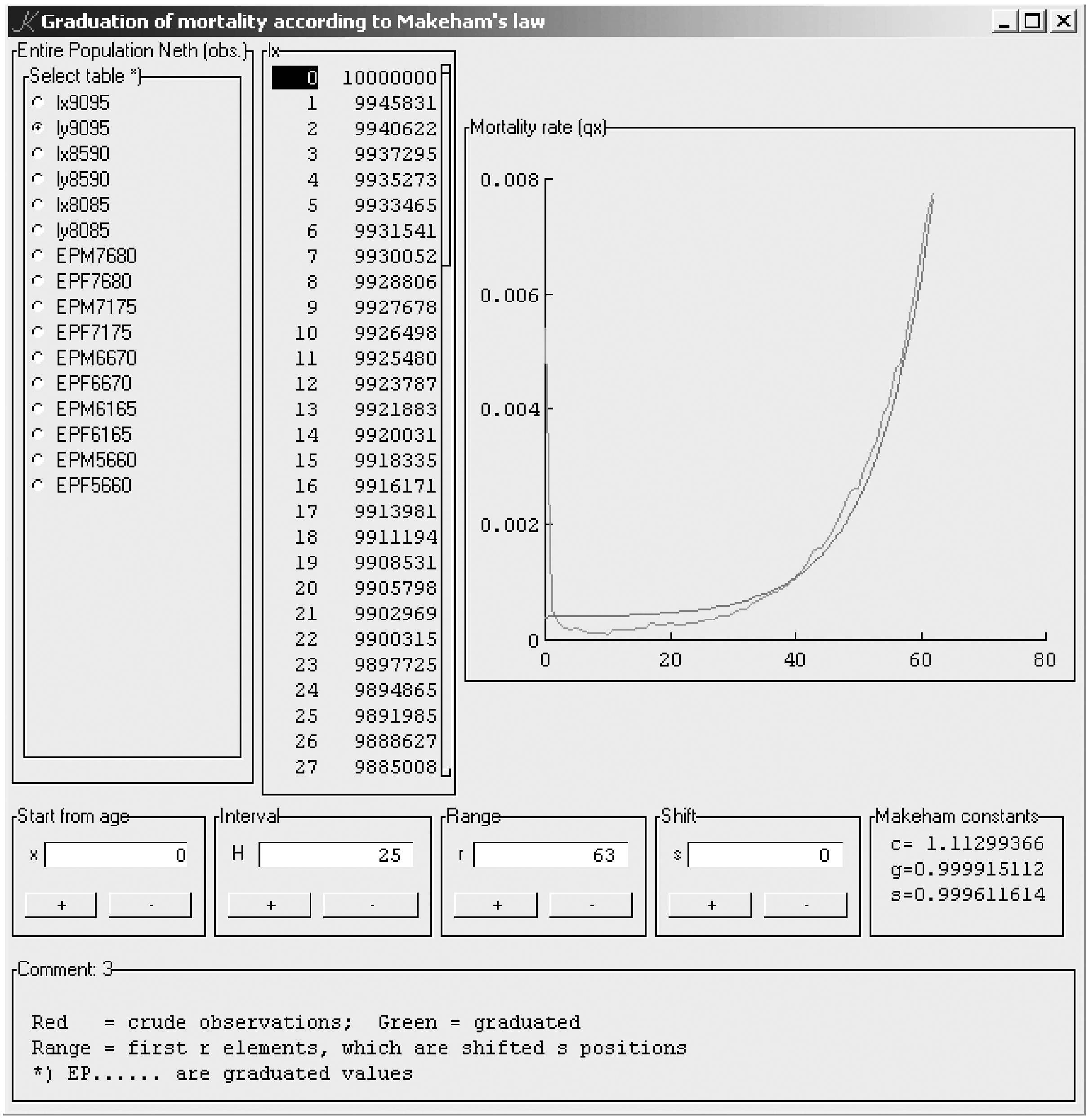
```
f:{[H;x;qx]
    h:sum each log 1-qx[x+(0 1 2*H)+\:til H];
    c::((h[2]-h[1])%h[1]-h[0])xexp reciprocal H;
    A:(-1 _ h) lsq flip(1.0*H),'(c xexp x,x+H)*((c xexp H)-1)%c-1;
    s::exp A[0];
    g::exp A[1]%c-1;
    1-s*g xexp(c xexp til count qx)*c-1}
```

Controls

There are four controls: starting age, interval (h), range of the observed data and a shift. The shift is nice in the graphical representation, because it helps you to look at differences in behaviour of the mortality in different segments of the entire table. The coding of the formulas is the easiest part. The GUI controls are quite different and they need to determine starting points and steps. For an example we take the interval h.

```
\d .k.I
H:25                                      /_ (#.k.qx)%4
                                          / interval in mortality
table
incH:"H:(H+1)&_((#.k.qx)-0|.k.A.x)%4"
decH:"H:7|H-1"                            / decrement interval
incH..c:`button; decH..c:`button          / display class
incH..l:"+";decH..l:"-"
H..d:"_((#.k.qx)-0|.k.A.x)%4"
.k.I..l:"Interval"
```

Picture


Graduation of mortality according to Makeham's law

Entire Population Neth (obs.)

Select table *)

- lx9095
- [✓ ly9095]
- lx8590
- ly8590
- lx8085
- ly8085
- EPM7680
- EPF7680
- EPM7175
- EPF7175
- EPM6670
- EPF6670
- EPM6165
- EPF6165
- EPM5660
- EPF5660

	lx
0	10000000
1	9945831
2	9940622
3	9937295
4	9935273
5	9933465
6	9931541
7	9930052
8	9928806
9	9927678
10	9926498
11	9925480
12	9923787
13	9921883
14	9920031
15	9918335
16	9916171
17	9913981
18	9911194
19	9908531
20	9905798
21	9902969
22	9900315
23	9897725
24	9894865
25	9891985
26	9888627
27	9885008

Start from age: x 0

Interval: H 25

Range: r 63

Shift: s 0

Makeham constants:

```
c= 1.11299366
g=0.999915112
s=0.999611614
```

Comment: 3

```
Red   = crude observations;  Green = graduated
Range = first r elements, which are shifted s positions
*) EP...... are graduated values
```

Fig. 1. Left-most, the names of the table files with the observed values, the lx-column with the graduated values and the graph. Next line, the four controls, and right-most, the parameters. (No, we have no spinboxes.) In the bottom line some comments.

Final remarks and question

Makeham's Law is under fire these days because of the deviations, particularly in the upper part of the life-table, but is still in wide use. In practice, you could use different formulas for different segments and glue the parts together afterwards. That's what is called "blending".

Q: "Why so complicated? Can't you just look up those figures in a table?"
A: "This is exactly how those tables are made."

The complete application is available online and can be downloaded freely from www.ganuenta.com/mortality.exe.

Acknowledgments

Repeatedly, I doubted whether I would finish this. But the encouragement of Stephen Taylor and Attila Vrabecz kept me at it. More, Attila generously offered to translate some essential parts of the code into q/k4. (The sociable atmosphere of the k community has not changed – hooray!). Many thanks too to Stevan Apter for referring me to the k4 listbox, where I met Attila.

Bibliography

1. Benjamin, B. & Pollard J.H., *The Analysis of Mortality and other Actuarial Statistics*, 1980, Heinemann, London
2. Zwinggi, E., *Versicherungsmathematik*, 1952, Birkhäuser Verlag, Basel

* Note by Stevan Apter

I think this is wrong.

Let's separate a few things here:

1. k4 - the language
2. Kx System's market objectives

I have nothing to say about (2).

K4 is a general programming language, no less than k3. It adds support for 'database applications' by rationalising the table and keytable datatype, adding two primitives which generalize SQL queries (`?` and `!`), and extending the atomic and list primitives (e.g. `+` and `#:`) to operate on these new types.

I think the evolution of k from k3 to k4 is consistent with Arthur [Whitney]'s lifelong intellectual commitments: identify some important computational concept, reduce it to its fundamentals, stripped of accidental and historical anomalies, figure out how it fits in with the current generation of k, then model the concept (kdb, the set of k3 functions to ‘do database’), and finally, implement the next k, which incorporates the new concept.

‘K’ is k4. Q is a thin syntax layer on k, implemented in the primitive parsing and interpretation machinery of k. The ‘q language’ is nothing more than this: replace monadic primitives with keywords. That is, q has no ambivalence. The phrase `#a--b` becomes `count a-neg b`. K and q are semantically identical. K and q have the same performance characteristics.

Kdb+ is k/q plus all the other stuff database people expect – ODBC, admin tools, etc. At least, that’s how I use the term, and to my knowledge, Kx has never offered a more precise definition.

I continue to use ‘k’ to refer to everything Kx offers, most generally to denote the whole research programme stretching from k1 to k4 and on into the future. K is just Arthur’s brain, past, present, and future.

Generating combinations in J efficiently

by R.E. Boss

The x combinations of y are the unordered x-sets of `i.y` and their number is `x!y`. For years the generation of combinations in J was done most efficiently by the verb `comb` on the `for.` page of the dictionary [1]. In this article a few ways of generating combinations in J will be considered and the performances are compared. A new and more efficient algorithm is presented.

The original, comb

See also [2] for the definition.

```
comb=: 4 : 0
 k=. i.>:d=.y-x
 z=. (d$<i.0 0),<i.1 0
 for. i.x do. z=. k ,.&.> ,&.>/\. >:&.> z end.
 ; z
)
```

At some point in the `for.`-loop, `z` represents the combinations boxed according to the starting number, like in the next figure (where `k = 0 1 2`).

```
+---+---+---+
|0 1|1 2|2 3|
|0 2|1 3|   |
|0 3|   |   |
+---+---+---+
```

Then the expression `k ,.&.> ,&.>/\. >:&.> z` first increases all numbers by one,

```
    >:&.> z
```

```
+---+---+---+
|1 2|2 3|3 4|
|1 3|2 4|   |
|1 4|   |   |
+---+---+---+
```

then rearranges the boxes by

```
    ,&.>/\. >:&.> z
+---+---+---+
|1 2|2 3|3 4|
|1 3|2 4|   |
|1 4|3 4|   |
|2 3|   |   |
|2 4|   |   |
|3 4|   |   |
+---+---+---+
```

and finally stitches `k` to it:

```
    k ,.&.> ,&.>/\. >:&.> z
+-----+-----+-----+
|0 1 2|1 2 3|2 3 4|
|0 1 3|1 2 4|     |
|0 1 4|1 3 4|     |
|0 2 3|     |     |
|0 2 4|     |     |
|0 3 4|     |     |
+-----+-----+-----+
```

As we can see, a lot of boxing is going on. A few remarks on that. We could get rid of the `>:&.> z` which would lead to

```
    k ,.&.> ,&.>/\. z
+-----+-----+-----+
|0 0 0|1 1 1|2 2 2|
|0 0 1|1 1 2|     |
|0 0 2|1 2 2|     |
|0 1 1|     |     |
|0 1 2|     |     |
|0 2 2|     |     |
+-----+-----+-----+
```

and replace the `; z` at the end by `(i.x)+"0 ; z` However this would worsen the efficiency more than a bit.

On the other hand, one could combine the `,` and `>:` by `>:@,`

This would lead to `k ,.&.> >:@,&.>/\. z` and, although this is somewhat leaner, it is also slower.

ComB1, the first improvement

We can do better than `comb` if we look at the numbers in the different columns. The next verb uses this. It is a slight modification of the `comb1` in [3].

```
comB1=: 4 : 0
 d=. x-~y
 k=. |.(>:d),\i.y
 z=. (d$<i.0 0),<i.1 0
 for_j. k  do. z=. j ,.&.> ,&.>/\. z end.
 ; z
)
```

For `'x y' =: 3 5` we get

```
   k=.|.(>:2),\i.5
2 3 4
1 2 3
0 1 2
```

and these rows are stitched subsequently to the prefixed boxes of `z` as is shown below. After the first two rows of `k` are used, we get `z` equal to

```
+---+---+---+
|1 2|2 3|3 4|
|1 3|2 4|   |
|1 4|   |   |
+---+---+---+
```

And just like the original, the next row is stitched by `j ,.&.>,&.>/\. z`

This new method at least made the boxing `>:&.> z` superfluous. So it is no surprise that it was faster and leaner by some 10% to 30%:

```
    rnk2 5&ts&>'x comb y'; 'x comB1 y'['x y'=: 8 24
1   1.07 1.13     0.172 7.915e7
0   1.00 1.00     0.161 6.998e7

    rnk2 5&ts&>'x comb y'; 'x comB1 y'['x y'=: 12 24
1   1.20 1.17     1.276 4.513e8
0   1.00 1.00     1.067 3.844e8

    rnk2 5&ts&>'x comb y'; 'x comB1 y'['x y'=: 16 24
1   1.28 1.30     0.644 2.069e8
0   1.00 1.00     0.504 1.594e8
```

The first column of the performance matrix gives the ranking, the 2nd column gives the relative execution time, the 3rd column the relative execution space and the last two columns give the real time and space [4].

ComB2, second improvement

In [5] – where it was called `comb2` – a further improvement on the generation of combinations is given, now even by a complete tacit and rather elegant verb.

```
   comB2=: [:; [:(,.&.><@;\.)/ >:@-~ [\i.@]
```

This verb starts with `>:@-~[\i.@]` by which a matrix is created in which each row contains the numbers which are used in the corresponding column. The matrix

```
      3(>:@-~ [\i.@]) 5
0 1 2
1 2 3
2 3 4
```

is used for the combinations

```
   3 comB2 5
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

The first row of the first matrix gives the numbers used in the first column of the combinations matrix, and likewise for the second and third row and column. From

```
   (,.&.><@;\.)/_2{. 3(>:@-~ [\i.@]) 5
```

```
+---+---+---+
|1 2|2 3|3 4|
|1 3|2 4|   |
|1 4|   |   |
+---+---+---+
```

and

```
   (,.&.><@;\.)/ 3(>:@-~ [\i.@]) 5
+-----+-----+-----+
|0 1 2|1 2 3|2 3 4|
|0 1 3|1 2 4|     |
|0 1 4|1 3 4|     |
|0 2 3|     |     |
|0 2 4|     |     |
|0 3 4|     |     |
+-----+-----+-----+
```

it is obvious what is happening. This verb `comB2` could be regarded as finishing the first improvement, which introduced the `>:@-~ [\i.@]` phrase.

Notice also that the `,&.>/\.` from the first two verbs is replaced by the `<@;\.` phrase. Although this alternative is slower for large numbers, it is of no influence for the relatively small numbers used in combinations.

If the performances are compared, we get

```
    rnk2 5&ts&>'x comb y'; 'x comB1 y'; 'x comB2 y'['x y'=: 8 24
2   1.00 1.13     0.156 7.915e7
0   1.01 1.00     0.158 6.998e7
1   1.02 1.00     0.159 6.997e7

    rnk2 5&ts&>'x comb y'; 'x comB1 y'; 'x comB2 y'['x y'=: 12 24
```

```
2    1.19 1.17      1.277 4.513e8
0    1.00 1.00      1.071 3.844e8
1    1.00 1.00      1.073 3.844e8

    rnk2 5&ts&>'x comb y'; 'x comB1 y'; 'x comB2 y'['x y'=: 16 24
2    1.25 1.30      0.642 2.069e8
1    1.06 1.00      0.543 1.594e8
0    1.00 1.00      0.513 1.594e8
```

which makes `comB2` the overall champion, be it close to `comB1`, and almost 50% more efficient as the original.

A remark must be made that for the trivial combinations with `x e. 0 1` the answer of `comB2` is incorrect. For that the verb should be adjusted to

```
   comB2`((!,[)$ i.@])@.(0 1 e.~[)
```

which hardly influences the performance. The conclusion is that the performances of `comB1` and `comB2` are more or less equal.

New improvement: combinations combined

What is really annoying is the fact that although `k comb n` and `(n-k) comb n` are mutually complementary, for relatively small `k` the calculation of `k comb n` is much more efficient than that of `(n-k) comb n`. See the next figures.

```
   rnk2 5&ts&>'8 comB2 24'; '16 comB2 24'
0    1.00 1.00      0.141 6.997e7
1    3.57 2.28      0.503 1.594e8
```

Also taking the complement is of no use, performance wise:

```
   rnk 5&ts&>'8 comB2 24'; '(i.24)-."1 [8 comB2 24'
0 1.00 1.00
```

```
1 5.66 1.44
```

So when writing this article an old idea came up again to reconstruct a combination from two (or more) simpler combinations.

The questions than are: which combinations can be combined and how to combine them? To answer the first question, let’s have a look at splitting a given combination.

```
   4 comb 6
0 1 2 3
0 1 2 4
0 1 2 5
0 1 3 4
0 1 3 5
0 1 4 5
0 2 3 4
0 2 3 5
0 2 4 5
0 3 4 5
1 2 3 4
1 2 3 5
1 2 4 5
1 3 4 5
2 3 4 5
```

If we subtract 2 from the last two columns we get

```
   0 0 2 2 -~"1[4 comb 6
0 1 0 1
0 1 0 2
0 1 0 3
0 1 1 2
0 1 1 3
```

```
0 1 2 3
0 2 1 2
0 2 1 3
0 2 2 3
0 3 2 3
1 2 1 2
1 2 1 3
1 2 2 3
1 3 2 3
2 3 2 3
```

Boxing according to the first two columns produces

```
   A =: (2&{."1</.])0 0 2 2 -~"1[4 comb 6
   A
+-------+-------+-------+-------+-------+-------+
|0 1 0 1|0 2 1 2|0 3 2 3|1 2 1 2|1 3 2 3|2 3 2 3|
|0 1 0 2|0 2 1 3|       |1 2 1 3|       |       |
|0 1 0 3|0 2 2 3|       |1 2 2 3|       |       |
|0 1 1 2|       |       |       |       |       |
|0 1 1 3|       |       |       |       |       |
|0 1 2 3|       |       |       |       |       |
+-------+-------+-------+-------+-------+-------+
```

As we can see `4 comb 6` appears to be built from two times `2 comb 4`. First of all, the last 2 columns of the boxes of `A` appear to be

```
   <@;\.({."1</.]) 2 comb 4
+---+---+---+
|0 1|1 2|2 3|
|0 2|1 3|   |
|0 3|2 3|   |
|1 2|   |   |
|1 3|   |   |
|2 3|   |   |
+---+---+---+
```

whereas the first columns in the boxes of A are the rank-1 boxes of `2 comb 4`

```
   <"1[2 comb 4
+---+---+---+---+---+---+
|0 1|0 2|0 3|1 2|1 3|2 3|
+---+---+---+---+---+---+
```

So A can be constructed by

```
(<"1[2 comb 4),”1&.><@;\.({."1</.]) 2 comb 4
```

Generalizing this idea we come to the verb

```
comB3=: 4 : 0
  if. x e. 0 1 do. z=.<((x!y),x)$ i.y
  else. t=. |.(<.@-:)^:(i.<. 2^.x)x
    z=.({.t) ([:(,.&.><@;\.)/ >:@-~[\i.@]) ({.t)+y-x
    for_j. 2[\t do.
z=.([ ;@:(<"1@[ (,"1 ({.j)+])&.> ])&.> <@;\.({&.><)~ (1+({.j)-
~{:"1)&.>) z
      if. 2|{:j do. z=.(i.1+y-x)(,.>:)&.> <@;\.z end.
    end.
  end.
  ;z
)
```

We will illustrate the verb in calculating `5 comB3 7`. We start with a check for `x` equal to `0` or `1`. For other `x` we floor and halve (`<.@-:`) until `2` or `3` is reached, for `x = 5` this gives `t = 2 5`. Then the combinations `2 comb 4` are determined, such that they are boxed according to the first number:

```
   2 ([:(,.&.><@;\.)/ >:@-~[\i.@]) 4
+---+---+---+
|0 1|1 2|2 3|
|0 2|1 3|   |
|0 3|   |   |
+---+---+---+
```

This property will be held invariant for all `z` to be computed.

Given such a `z`, we will combine it with itself as described before, which is written in J as given in the long line of the verb.

This long line consists of three parts, the left part (`[`) being `z` itself. The right-hand side `<@;\.({&.><)~ (1+({.j)-~{:"1)&.>` is a bit complicated. Let's split it first according to

```
   (<@;\. ,&< (1+({.j)-~{:"1)&.>) z
+-------------+-------------+
|+---+---+---+|+-----+---+-+|
||0 1|1 2|2 3|||0 1 2|1 2|2||
||0 2|1 3|   ||+-----+---+-+|
||0 3|2 3|   ||             |
||1 2|   |   ||             |
||1 3|   |   ||             |
||2 3|   |   ||             |
|+---+---+---+|             |
+-------------+-------------+
```

So we see at the left the boxed outfix of `z`, and at the right, the last columns of the boxes of `z`, diminished by `1`. The last are used to select from the first delivering

```
   B =: (<@;\. ({&.><)~ (1+({.j)-~{:"1)&.>) z
   B
+------------+--------+-----+
|+---+---+---+|+---+---+|+---+|
||0 1|1 2|2 3|||1 2|2 3|||2 3||
||0 2|1 3|   |||1 3|   ||+---+|
||0 3|2 3|   |||2 3|   ||     |
||1 2|   |   ||+---+---+|     |
||1 3|   |   ||         |     |
||2 3|   |   ||         |     |
|+---+---+---+|         |     |
+------------+--------+-----+
```

Now we come to the central part of the long line

```
   ;@:(<"1@[(,"1 ({.j)+])&.> ])&.>
```

Each box of `z` is combined with each box of `B`, by concatenating a row in a box left with a sub-box in a box of `B`. The last one has to be increased by `2 = ({,j)` and the end result should be ravelled. Then the new `z` is again boxed according to its first column.

```
   ([ ;@:(<"1@[ (,"1 ({.j)+])&.> ])&.> <@;\.({&.><)~ (1+({.j)-
~{:"1)&.>) z
+-------+------+-------+
|0 1 2 3|1 2 3 4|2 3 4 5|
|0 1 2 4|1 2 3 5|       |
|0 1 2 5|1 2 4 5|       |
|0 1 3 4|1 3 4 5|       |
|0 1 3 5|       |       |
|0 1 4 5|       |       |
|0 2 3 4|       |       |
|0 2 3 5|       |       |
|0 2 4 5|       |       |
|0 3 4 5|       |       |
+-------+-------+-------+
```

Since `2|{:j` because `5={:j` an extra column has to be added as in the other verbs. And this is also the final result, be it in an unravelled way.

Now, how about the performances?

```
    rnk2 5&ts&>'x comb y'; 'x comB2 y'; 'x comB3 y'['x y'=: 8 24
2   1.14 1.24     0.157 7.915e7
1   1.15 1.09     0.158 6.997e7
0   1.00 1.00     0.138 6.396e7

    rnk2 5&ts&>'x comb y'; 'x comB2 y'; 'x comB3 y'['x y'=: 12 24
2   1.76 1.42     1.277 4.513e8
1   1.48 1.21     1.074 3.844e8
0   1.00 1.00     0.726 3.182e8

    rnk2 5&ts&>'x comb y'; 'x comB2 y'; 'x comB3 y'['x y'=: 16 24
2   2.24 1.68     0.644 2.069e8
1   1.75 1.30     0.504 1.594e8
0   1.00 1.00     0.287 1.230e8
```

which is a major difference, even with the last record holder. And as it should be, the difference is bigger in the cases where `x` is close(r) to `y`.

Some other figures are:

```
   rnk 5&ts&>'x comb y'; 'x comB2 y'; 'x comB3 y'['x y'=: 20 26
2 2.88 2.64
1 2.10 1.98
0 1.00 1.00
```

```
   rnk 5&ts&>'x comb y'; 'x comB2 y'; 'x comB3 y'['x y'=: 20 28
|out of memory: comb
|   z=.k    ,.&.>,&.>/\.>:&.>z


   rnk 5&ts&> 'x comB2 y'; 'x comB3 y'['x y'=: 20 28
1 1.94 1.55
0 1.00 1.00
```

So finally a new way of generating combinations in J efficiently is found and described.

Epilogue

One could ask the use of spending so much time speeding up a process that takes seconds or less? Even twice as fast – who cares?

There are other reasons to look for efficiency.

First of all, it is fun to do. Just as trying to break a world record, it costs you sweat, time, pain and money, and is a uniquely rewarding experience when you succeed. "Why did you climb that mountain?" a mountaineer was asked. "Because it was there." Was the answer. So performances have to be improved, just like records.

The second and by no means less important answer is that if you design an algorithm which is faster and leaner than the existing ones, you have understood the problem and its structure deeply. And often that algorithm reveals that structure in using it. This is very rewarding, to see patterns unseen by others.

Apart from that, J is a language that makes it easy to examine most of the structures you discover. So, as long as there are nice problems for which performance can be improved, I'll keep on doing it.

Notes

1. www.jsoftware.com/help/dictionary/cfor.htm
2. www.jsoftware.com/jwiki/Essays/Combinations
3. www.jsoftware.com/pipermail/programming/2007-August/007819.html
4. `rnk2=: 1 7j2 5j2 10j3 8j_3 ":`
   `(,.~[: (/:@/:@:({."1),. }."1) (%"1 <./)@:(,.~ */"1))`
5. www.jsoftware.com/pipermail/programming/2008-January/009488.html

Ten divisions to Easter

Ray Polivka (polivkar@juno.com)

See also Ray Cannon's article "When is Easter" in *Vector* 10.3. *Ed.*

No doubt many of you have heard of the problem of determining the calendar date of Easter. Perhaps you have seen a program solving this problem or have written one.

While browsing in one of my mathematical puzzle books, I found this intriguing solution. It is an unambiguous algorithm, involving only ten divisions, that produces the proper date for Easter given only an integer representing the year for which you wish it. This intriguing algorithm is described by T.H. O'Beirne in his book entitled *Puzzles and Paradoxes*.[1] He described the algorithm in great detail. This is preceded by a lengthy and fascinating history of the calculation of Easter. The basic difficulties in computing the date of Easter occur because of the discrepancies between the ecclesiastical needs and astronomical data.

This ten-division algorithm is organized as a table. In presenting this table, O'Beirne stated "we shall give a purely number rule – subject to no exceptions of any kind – which calculates the date of Easter from a knowledge only of the year number, with no reference to any other rules or tables." He went on to state his tabular scheme is based on a correct procedure published in *Nature* in 1876 by an anonymous New York author. He then said "Based on this procedure, we have developed similar methods of our own, which are now published for the first time." Here is the table he gave.

First Ten-Division Table

Step	Dividend	Divisor	Quotient	Remainder
1.	x	19	–	a
2.	x	100	b	c
3.	b	4	d	e
4.	8b+13	25	g	–
5.	19a+b-d-g+15	30	–	h

First Ten-Division Table

Step	Dividend	Divisor	Quotient	Remainder
6.	a+11h	319	u	–
7.	c	4	i	k
8.	2e+2i-k-h+u+32	7	–	q
9.	h-u+q+90	25	n	–
10.	h-u+q+n+19	32	–	p

In the *x*th year A.D. of the Gregorian calendar, Easter Sunday is the *p*th day of the *n*th month. The date of the Easter full moon is obtained if -1 is substituted for q in steps 9 and 10.

O'Beirne also included a second table in which he felt computation may be easier. This is the second table.

Second Ten-Division Table

Step	Dividend	Divisor	Quotient	Remainder
1.	x	100	b	c
2.	5b+c	19	–	a
3.	3(b+25)	4	d	e
4.	8(b+11)	25	m	–
5.	19a+d-m	30	–	h
6.	a+11h	319	n	–
7.	60(5-e)+c	4	j	k

Second Ten-Division Table

Step	**Dividend**	**Divisor**	**Quotient**	**Remainder**
8.	2j-k-h+n	7	–	w
9.	h-n+w+110	30	n	q
10.	q+5-n	32	–	p

Again, in the *x*th year A.D. of the Gregorian calendar, Easter Sunday is the *p*th day of the *n*th month.

This is a fine problem for someone learning APL. It is a nice illustration of *encode*. The problem could be stated as:

> Create a function `easter` that, given the year, will calculate the date of Easter using the second Ten Division Table.

```
      easter 2009
Easter is April 12, 2009
```

Here is a solution using the first table.

```
∇ z←easter y;⎕ML;a;b;c;d;e;g;h;i;k;l;m;n;p;q;u
 ⍝ y: year as an integer
 ⍝ z: date of Easter
  ⎕ML←3
  a←19⊤y
  (b c)←0 100⊤y
  (d e)←0 4⊤b
  g←↑0 25⊤13+8×b
  h←30⊤(19×a)+b+15-d+g
  u←↑0 319⊤a+11×h
  (i k)←0 4⊤c
  q←7⊤(2×e+i)+u+32-k+h
  n←↑0 25⊤h+90+q-u
  p←32⊤h+q+n+19-u
  z←'Easter is ',(↑(n=4)⌽'March' 'April'),' ',(⍕p),', ',⍕y
∇
```

References

15. O'Beirne, T.H., *Puzzles and Paradoxes*, Oxford University Press, 1965, pages 168-184

J-ottings 52

All but one

by Norman Thomson

A frequent requirement in applied mathematics and statistics is to evaluate sums and products omitting just one variable or dimension. The notion of ‘all but one’ can be interpreted in two ways, depending whether the ‘one’ is to be systematically omitted, or obtained by a merge with an existing dimension, that is by reduction.

Retaining all but one

As an example of the first case, adding or multiplying ‘all but one’ items in a list progressively can be done by using the hook `invf~f/` which means `(f/x)invf x`, where `invf` is the inverse verb of `f`, for example:

```
   i.5
0 1 2 3 4
   (-~+/)i.5          NB. sum five items omitting one at a time
10 9 8 7 6
   (%~*/)1+i.5        NB. multiply five items omitting one at a time
120 60 40 30 24
```

This extends readily to objects of higher rank:

```
    ]b=.i.3 4
0 1  2  3
4 5  6  7
8 9 10 11
   (-"1~+/)b          NB. sums of all rows but one
12 14 16 18
 8 10 12 14
 4  6  8 10
   (-"2~+/"1)b         NB. sums of all columns but one
 6  5  4  3
18 17 16 15
30 29 28 27
```

The rule is that the rank operand for the *inverse* verb should be one greater than that of the inserted verb.

Generalising ‘retaining all but one’

Another tool which deals with ‘retaining all but one’ is the *suffix* adverb which eliminates n items from a list in all possible ways:

```
   (1+\.i.5);(2+\.i.5);(3+\.i.5)
+-------+-----+---+
|1 2 3 4|2 3 4|3 4|
|0 2 3 4|0 3 4|0 4|
|0 1 3 4|0 1 4|0 1|
|0 1 2 4|0 1 2|   |
|0 1 2 3|     |   |
+-------+-----+---+
```

In the J line above `+` is monadic, which for real numbers is a ‘do nothing’ verb; that is, the left arguments 1, 2 and 3 are not added to elements of `i.5` but are arguments of the derived verb `+\.` that indicate how many items are to be dropped, progressively working from the left. The first case with 1 as left argument is thus the ‘all but one’ case.

An application of this is the calculation of minors of a determinant. Consider the rank-2 object `z`:

```
   z
3 6 7
9 3 2
```

```
9 7 7
   (1&(+\.))"2 z      NB. select 'all but' one rows
9 3 2
9 7 7

3 6 7
9 7 7

3 6 7
9 3 2
```

Now combine the structural verb *transpose* with the adverb *suffix* to switch rows and columns for all but one row at a time:

```
   transuff=.1&(|:\.)"2
   <"2 transuff z
+---+---+---+
|9 9|3 9|3 9|
|3 7|6 7|6 3|
|2 7|7 7|7 2|
+---+---+---+
```

and the result is a list of the transposed planes of `(1&(+\.))"2 z`.

Do this twice using the *power* adverb to generate three boxes within each of the above boxes; the row/column switch is fortuitously reversed and the minors of `z` obtained:

```
   <"2 transuff^:2 z
+---+---+---+
|3 2|9 2|9 3|
|7 7|9 7|9 7|
+---+---+---+
|6 7|3 7|3 6|
|7 7|9 7|9 7|
+---+---+---+
|6 7|3 7|3 6|
|3 2|9 2|9 3|
+---+---+---+
```

Define

```
   minors=.transuff^:2  NB. minors unboxed
   det=.-/ .*           NB. determinant
```

The determinants of the minors are given by

```
   each=.&>
   ]dz=.det each <"2 minors z
 7  45  36
_7 _42 _33
_9 _57 _45
```

This is verified by using the verb `det` dyadically:

```
    (det z);dz det |:z
+-+-------+
|3|3  0 0|
| |0 _3 0|
| |0  0 3|
+-+-------+
```

It is often convenient to use cofactors, that is the signed determinants of minors. This requires multiplication by a matching matrix whose diagonals are alternately +1 and –1. One way of obtaining this matrix is:

```
   signs=.monad : '-_1+2*2|+/~i.#y.'
```

so that

```
   cof=.signs * det each @<"2@minors
   cof z
 7 _45  36
 7 _42  33
_9  57 _45
```

To summarise this section:

```
   transuff=.1&(|:\.)"2             NB. transpose with suffix
   minors=.transuff^:2              NB. minors unboxed
   det=.-/ .*                       NB. determinant
   each=.&>
   signs=.monad :'-_1+2*2|+/~i.#y.' NB. alternate 1,_1
   cof=.signs * det each@<"2@minors NB. cofactors
```

Reducing all but one

Rank again is at the heart of the matter, especially as typical experimental data is structured so that dimensions correspond to variables. However, J inherently structures higher-rank data as lists, then lists of lists, lists of lists of lists and so on, which implies a nesting within dimensions which is not usually reflected in the real world variables to which the dimensions correspond. For definiteness use q to demonstrate:

```
   ]q=.i.2 3 4
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
```

The standard definition of rank is:

```
   rk=.#@$   NB. rank
```

+/ 'inserts' + between items of the list at the topmost level, thereby reducing rank by one, that is merging planes:

```
   +/q      NB. equivalent to +/"n q for n>2
12 14 16 18
20 22 24 26
28 30 32 34
```

Explicit rank conjunctions allows such reduction to take place at any level in the rank hierarchy:

```
   (+/"1 q);(+/"2 q)  NB. merge cols ; merge rows
+--------+-----------+
| 6 22 38|12 15 18 21|
|54 70 86|48 51 54 57|
+--------+-----------+
```

Progressive reduction to the level of a rank-1 object is obtained using a recursive verb:

```
   sum=.sum@(+/)` ] @.(=&1@rk)  NB. sum down to rank 1
   sum q
60 66 72 78
```

The above values are readily verifiable as the column sums of q. To find other such sums, say row sums, transpose the data so as to bring the corresponding dimension to the top level. This suggests a general verb which takes the list i.n and performs the a set of n possible shifts necessary to bring each item in turn into the leading position:

```
   EACH=.&.>
   shifts=.|.EACH< NB. all distinct shifts of a list
   shifts i.rk q
+-----+-----+-----+
|0 1 2|1 2 0|2 0 1|
+-----+-----+-----+
```

If the argument to `shifts` is a list generated by `i.`, the result is left arguments to *transpose* which provide all the restructured forms of `q` to which `sum` can be applied. This in turn is determined by the rank of the data matrix, so define

```
   targs=.shifts@(i.@rk)  NB. arguments for |:
   targs q
+-----+-----+-----+
|0 1 2|1 2 0|2 0 1|
+-----+-----+-----+
```

The full set of transpositions to supply all possible sums by dimension is then

```
   transposes=.targs |:EACH <     NB. reqd. transposes
   transposes q
+-----------+-----+--------+
| 0  1  2  3| 0 12| 0  4  8|
| 4  5  6  7| 1 13|12 16 20|
| 8  9 10 11| 2 14|        |
|           | 3 15| 1  5  9|
|12 13 14 15|     |13 17 21|
|16 17 18 19| 4 16|        |
|20 21 22 23| 5 17| 2  6 10|
|           | 6 18|14 18 22|
|           | 7 19|        |
|           |     | 3  7 11|
|           | 8 20|15 19 23|
|           | 9 21|        |
|          |10 22|        |
|          |11 23|        |
+----------+-----+--------+
```

These values are readily verifiable by summing the columns in the boxes above:

```
   sum EACH@transposes q
+----------+-----+---------+
|60 66 72 78|66 210|60 92 124|
+----------+-----+---------+
```

To give these sums in dimension order, that is so that the `$EACH` of the result matches the shape of `q`, write

```
   allsums=.1&|.@(sum EACH@transposes)
   allsums q
+------+--------+-----------+
|66 210|60 92 124|60 66 72 78|
+------+--------+-----------+
```

To summarise this section:

```
   rk=.#@$                             NB. rank
   sum=.sum@(+/)`] @.(=&1@rk)          NB. sum down to rank 1
   shifts=.|.EACH <                    NB. all shifts of i.n
   targs=.shifts@(i.@rk)               NB. arguments for |:
   transposes=.targs |:EACH <          NB. reqd. transpositions
   allsums=.1&|.@(sum EACH@transposes)
```

For comparison, here is a one-liner picked from the J Software forum some years ago which performs the same function by using the *power* adverb `^:` to apply `+/` the requisite number of times with transpositions required between steps as in the version above:

```
   mt=.(<@(+/^:(<:@rk)@:|:)"0 _~i.@rk)
   mt q
+------+--------+----------+
|66 210|60 92 124|60 66 72 78|
+------+--------+----------+
```

Subtotaling

It can be useful to be able to append such reductions to the original data as in:

```
  total=.,+/ NB. append totals for leading dimension
  sub=.3 :0
i=.0 [ r=.total y.
while.(i<<:rk y.)do.r=.total"i r [ i=.i+1 end.
)
   sub q
 0  1  2  3   6
 4  5  6  7  22
 8  9 10 11  38
12 15 18 21  66

12 13 14 15  54
16 17 18 19  70
20 21 22 23  86
48 51 54 57 210

12 14 16 18  60
20 22 24 26  92
28 30 32 34 124
60 66 72 78 276
```

Multi-statement lines using `[` as a separator as in `sub` allow something approaching the succinctness of tacit definition. However the individual statements are executed from the right since `[` itself is just another verb. It is easy to remember that it is `[` rather than `]` which is the separator, since, for example, it is 0 to the left which is assigned to `i` in the first line of `sub`. As far as the J interpreter is concerned it is really only one line which is executed; multiple lines are essentially just an orthographic device.

The Snap SALT command

by Daniel Baronet
(danb@dyalog.com)

Abstract In 'SALT: Simple APL Library Toolkit' and 'SALT II' the author introduced Dyalog's toolkit for loading and saving APL scripts. Snap extends SALT by saving unscripted functions, operators, variables and namespaces as SALT scripts.

Syntax for the SALT command line

```
Snap [path]  -class=   -convert   -fileprefix=   -loadfn[=]  -
makedir
             -noprompt  -pattern= -show[=]  -vars[=]   -version
```

In the following, *object* refers to any APL object, either code (function, or operator) or data (including all namespaces).

Snap by default saves all new objects in the current namespace to the path specified as argument and saves modified objects to their proper location. It returns the list of the names of the objects that have been saved, if any. If the path is not specified then the current working directory is used.

Snap, in its present state, cannot save everything, and variables in the workspace root (#), function references, idioms and instances cannot be saved.

Snap uses `Save` to store objects.

Procedure

Snap first finds the list of all objects and from those the ones that can be saved. It then determines which ones have been modified or are new. If any of them needs saving or if it cannot determine if they need to be saved (e.g. unscripted namespaces) then each one of these object is saved using `Save`.

To find out which object needs to be saved SALT marks objects (functions and namespaces) with special tags when it loads them or when they are first saved. Because # variables cannot be tagged they cannot be saved in canonical form on file. Variables in namespaces, on the other hand, can be saved as part of the namespace without any problem as long as they don't contain anything 'illegal' like forms, ⎕ORs, etc.

Unscripted namespaces can be saved by making them into a script, but tagging information cannot be retained and Snap will always overwrite (given permission) the existing file where they reside. Snap is non-destructive in that respect, and unless told otherwise (see below) it will keep the original unscripted form even though it writes a source for them.

Alternate behaviour

Snap accepts a series of modifiers to alter its operation.

Selecting the objects to save

It may be desirable to save only a subset of the workspace, for example only the functions or names starting with a specific pattern.

To select specific name classes (e.g. functions or D-ops) you use the `-class=` modifier. It takes a value of one or more classes. Acceptable classes are 3 (all functions), 4 (all ops) and 9 (all namespaces). Rational numbers are more specific, e.g. 4.1 for trad ops or 9.5 for interfaces.

To select objects following a specific pattern you use the `-pattern=` modifier. It takes a string where `*` means *any number of chars*. At the moment, the pattern matching is restricted to strings ending in `*` and stars are ignored, effectively meaning *only names starting with the string given before the star* as in `util*`[1].

Both can be combined and the following will save all namespaces starting with `GUI`:

```
⎕SE.SALT.Snap '/ws/utils -class=9.1 -pattern=GUI*'
```

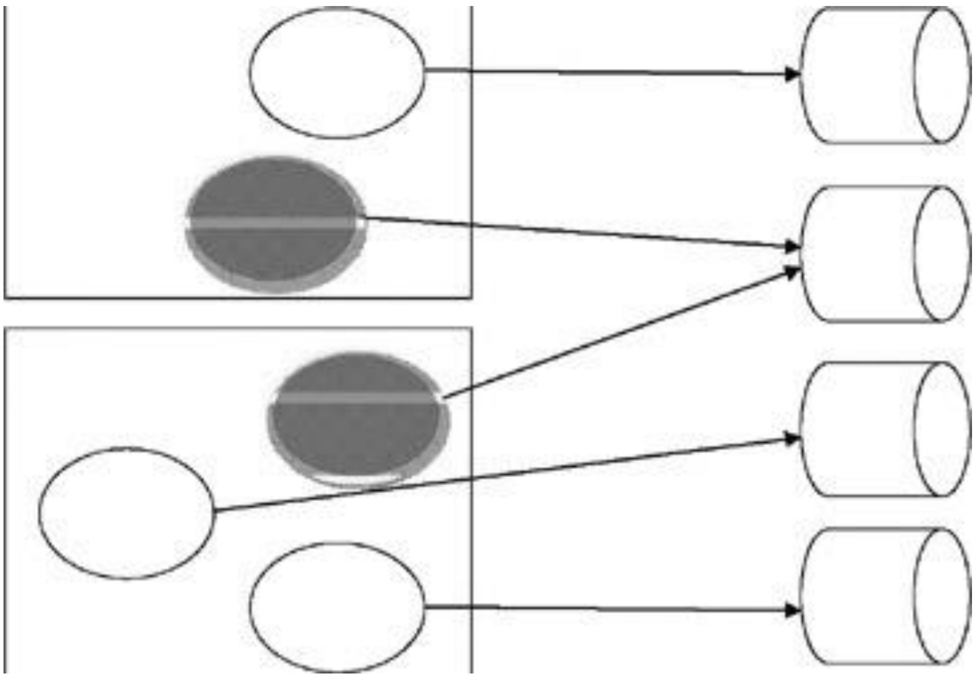
Both also accept the character `~` in the first position to exclude specified class or pattern, e.g.

```
⎕SE.SALT.Snap '/ws/utils -pattern=~GUI*'
```

will save all objects *not* starting with `GUI`.

These modifiers allow you to save various subsets of functions and namespaces to different locations. Two or more different workspaces can use the same subset, which can be updated independently of the workspaces.

How two workspaces can share code



Verifying what will happen

If you are not sure what will happen when you use Snap you can ask it to show you which objects would be saved only. For example,

```
⎕SE.SALT.Snap '/ws/utils -class=~9 -show'
```

will show you only the names that would be saved without actually saving them.

Changing the filenames

By default the filenames used are the same as each object's name followed by `.dyalog`. If you wish to prefix these names by a special string you can use the `-fileprefix` modifiers. For example,

```
⎕SE.SALT.Snap '/ws/utils -pattern=GUI* -fileprefix=Win'
```

will save all new objects starting with `GUI` to files starting with `Win`, thus function `GUImenu` will be saved in file `/ws/utils/WinGUImenu.dyalog`.

Skipping prompts

Unless your general prompting settings are set to `NO`, Snap will prompt you for replacement each time it finds a file that already exist. You will then have a choice of `Yes` to replace the file, `No` to skip that file or `Cancel` to skip the rest of the files. Snap will return only the names of the objects that have been effectively saved. If you wish to skip the prompting you can add the modifier `-noprompt`.

Using version numbers

SALT can keep version numbers for each file. If you wish to enable that feature for the objects Snap is about to save, use the modifier `-version`. You can also specify the version number to use in which case *all* objects will be saved with the same version number.

Creating intermediate directories

When Snap saves an object it assumes the directories in the path given as argument already exist and won't attempt to create them. If you know they don't exist (typically the first time you save objects) you must ask Snap to create the directories for you by using the `-makedir` modifier otherwise the command will fail.

Converting namespaces to scripted form

When Snap saves namespaces in unscripted form it has to convert them first into scripted form in order to perform the save. It then gets rid of the converted format and keeps the original one. If you subsequently Snap the workspace it will redo the work even is no change occurred to these namespaces because they cannot be tagged to let know Snap of the changes. If you wish to retain that form use the `-convert` modifier.

Recreating the workspace

You may be interested in recreating the workspace at a later moment in time. To do so you must remember which objects made up the workspace and bring them back one by one. Doing this by hand would be too tedious and prone to errors so Snap provides a way to do that for you.

If you use the `-loadfn` modifier Snap will create a function named `load_ws` in the same path given as argument. That function, when executed, will bring every object needed in the current workspace[2] and run the `⎕LX`. If you want the function to exist elsewhere use `-loadfn=/new/path` instead. If the name `load_ws` does not suit you then you can change it by adding an extension.

There are two possible extensions: `.dyalog` (or more simply `.`), which will create a function, or `.dyapp`, which will create a script. Both are suitable for use by `Boot` and have the same effect. For example, to create a function named `loadit` do

```
⎕SE.SALT.Snap '/ws/myapp -loadfn=/ws/myapp/loadit.dyalog'
```

Trick

If you have several workspaces to snap whose directory name is the same as the workspace's name you can use = instead of the workspace name when specifying the target folder. For example, let's say you want to store the workspaces in your personal library under /apl/wslib so that workspace W is stored under `/apl/wslib/W` you can use

```
)Xload W

SE.SALT.Snap '/apl/wslib/= -loadfn -makedir'
```

for each workspace without having to retype the Snap line for each workspace; simply grab the line from the session log to re-execute it after each `)XLOAD`.

Restrictions

In its present form Snap cannot store some objects such as instances of GUI controls, or variables in #. This restriction is unlikely to be lifted soon. Idioms and function refs are another kind of object impossible to store in this version.

Workarounds

Recreating the variables or instances upon loading the workspace (i.e. thru `⎕LX`) will always work but may not always be practical because sometimes the work involved in recreating them will be substantial. For variables another way to do this would be to store them in a separate (scripted) namespace. When the workspace is reloaded the variables are moved to the root space. There is a modifier to do just that.

This modifier, `vars`, will create a namespace with the definitions of all legal root variables and save it under the name `vars_ns`. When used in conjunction with the `loadfn` modifier the load function created will contain a statement to load the contents of the file and disperse its contents into root, effectively recreating all globals. If you prefer another name for the file you can use –vars=newname

Epilogue

For all its elaborate definition, Snap is deceptively simple. It merely saves all new objects where specified, and modified ones where they belong. All the modifiers simply allow you to fine-tune the process.

Notes

1. Without the star, the pattern would be an exact match for the name specified in which case the `Save` command might as well be used.
2. It will merge the objects with the existing ones in the workspace, replacing them if necessary.

Coding for a shared world

suggested by A Reader

Abstract Writing for the DotNet world means accommodating multiple metric systems. An anonymous reader challenges you to manipulate GPS coordinates in APL.

Introduction

DotNet languages compile to intermediate code which is run by the Common Language Runtime (CLR) [1]. This opens the field to many languages and raises the question: what advantage does one language have over another?

> Different programming languages allow you to develop using different syntax. Don’t underestimate the value of this choice. For mathematical or financial applications expressing your intentions by using APL syntax can save many days of development time when compared to expressing the same intention by using [other languages] [1]

APL’s mathematical primitives and array-handling make it a powerful tool for addressing problems involving GPS (Global Positioning Systems). These days you could choose to use an APL interpreter to produce a DotNet assembly as all or part of your commercial solution.

DotNet assemblies commonly accept data in all or most of the units in common use. Programmers expect such assemblies to handle unit conversions for them, rather than requiring them to convert their data to a single metric, or match assembly to metric. APL programmers will not want to do less.

C# syntax supports a technique that can be imitated profitably in APL, namely, method- (for APL, read *function*-) *signatures*.

> The *signature* of a C# method consists of the name of the method and the number, modifiers, and types of its formal parameters. [2]

Signatures and valence

The APL concept of *valence* is similar: a *niladic* function takes no arguments, a monadic function takes a right–hand argument, a *dyadic* function takes two arguments, left–hand

and right–hand arguments, and an *ambivalent* function that is essentially dyadic but with the left–hand argument being optional.

We can use dyadic or ambivalent functions to produce the equivalent of C# signatures: we use the left argument to specify how the right is to be understood. But functions that are to be exported as DotNet assemblies might have to be monadic, in which case we need, like a C# function, to respond to the structure of the right argument.

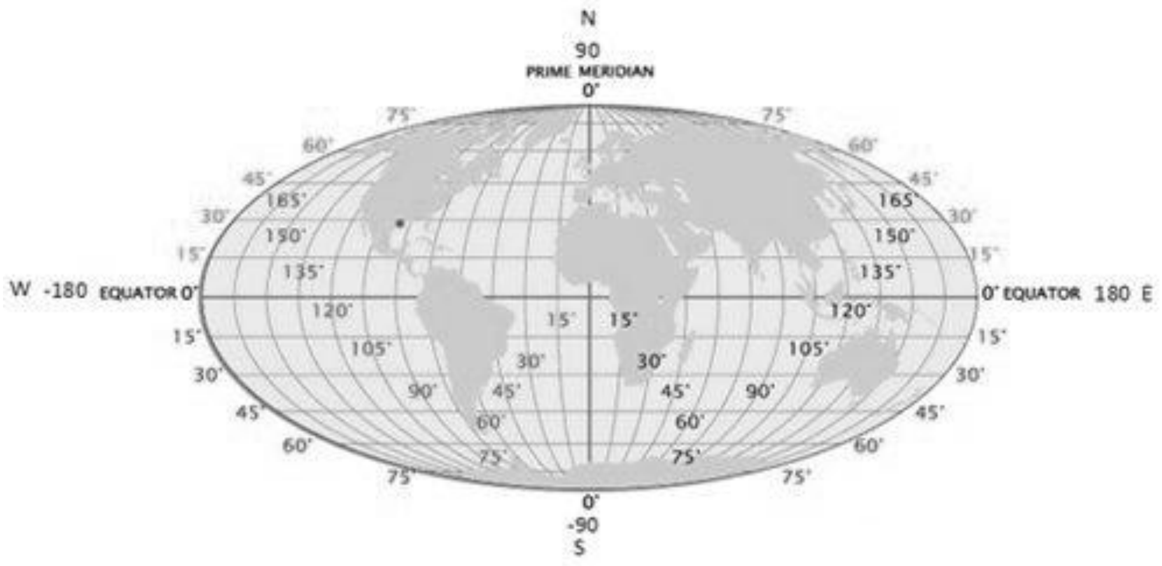
In C#, when a function has multiple signatures, each version is coded separately under the same name. Consider this example:

```
public static int First(int abc, int def)
{
  return (abc + def);
}
public static int First(int def)
{
  int abc = 90;
  return (abc + def);
}
```

The function `First` is coded twice; the first signature requires two arguments and the second requires only one. The function can be called either as `First(n)` or as `First(n,m)`, where `n` and `m` take any integer value.

In APL a function has but one definition, but, unlike C#, there is no constraint on the type or shape of the arguments. So APL can take a right-hand argument as a scalar, or a vector of two elements, or a vector of three elements. Effectively, this makes it possible to code with different signatures: a scalar represents an argument of one type, a vector of two elements another, and a vector of three elements yet another.

The problem with the world



© *worldatlas.com*

GPS is a rectangular coordinate system that allows any point on the planet's surface to be uniquely identified through two numbers: *latitude* and *longitude*. Latitudes and longitudes are at right angles to each other only at the point of intersection (0 the equator, 0 the prime meridian) on the surface of a sphere and they share the compass directions – points above the equator are North, those below are South and points to the right of the prime meridian are East and those to the left are West.

The similarity with the planar Cartesian coordinate system ends here since the distance between a pair of longitudes different by one degree is not uniform – the distance decreases progressively when moving from the equator towards the poles.

The data types

Latitudes and longitudes are data types, expressible in a variety of ways:

	Latitude	**Longitude**
Degrees, minutes, seconds	4° 14' 21" -4° 14' 21"	23° 16' 48" -23° 16' 48"
Degrees, minutes	4° 14.35' -4° 14.35'	23° 16.8' -23° 16.8'

	Latitude	**Longitude**
Degrees	4.2391666667° -4.2391666667°	23.28° -23.28°

Alternatively,

	Latitude	**Longitude**
Degrees, minutes, seconds	4° 14' 21" N 4° 14' 21" S	23° 16' 48" E 23° 16' 48" W
Degrees, minutes	4° 14.35' N 4° 14.35' S	23° 16.8' E 23° 16.8' W
Degrees	4.2391666667° N 4.2391666667° S	23.28° E 23.28° W

- Latitudes are in the range -90 to 90.
- Longitudes are in the range -180 to 180
- A minute has 60 seconds; minutes and seconds are in the range 0 to 59.
- Only the first number in any numeric representation i.e. DMS, DM or D, is negative.
- Negative latitude indicates a position below the equator. Alternatively, keep the numbers positive and add the compass point S for South; N indicates a position above the equator.
- Negative longitude indicates a position left of the prime meridian. Alternatively, keep the numbers positive and add the compass point W for West; E indicates a position right of the prime meridian.

Further notes

Latitudes and longitudes are expressed consistently across the world using DMS, DM or D; there are no localisation issues.

Rarely, 90 is added to a latitude in order to make it have the range 0 to 180; likewise, 180 is added to longitude to give it the range 0 to 360 in order to be able to work with positive numbers only and without the compass point. It is prudent to allow for this convention for incoming data. However, this approach is counter-intuitive and should not become part of a user interface as it will simply add confusion – users should be free to specify DMS, DM, or D.

The challenge

Write four functions:

1. `COMP` converts a latitude or longitude from D, or DM, or DMS to the corresponding D, or DM, or DMS with the compass indicator, and vice-versa.
2. `CAST` converts an argument from one representation to another; for example from D to DMS or DM, from DM to D or DMS, or from DMS to D or DM. The argument and result are free of the compass point, courtesy of the function defined in the first task.
3. `LATVAL` validates an argument expressed as either D or DM or DMS as latitude; this function returns 1 for valid and 0 otherwise. The argument is free of the compass point.
4. `LONVAL` validates an argument expressed as either D or DM or DMS as longitude; this function returns 1 for valid and 0 otherwise. The argument is free of the compass point.

Obviously, the functions should cope with scalar and array arguments! Write up your solutions (code plus notes, please) – and send them to the Editor. Our panel of distinguished judges will select the most interesting for publication.

References

16. Jeffrey Richter, *CLR via C#*, 2nd Edition, Microsoft Press, February 2006, ISBN-13 978-0735621633
17. Anders Hejlsberg & Scott Wiltamuth, *C#* Language *Reference*, Microsoft

3: Structural ingredients

by Neville Holmes (neville.holmes@utas.edu.au)

This article is the fourth in a series (numbered origin-zero *Ed.*) expounding the joys of functional calculation. Functional calculation does with operations applied to functions and numbers what numerical calculation does with functions applied to numbers. The functional notation used as the vehicle in this series is provided by a freely-available calculation tool called J.

This article reviews the numerical calculation capabilities of J which enable lists and tables to be used in calculations. To understand the functional calculation capabilities proper, structural capabilities – dealing with tables and lists – must first be understood. The capabilities described in this article will be illustrated and explained in detail in the next article.

Calculation

Calculation is generally reckoned to be the systematic manipulation of numeric values. Previous articles in this series have reviewed the arithmetic functions available in J, the interpreter used here as the vehicle for numerical calculation.

The numerical calculation described so far is merely an elaboration of what might be achieved by a conventional electronic calculator. There is an elaboration of ways numbers can be expressed, and an elaboration of functions that can be applied to any number, but that is all.

However rich the set of arithmetic functions available to the user, they cannot do much more than an ordinary pocket calculator can do, unless those arithmetic functions can be applied to lists of numbers, and to lists of lists. Functions that manipulate lists and tables, functions beyond the merely arithmetical, are essential to support the arithmetic functions, and these functions can be called *structural*.

The next article will illustrate the use of these structural functions. They provide an elaboration still within the realms of numerical calculation, namely the ability to structurally manipulate more than one number at a time. Of course, the arithmetic functions already described can also be applied to lists and tables of numbers.

Building structures

In the first place, constant lists of numbers can be keyed in simply by separating the elements of the list by blanks. So what ? Well, the scalar functions can be applied to compatible lists, that is, to lists with the same number of elements: `3 4 5*6 4 2` yields `18 16 10`, for instance. And if one of the arguments is a scalar (not a list, but a naked number) then it is coupled to each element of the other argument: `60%3 4 5` yields `20 15 12`, for instance.

But other functions are needed, functions that allow lists to be built easily, that allow lists of lists to be built, that allow lists of different kinds to be built. The following table gives the symbols and names for some such functions.

$	shape	reshape						
#	tally	copy	#.	unbits	undigits	#:	bits	digits
?		deal						
<	box		+.	cartesian		":	format	format
>	unbox		*.	polar				
;	raze	link	i.	integers		;:	words	

The *reshape* function is the most useful of these, and its explanation covers many important issues about structures.

- The left argument of `$` specifies what shape is to be given to the result, the right argument supplying the item or items to be used in the result, and these items are reused cyclically if necessary. Thus, `5$6` yields `6 6 6 6 6`, and `5$6 7` yields `6 7 6 7 6`.
- Lists of lists may be directly formed, for instance, `2 3$4` yields a two item list, each item of which is itself a list of three `4`s. This is very much like a matrix or table with two rows and three columns. If a table is reshaped then the items that go into the new structure are the rows of the table. Thus the expression `4$3 2$1` will yield a table of four rows and two columns.
- Monadic `$` yields the *shape* of its argument, always a list, so the shape of a shape yields a single item list which gives the number of dimensions of the

original argument, its *rank*. The rank of a table is two, of a list, one, and of a scalar, zero.

- A structure can have a shape of `0`, that is, it may hold no elements at all. Such a structure is called an *empty* list, and is of particular interest under calculation. Consider, for example, what the average of an empty list should be. Consider also that a table might be empty because it has no rows, or empty because it has no columns, and not necessarily both.
- A list of one item, where that item is an element (not a list), is not the same thing as a scalar. A list of one elementary item has a shape of `1`, while the shape of a scalar is the empty list.

The other functions in the table have more particular properties, and their explanation also raises some interesting issues.

- The *copy* function is like a reshaping, except the items are reshaped individually. Thus, `2#3 4` yields `3 3 4 4`, and `2j1 1j2#3 4` yields `3 3 0 4 0 0`.
- *Tally* gives a simple count of the items in its argument. Thus the tally of a scalar is the same as the tally of a list with one element, `1`, though their shapes are different. The tally of an empty list will always be `0`, but the tally of an empty table will only be `0` if it has no rows.
- *Integers*: monadic `i.` is like a reshape but the argument gives the shape of the result, and the elements being reshaped are the nonnegative integers taken in sequence. *Deal* randomly selects from the non-negative integers smaller than its right argument as many as specified by its left argument. Of course the results of both roll (monadic `?`) and deal are only pseudorandom, and the pseudorandom seed can be both inspected and set.
- *Cartesian* and *polar* split numbers in their arguments into two components, as their names suggest. *Bits* and *digits* split their argument numbers into digits, binary in the monadic case, to the base given in the left argument in the dyadic case. *Unbits* and *undigits* reverse the effect of the previous two functions, and so don't really build structures.
- Here it should be stressed that a structure must be composed entirely of the same kind of element, and there are three kinds of elements – numbers, characters, and boxes. (In fact J also provides for structures of functions, but these are passed over here.) Boxes are explained just below. A character string is a list of characters. For example, `$'this'` yields `4`. The two *format* functions are used to convert numbers to character strings, while *do* can convert character strings to numbers. Incidentally,

there is a special name `a.` which stands for the *alphabet*, the list of all possible characters.

- It is often useful to combine different items into a structure, for example to mix numbers and their names, or to mix structures of different shapes. This can be done by putting them into boxes using *box*, which produces a scalar box, or using *link*, which produces at least two boxes in a list of boxes, or *words*, which when applied to a character string yields a boxed list of the words and punctuation in the string. *Words* is particularly useful to apply to a J expression because it boxes its syntactic components, which can help to understand an expression. Arithmetic cannot be done on characters, nor on boxes, but the *unbox* (or *open*) function can remove the box that holds a structure, and *raze* can remove the boxes from a structure of boxes.

Rebuilding structures

In contrast to functions that build structures from typically lesser structures, like scalars, there are functions that rearrange structures, or extract lesser structures from greater. Such functions are given in the following table.

			|.	reverse	rotate	|:	transpose	transpose
,	ravel	append	,.	knit	stitch	,:	itemise	laminate
						/:		sort up
						\:		sort down
{		from	{.	head	take	{:	tail	
			}.	behead	drop	}:	curtail	
			~.	nub				

Functions that merely rearrange their components are those shown in the first four rows of the table.

- *Reverse* reverses the sequence of the items of its argument, while *rotate* cyclically shifts the items of its right argument a number of places depending on its left argument. The shift is to the left for a positive number, to the right for a negative. If the right argument is a table then the left argument can be a list of two numbers, the first saying how far the items are to be rotated, the second how far the elements within each item.
- The *transpose* functions rearrange the axes of their right argument. This can be seen as a rearrangement of the elements of the shape of the right argument, a kind of second order rearrangement. Thus, monadic *transpose* will make the rows of a table become its columns, and vice versa. Dyadic *transpose* can be used simply to rearrange axes of structures of rank greater than two, but it can also be used to run axes together and to pick out diagonals of various kinds.
- *Sort up* and *sort down* are anomalous compared to the functions just described, because they reorder the items of their left argument rather than of their right! The reordering sequence is defined by the lexical sequence of the items of the right argument, using `a.` to define the collating sequence of characters in the case of a character structure for right argument. In particular, `x/:x` will yield a list of the items of `x` in ascending sequence, and `x\:x` in descending.
- *Ravel* runs all the axes of its argument together, yielding a simple list of the elements of the argument. *Append* runs together the items of its arguments, and will expand a scalar argument if necessary. *Knit* ravels the items of its argument, while *stitch* appends corresponding items of its arguments. Understanding the difference as described between these two pairs of functions depends on appreciating that the items of a structure are the components of its highest level list. Thus, `1 2,3 4` yields a list of four items, while `1 2,.3 4` yields a two by two table.
- *Itemise* yields its argument as a single item list, while *laminate* appends its arguments after itemising each of them. Note that `2 3$4 5` does not yield the same result as `2$,:3$4 5`, though the shape of the results is the same.

The functions given in the lower section of the preceding table typically pick out items from their right argument.

- *Head* yields the first item of its argument, *tail* the last; *behead* all but the first, *curtail* all but the last.

- *Take* with a positive left argument yields that many leading items of its right argument; with a negative, that many trailing items. The *take* function can extend an existing structure, normally with `0`s in the case of a numeric structure; with blanks in the case of a character structure; or with empty boxes in the case of a box structure. Thus, `3{.6` yields `6 0 0`, and `_3{.4 5` yields `0 4 5`. One little trap is that although the result of *head* looks the same as the result of *one take*, their shapes are different.
- *Drop* is the opposite to *take* in the obvious way. *Nub* takes all the distinct items of its argument, dropping duplicates out.
- *From* selects from its right argument the items indexed by its left argument. Here it is important to realise that the indexes start from zero, so that `1{x` will pick out the second item of `x`. Interestingly, `_1{x` will pick out the last element of `x`.

Structural data

Finally, there are a few interesting functions which, like shape and tally, extract data about their arguments.

		%.	invert	project			
=	classify				-:		match
		e.	raze in	member	~:	sieve	
		i.		index	/:	grade up	
					\:	grade down	

These are something of a mixed bunch, *invert* and *project* being simply a generalisation of matrix inversion and matrix division.

The functions given in the vertically centre section of the table all produce boolean values, that is, values composed only of `0`s and `1`s, like the results of the comparison functions.

- *Match* is a kind of hypercompare, comparing its arguments in their entirety, yielding a `1` if they match in every respect, and a `0` otherwise.
- *Sieve* marks the occurrence of the first instance of each distinct item of its argument. Thus, `(~:x)#x` yields the nub of `x`.
- *Member* works in two ways. If its left argument has the same shape as the items of its right argument, then it looks for a match for its left argument with any item of its right argument. Otherwise it reports whether the elements of its left argument are present anywhere in the right argument.
- *Classify* and *raze in* both produce a boolean table. For classify, the rows of the table correspond to the items of the nub of the argument, and within each row the occurrences of the item of the nub are marked for each item of the argument. For structures of numbers or characters, raze in will mark the occurrence of each of the elements of its argument in its ravel. Thus `e.i.4` will produce the unit matrix of size four, while `$e.i.2 3` will yield `2 3 6`. For structures of boxes, membership of the elements within each box are marked for each element of the raze.

Given that so many results depend on a comparison of numbers, it is important to realise that these comparisons are done with a certain amount of tolerance to allow for the imperfection of traditional computer arithmetic, remembering that, for example, present day computers cannot store the value `1r3` exactly. However, the J interpreter does provide for the comparison tolerance to be respecified, even to zero.

The functions given in the bottom section of the table all yield an index to items of the left or only argument.

- The *index* function is very frequently used, and yields, for the items of its right argument, their location in the left argument. If an item is not present in the left argument then it is given the first invalid location. Thus, `(i.5)i.2 3 3.5 4 5 6` yields `2 3 5 4 5 5`.
- *Grade up* and *grade down* yield the permutation index of their argument that would select its items in the required sequence from its argument. Thus, `(/:x){x` will yield the list of `x`'s items in ascending sequence.

Summary

This article is like a list of ingredients that can be used for structural calculation using the notation provided by J. These ingredients allow lists of numbers and tables (lists of list) of numbers to be manipulated.

The arithmetic functions described and illustrated in previous articles can also be applied to lists and tables. Together, the arithmetic functions and structural functions provide a very powerful capability for relatively complex numerical calculations. This capability is needed as the basis for functional calculation proper, as will be described in later articles of this series.

However, the next article is devoted to illustrating simple use of the structural functions described above.

In Session

On average

Roger Hui

How do you compute the average in APL? Many authors and speakers say:

```
      avg←{(+/⍵)÷⍴⍵}
```

The function is often used to demonstrate the beauty and power of the language. It is presented in the *About the APLs* page of *Vector*. It is the first program in the *Dyalog APL Tutorial* (page 10). At the recent Dyalog User Conference in Princeton it was presented in at least three sessions, and in one was described as “the very best APL expression”.

Let’s look into it:

```
      avg 1 2 3 4
2.5
```

So far so good. What about

```
      avg¨ (1 2) (3 4 5) (6 5 4 3.14159)
 1.5  4  4.5353975
```

The extra blanks hint at the trouble. Application of the monad `↑` to the result brings the problem into relief:

```
      ↑ avg¨ (1 2) (3 4 5) (6 5 4 3.14159)
1.5
4
4.5353975
```

That is, `avg` should return a scalar rather than a vector result.

Now consider:

```
      avg 1 2 3 4
2.5
      avg 1 2 3
2
      avg 1 2
1.5
      avg 1
```

What just happened with the last expression? A few moments of reflection reveal that `avg` mishandles scalar arguments.

What about matrices and higher-ranked arrays?

```
      ⎕←data←?3 4⍴10
1 7 4 5
2 0 6 6
9 3 5 8

      avg data
RANK ERROR
avg[0] avg←{(+/⍵)÷⍴⍵}
          ^
```

In summary, the problems with `avg←{(+/⍵)÷⍴⍵}` are:

- It gives a non-scalar result for vector arguments.
- It fails on non-vector arguments; in particular, it gives a puzzling (and wrong) result for scalar arguments.

Fortunately, a better definition readily obtains. It is somewhat longer than the commonly promulgated version, but fixes its defects:

```
      avg1←{(+⌿⍵)÷⍬⍴(⍴⍵),1}

      avg1 1 2 3 4
2.5

      avg1¨ (1 2) (3 4 5) (6 5 4 3.14159)
1.5 4 4.5353975

      ↑ avg1¨ (1 2) (3 4 5) (6 5 4 3.14159)
1.5 4 4.5353975

      avg1 1
1

      avg1 data
4 3.333333333 5 6.333333333
```

There is one more point:

```
      avg ⍳0
1
      avg1 ⍳0
1
```

I would argue that `0` is a better answer for the average of an empty vector. For example, `(+/⍵)÷⍴⍵ ←→ +/⍵÷⍴⍵` and `(+⌿⍵)÷⍬⍴(⍴⍵),1 ←→ +⌿⍵÷⍬⍴(⍴⍵),1` for vector `⍵`, except when `⍵` is an empty vector. If instead `0` is that average, then the identity holds for all vectors. Possibly `1` is a reasonable answer, but if so it should be a considered answer and not an unintended consequence of that `0÷0` is `1`.

Finally, the common definition of average in J *is* arguably “the best expression in J”:

```
   avgj=: +/ % #


   avgj 1 2 3 4
2.5


   avgj&> 1 2 ; 3 4 5 ; 6 5 4 3.14159
1.5 4 4.5354


   avgj data
4 3.33333 5 6.33333


   avgj 1
1
   avgj i.0
0
```

Together with the rank operator ("), this definition of the average makes it easy to find the average of various parts of an array.

```
   avgj"1 data
4.25 3.5 6.25


   ] x=: ? 2 3 4 $ 20
10  7 18  9
 2 12 15  5
15 13 18 10

 0  5  9  6
14  0  6  4
 9  8 15 17
```

```
   avgj x
 5    6 13.5  7.5
 8    6 10.5  4.5
12 10.5 16.5 13.5

   avgj"3 x
 5    6 13.5  7.5
 8    6 10.5  4.5
12 10.5 16.5 13.5

   avgj"2 x
      9 10.6667 17 8
7.66667 4.33333 10 9

   avgj"1 x
11 8.5    14
 5   6 12.25

   avgj"0 x
10  7 18  9
 2 12 15  5
15 13 18 10

 0  5  9  6
14  0  6  4
 9  8 15 17
```

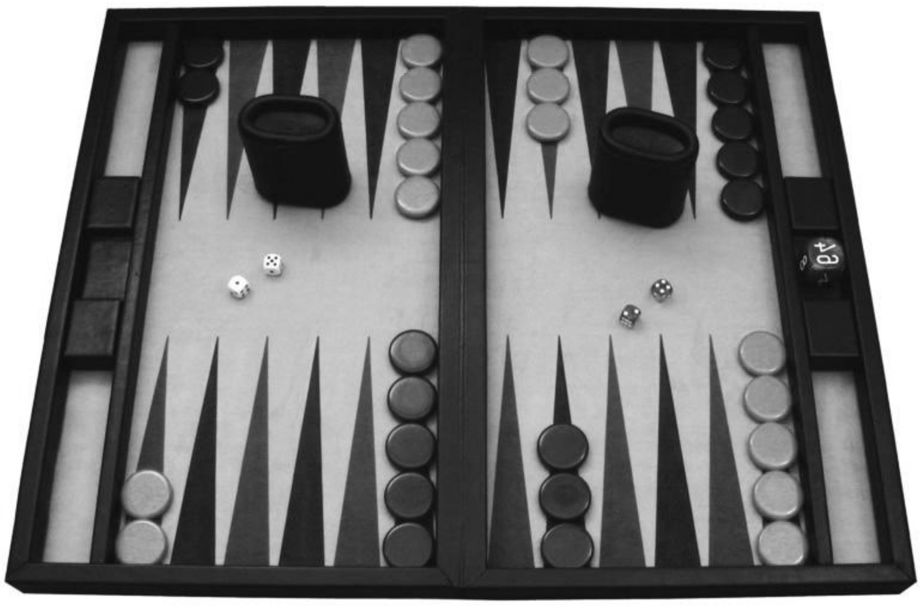
P R O F I T

Backgammon tools in J

1: Bearoff expected rolls

by Howard A. Peelle (hapeelle@educ.umass.edu)

J programs are presented as analytical tools for expert backgammon [1]. Part 1 here develops a program to compute the number of rolls expected to bear off all pieces in a player's inner board (with no contact).



The inner board is represented as a list of six integers. For example, `0 0 0 2 3 4` represents 0 pieces on the 1, 2, and 3 points, 2 on the 4-point, 3 on the 5, and 4 on the 6.

Assign a few utility names for readability:

```
ELSE =: `

WHEN =: @.
```

`Rolls`is defined for an inner board input, as in: `Rolls 0 0 0 2 3 4`. It is the master program and calls `Bearoff` or else returns a default result of `0` when all pieces are off the board:

```
Rolls =: Bearoff ELSE 0: WHEN AllOff
```

Subprograms are:

```
AllOff =: All@(Board = 0:)
  Board =: ]
  All =: *./
```

`Bearoff` is 1 plus the average of expected values for all 36 rolls:

```
Bearoff =: 1: + Average@AllRolls
  Average =: +/ % #
```

`AllRolls` computes the best expected value among all possible moves for each roll of dice – six doubles and two copies of the minimum of 15 pairs of singles rolls, as follows:

```
AllRolls =: Doubles , 2: # Min@Singles
doubles =: > 1 1 ; 2 2 ; 3 3 ; 4 4 ; 5 5 ; 6 6
singles =: > 1 2 ; 1 3 ; 1 4 ; 1 5 ; 1 6 ; 2 3;2 4;2 5;2 6;3 4;
                                              3 5 ; 3 6 ; 4 5 ; 4 6 ; 5 6
singles =: singles ,: |."1 singles
  Doubles =: doubles Best@AllMoves"1 Board
  Singles =: singles Best@AllMoves"1 Board
```

`Best` calls `Rolls` co-recursively to get expected numbers of rolls for each board and returns the minimum – effectively finding the best move:

```
Best =: Min@:(Rolls"1)
  Min =: <./
```

`AllMoves` generates alternative moves with the first die, then generates moves for each with the last die, or else moves four times repeatedly when the dice are doubles:

```
AllMoves =: (Last Moves First Moves Board) ELSE (First Moves^:4
Board)
```

```
                                              WHEN (First=Last)
  First =: {.@Dice
  Last =: {:@Dice
  Dice =: [
```

`Moves` produces a table of legal moves for each board given or else `0 0 0 0 0 0` as a default when all pieces are already off. `LegalMoves` calls `Move` and `Possibles` which calls `FromTo` to identify indices of each possible move from and to points:

```
  EACH =: &.>
Moves =: ;@(LegalMoves EACH <"1)
  LegalMoves =: (Possibles Move"1 Board) ELSE Board WHEN AllOff
    Possibles =: FromTo Where FromTo OK"1 Board
      FromTo =: On ,. On-First
      Where =: #~
        On =: Points Where Board > 0:
      OK =: Off +. Inside +. Highest
        Inside =: Last > 0:
        Off =: Last = 0:
        Highest =: First = Max@On
          Max =: <./
```

`Move` adds 1 to the point to move to and subtracts 1 from the point to move from:

```
Move =: Board + To - From
  From =: Points e. First
  To =: Points e. Last
  Points =: 1: + i.@6:
```

Although these programs generalize easily to the full 24-point backgammon board, they are intended only as prototypes – not for practical use. (See Appendix for an efficient program.)

Rolls may be used to compute the number of expected rolls for any number of pieces on any points in the inner board. For example:

```
   Rolls 0 0 0 1 1 1
2.48642
```

So, it takes about two and a half rolls to bear off these pieces on the 4, 5, and 6 points.

Beware: Since multiple recursion is extremely inefficient here, input boards with more than a few pieces will be impractically time-consuming. Accordingly, Rolls should be re-defined to look up expected values from a database produced by the script in the Appendix below.

Problem

As a simple illustrative problem, compare the well-known end-game cases of bearing off one piece on the 1 point and one on the 6 point vs. one piece on the 2 and 5 points. Since

```
   Rolls 1 0 0 0 0 1
1.58642
   Rolls 0 1 0 0 1 0
1.47531
```

the latter is better.

Now consider using Rolls to decide the best move for a roll of 6 1 in the position 0 2 2 2 0 4. (Problem 487 in [2].)

Answer: Play 6-Off, 6-5 (not 6-Off, 2-1) since

```
   Rolls 0 2 2 2 1 2
5.30267
   Rolls 1 1 2 2 0 3
5.40427
```

Acknowledgements

Thanks to the *Vector* reviewer for testing these programs and for helpful suggestions.

Special thanks to friend and colleague UMass Computer Science Professor Paul Utgoff for years of enjoyment and sparkling discussions of backgammon and programming.

References

18. For an introduction to backgammon, see http://en.wikipedia.org/wiki/Backgammon
19. Bill Robertie, *501 Essential Backgammon Problems*, Cardoza Publishing, New York, 2000
20. For on-line backgammon gaming, see www.GammonVillage.com

Appendix

Script for backgammon bearoff (with no contact) to build list of all 54264 `ncodes` and corresponding list of `ns` using base-16 coding for all inner boards (6 points):

```
Build =: verb define                               NB. Build (6)

boards =. (6#16) #: i.16^6

boards =. boards Where (+/"1 boards) <: 15         NB. select real
boards

ncodes =: 16 #."1 boards                           NB. base-16 codes

ns =: (#ncodes) $ 0                                NB. Initialize

N"1 boards                                         NB. Update all ns

)

N =: verb define                                   NB. N (6 integers)

code =. 16 #. y

index =. ncodes i. code

n =. index { ns

if. n=0 do. n =. Rolls y

             ns =: n index} ns                     NB. Update n in ns

end.          n

)
```

```
  ELSE =: `
  WHEN =: @.

Rolls =: Bearoff ELSE 0: WHEN AllOff
  AllOff =: All@(Board = 0:)
    Board =: ]
    All =: */
  Bearoff =: 1: + Average@AllRolls
    Average =: +/ % #
    AllRolls =: Doubles , 2: # Min@Singles
doubles =: > 1 1 ; 2 2 ; 3 3 ; 4 4 ; 5 5 ; 6 6
singles =: > 1 2;1 3;1 4;1 5;1 6;2 3;2 4;2 5;2 6;3 4;3 5;3 6;4 5;
                                                             4 6;5 6
singles =: singles ,: |."1 singles
  Doubles =: doubles"_ BestN@AllMoves"1 Board
  Singles =: singles"_ BestN@AllMoves"1 Board

BestN =: Min@:(N"1)                       NB. calls N co-recursively
  Min =: <./
AllMoves =: (Last Moves First Moves Board) ELSE
                             (First Moves^:4 Board) WHEN (First=Last)
  First =: {.@Dice
  Last =: {:@Dice
  Dice =: [
  EACH =: &.>
Moves =: Unique@;@(LegalMoves EACH <"1)
  Unique =: ~. ELSE ,: WHEN (#@$ < 2:)
  LegalMoves =: (Possibles Move"1 Board) ELSE Board WHEN AllOff
    Possibles =: FromTo Where FromTo OK"1 Board
      FromTo =: On ,. On-First
      Where =: #~

        On =: Points Where Board > 0:

      OK =: Off +. Inside +. Highest

        Inside =: Last > 0:

        Off =: Last = 0:

        Highest =: First = Max@On

          Max =: >./

    Move =: Board + To1 - From1

      From1 =: Points e. First

      To1 =: Points e. Last

      Points =: 1: + i.@6:

Build 6                        NB. 43 minutes on Octiplex PC running
J6.01

NB. Now #ns is 54264

NB. Next run bgn.ijs
```

Script `bgn.ijs` re-defines program `N` to compute expected number of rolls efficiently using `ncodes` and `ns` .

```
load 'jfiles'

jcreate 'bgns'

ncodes jappend 'bgns'

ns jappend 'bgns'

N =: verb define                             NB. N (inner board)

code =. 16 #. 6 {. y

index =. ncodes i. code

n =. index { ns

)

NB. Now use N for fast look up

NB. Example: N 0 0 0 0 0 1 is 1.25

NB. Example: N 0 0 0 2 3 4 is 6.56681
```

A genetic algorithm primer in APL

by Romilly Cocking
(romilly@cocking.co.uk)

Abstract This primer shows how you can implement and experiment with genetic algorithms simply and efficiently in APL.

Introduction

A genetic algorithm (GA) is a particular process that searches for a good solution to an optimisation problem. GAs use a technique that is based on Darwinian Natural Selection. They work well on problems that are too complex for an exact (analytic) solution, and are used in engineering and finance.

This primer shows how you can implement and experiment with GAs simply and efficiently in APL.

First, a disclaimer. GAs are very useful, but they are just one member of a large family of optimisation techniques. For example, optimisation problems can sometimes be solved by 'hill-climbing'. This process repeatedly improves approximate solutions by varying them in the direction which 'goes up-hill'.

Hill-climbing is simple to implement but it can easily become trapped by local optima. If you try to find high ground by always turning up-hill, you may get stuck on top of a low hill and miss a mountain.

If the function that you want to optimise is complex and has many hills, hill-climbing will not work well. In this case your best option may be to use a GA.

Genetic algorithms were introduced in a book by John Holland[1]. GAs are widely used and well-documented in the literature. Since GAs are inspired by biological systems they are described in biological terms.

GAs are based on the idea of reproducing populations. Throughout this paper, we will ignore sex, on the grounds that it only serves to make life more complicated. Instead we will assume that reproduction is asexual, as it is in bacteria.

Here are some simplified biological definitions (extended and adapted from Wikipedia):

Chromosomes are organized structures of DNA found in cells. A chromosome is a singular piece of DNA.

A *gene* is a locatable region of a chromosome that corresponds to a unit of inheritance. A chromosome contains many genes. One gene might determine your hair colour, while another might determine the shape of your ear lobe.

An *allele* is one member of two or more different forms of a gene. Thus the hair colour gene might have alleles for fair hair, brown hair, red hair and so forth.

A *locus* (plural *loci*) is a fixed position on a chromosome which defines the position of a gene.

We shall use the (non-standard) term *locus* set to mean the set of alleles that can exist at a particular locus.

Representation

We can represent the full range of genetic possibilities for an organism by listing the locus sets. If an organism is characterised by hair colour, eye colour and height we might set

```
      localSets←('short' 'tall')('fair' 'brown' 'red')
                                …('blue' 'green' 'brown' 'blue-green')
```

and evaluate

```
      ⍉↑localSets
short  fair   blue
tall   brown  green
       red    brown
              blue-green
```

While this notation is easy to read, it is fiddly to implement. Most GAs do not use names to represent alleles. In this primer we will use integers to represent alleles; we will represent an allele by its origin - index within its locus set. (Note to purists: sets have no order, but we will represent locus sets by vectors, which do.)

In this notation, we can specify the chromosome of a tall individual with fair hair and brown eyes by the vector `2 1 3`. We can represent a population of individuals by a matrix. Each row then represents the chromosome of an individual in the population.

Since we are using integers to represent alleles, we can easily generate a random population. We just need to know how many individuals it should contain and the size of each locus set.

```
      5 populate sizes ← 2 3 4
1 2 4
2 3 4
1 3 4
2 2 4
1 3 1
```

In Dyalog APL `populate` is a trivial dfn: `populate←{?ωρ⍨α,ρω}`

Scoring

Recall that a genetic algorithm is an optimisation technique. To implement a GA we need to know how to calculate the value we are trying to optimise. In other words we need a fitness function which we can apply to an individual or a population to find out how good a solution each individual represents.

In our case, we might envisage a 'Mr/Ms World' competition where each (sexless) participant gets a score based on their height, their hair colour and the colour of their eyes.

We'll assume that the judges have given us a scoring function which represents the value to be optimised. I've chosen a trivial evaluation function; it is just the sum of the allele indices. The winner is thus a tall redhead with blue-green eyes. Interesting real world problems have so many solutions that exhaustive search is impractical. Let's pretend that is true for our example.

Of course, the definition of the evaluation function is trivial. It is

```
      fitness←+/
```

Let’s do a little exploratory programming. First we’ll generate a population

```
      pop ← 6 populate sizes
      pop
1 1 3
2 1 2
1 3 2
2 2 1
1 2 1
1 2 1
```

How fit are the individuals in the population?

```
      fitness pop
5 5 6 5 4 4
```

Breeding

Now we breed the next generation by combining three biologically inspired processes – selection, recombination and mutation.

Selection

Selection takes the result of our evaluation function and uses it to favour the fitter (more valuable) members of the population. Fitter members get a higher chance of being selected for breeding.

Let’s see what happens when we apply selection. First we’ll generate a larger population, so that we have a good mix of individual types.

```
      pop ← 10 populate sizes
      pop
1 1 4
1 1 1
```

```
2 3 1
2 3 1
2 2 1
1 2 1
1 1 3
2 2 4
1 2 2
1 1 3
      fitness pop
6 3 6 6 5 4 5 8 5 5
```

The highest possible fitness is actually 9 (corresponding to the gene 2 3 4); our initial population includes a good individual with a score of 8, but no one who is perfect.

Let's apply selection:

```
      ng ← select pop
      ng
2 3 1
1 1 4
1 1 3
1 1 3
2 2 4
2 3 1
1 2 1
1 2 2
2 3 1
2 3 1
```

I've defined `select` so that it uses the terms of the domain. Here it is with the utilities which it uses:

```
      select←{(size ⍵) cull sort ⍵ copied(size ⍵)×normalise fitness
⍵}
      size←{1↑⍴⍵}
      cull←{(⍺,1↓⍴⍵)↑⍵}
      sort←{⍵[⍒fitness ⍵;]}
      copied←{⍺⌿⍨⌈⍵}
      normalise←{⍵÷+/⍵}
```

What effect did selection have on the fitness of the next generation? Let's use an `average` function to find out:

```
      average←{(+/⍵)÷⍴⍵}
      average fitness pop
5.3
      average fitness select pop
6.2
```

As we might hope, selection has increased the fitness of the population.

Recombination

Selection is pretty effective, but not enough on its own. In our example it will increase the number of good individuals, but it won't invent ones that are any better than the best we've currently got. We can improve things if we follow selection by recombination.

In recombination, we mate pairs of individuals and mix up their chromosomes. More specifically, we take a pair of chromosomes and cross them as shown below:

before:

```
1111
2222
```

after crossing over at the third locus:

```
1112
2221
```

If we consider longer chromosomes, we might have more than one-crossover.

before:

```
11111111111111111111111111111111111
22222222222222222222222222222222222
```

after a double cross-over:

```
11111112222222222222222111111111111111
22222221111111111111112222222222222222
```

To recombine, we need to specify a recombination probability as well as a population.

```
      0.3 recombine ng
2 3 1
1 2 1
1 1 1
2 3 1
2 1 4
2 2 1
1 1 4
1 1 3
2 3 1
1 3 1
```

If you compare this with `ng`, you will see that it has affected most of the individuals but not all. That's because we specified a recombination probability of 0.3, which is the probability of recombination at any locus; so the probability of at least one recombination is just over 0.75.

Here’s the definition of `recombine` and its sub-functions:

```
      recombine←{(⍴⍵)⍴pairs⊖⍨parity ⍺ of 1↓⍴pairs←mate ⍵}
      mate←{(⌈1 2 1÷⍨2,⍴⍵)⍴⍵}
      of←{1000≥?⍵⍴(⌈1000÷⍺)}
      parity←≠\
```

Mutation

Recombination and selection are pretty powerful, but there is one trick left up our sleeve: mutation. We may be unlucky; it might happen that a particular allele does not occur in any of the individuals in our starting population. In the GA implementation that we’ve chosen, neither selection nor recombination will ever generate a new allele. If the allele that’s missing from our starting population is required for the optimal solution, we will never get there.

We can be rescued by mutation. Mutation replaces alleles at random; if we evolve our population for long enough, each possible allele will be injected. It may take a while but we’re no longer trapped in some non-optimal subset of the search space.

Here’s the definition of `mutate`, which needs to know the probability of mutation, the size of each locus set, and the population to mutate.

```
      mutate←{
        (sizes pop)←⍵
        i←i/⍳⍴i←⍺ of×/⍴pop
        v←,pop
        v[i]←?(⍴i)⍴sizes[1+(⍴sizes)|i-1]
        (⍴pop)⍴v
      }
```

We can combine all these into a single breeding function, and define a `generate` function which applies it repeatedly.

```
      breed←{0.3 recombine 0.001 mutate ⍺(select ⍵)}
      generate←{((locusSizes∘breed)⍣ω)α}
```

Let’s evolve for two generations:

```
      pop generate 2
2 2 4
2 2 4
2 3 4
2 3 4
2 2 4
2 3 4
1 3 4
1 2 4
2 3 1
2 3 1
```

We’ve now got several copies of the optimal solution. Of course the algorithm doesn’t know it’s optimal, just that it’s the best we’ve seen so far.

Let’s run two, three and four generations checking the average fitness.

```
      average fitness pop generate 2
7.2
      average fitness pop generate 3
8.6
      average fitness pop generate 4
9
```

By the fourth generation the whole population is made up of the optimal solution.

Genetic algorithms work well. That’s not surprising. The blind watchmaker[2] has been using them with excellent results for quite a while.

References

1. Holland, John H. *Adaptation in Natural and Artificial Systems* 1975
2. Dawkins, Richard *The Blind Watchmaker* 1986

Functional calculation

DNA identification technology and APL

by Charles Brenner
(chb@dna-view.com)

Abstract Biology and biological data, unlike engineering data, are inherently irregular, incomplete, and inaccurate. The author exploits the nimbleness of APL to identify criminals, fathers, World Trade Center and tsunami victims, and determine race using DNA in a world of fast-changing DNA identification technology.

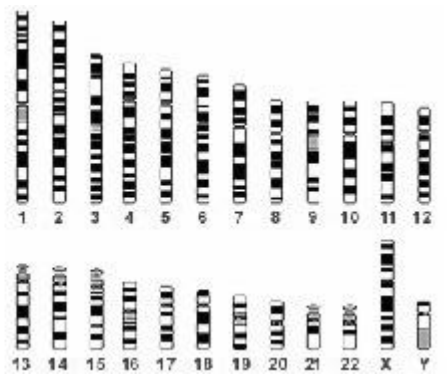
Introduction

By the late 1980s DNA was well toward replacing serology as a far superior method of genetic testing, new technological frontiers were consequently opening up, and I was glad for the chance to put behind me 'programming for hire' in APL development, business applications, or whatever, and to turn exclusively to finding ways to merge my interests in computers, mathematics, and science.

The essence of the genome is a sequence of 3×10^9 base-pairs, the bases (nucleotides) being A, C, G, T and the pairs being A-T (or T-A) and C-G (or G-C).

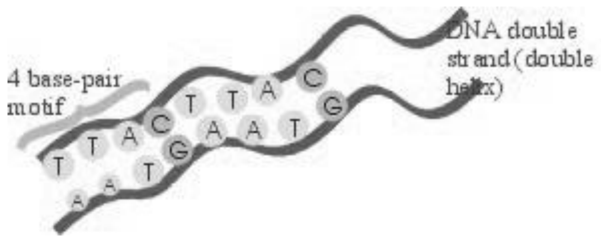
Note that there is pairing at two levels: the chromosomes are paired in that there are two of each number, which is physically significant during cell division and reproduction and genetically significant because most genes occur in two copies. Second, the DNA strand in each chromosome is a double strand (double helix), with the sequence of A, C, G and Ts on one strand mirrored by a complementary sequence of T, G, C and As on the other by the pairing rule. During cell division the strands separate and each single strand then serves as a template to recreate a copy of the original double strand.

The human genome consists of 46 chromosomes. There are two each of 1-22, the autosomal chromosomes. XX or XY are the two sex chromosomes.

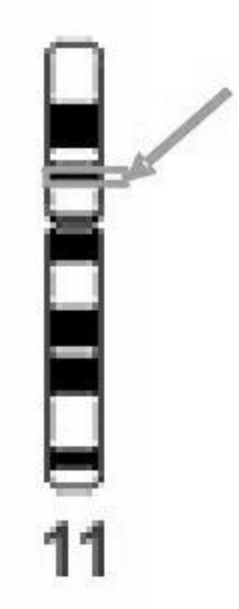


The human genome

An average chromosome is about 100,000,000 base pairs long. A gene is a small subset of a chromosome, typically a few thousand base pairs. Any random error in the sequence is likely to incapacitate the gene and in turn the organism; consequently there is relatively little variation in the genes among individuals.



Fragment of a gene



Locus TH01 in chromosome 11

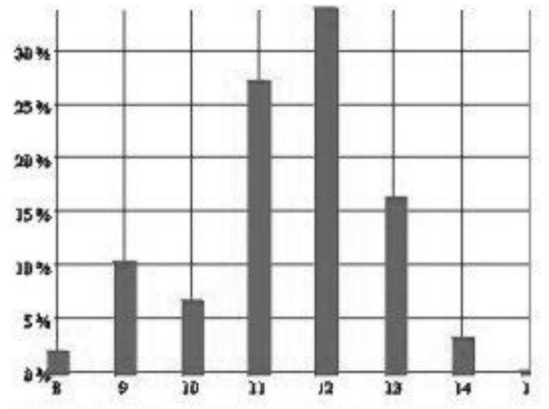
By contrast the genome is replete with junk DNA, variations in which carry no survival penalty and therefore a lot of variation has accumulated over time and can be used to distinguish individuals. Traditional (time span = handful of years) locations (*loci*, singular *locus*) in the genome, sometimes called *pseudogenes* are thus employed. The locus TH01 (near the gene for tyrosine hydroxylase) is defined as a particular stretch of about 200 bases at 11p15.5 – chromosome 11, the p or short arm, within band 15.

The ‘business’ part of TH01 consists of 6 to 11 short tandem repeats (STRs) of the same 4-base motif (tetramer): AATG, e.g.

AATGAATGAATGAATGAATGAATG

The variant forms, called *alleles* (analogous to isotopes or isomers) vary between individuals and also between the two chromosomes of a pair within an individual. For example, a person can have a TH01 type of 8,10.

A DNA profile is typically 13 or so loci: ({13,15}, {28,28}, {8,10}, …). Alleles vary greatly in frequency, average being around 1/5. The average match probability between unrelated people is therefore 0.1 per locus, 10^{-13} for a profile. Hence near-certain association of criminals with crimes.



Allele frequencies at D16S539 among 202 Caucasians

Naturally all this leads to myriad opportunities for computer tools to assist the DNA analyst. Biology and biological data, unlike engineering data, are inherently irregular, incomplete, and inaccurate and software must therefore be forgiving. For this reason and others there are many technical complications.

Programming note

DNA•VIEW (the package containing my 20 years of software development) includes 1000 or so population samples for various populations and loci. There is a menu with names for each, each name of course multi-part. Therefore I have found it very convenient to use an idea I learned from Eric Lescasse and extended myself for menu selection.

- *context selection* – as the user types characters, the menu shrinks to items that contain – not merely begin – with the typed phrase. To this I have added some elaborations of my own:
- *multi-phrased* – if the next character creates an impossible phrase, then the program tries splitting the input into two phrases and limits the list to items containing both phrases. There is a user keystroke (Tab) to force phrase-splitting but in practice is never necessary. Stream-of-consciousness typing gets the job done. If I want to find the Iraqi data compiled by the Amsterdam lab on locus D18S51 named: *D18S51 STR Amsterdam Iraqi 70 03/03/27 2pm* I can type `amstd18iraq` or `iraqd18` or many other possibilities.
- *case-observant but forgiving* – the list shrinks irrespective of the case entered, but the bounce bar (indicating the entry that will be chosen if *enter* is pressed) prefers an entry that matches case.
- *wandering bar* – the bounce bar tends to move around as you type more characters, on the theory that if it were already sitting on the desired choice you would have stopped typing.
- *multi-mode operation* – mouse click or arrow movement are always permitted of course.

Kinship identification

Besides individual variation, which DNA shares with fingerprints, DNA has another property of great utility for identification. It is inherited. Moreover, the rules of inheritance for DNA traits are extremely simple. Remember recessive genes? Incomplete penetrance? Forget them. With DNA you can see right down to the basics, every system is (well, nearly every – nothing in biology is regular) a co-dominant system. There is virtually no distinction between *phenotype* (what you can see) and *genotype* (what's on the chromosome).

Mendelian inheritance

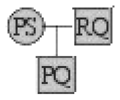
Autosomal loci are inherited by the child from the parents, randomly getting one of the two alleles from each parent. That's it. That's the Mendelian model, though as G.E. Box said, all models are wrong (but some models are useful). Sometimes we need to acknowledge –

Mutation

Occasionally (1/200 – 1/2000) an allele will gain or lose a single motif from parent to child. Rarely more. Very rarely, a less regular change.

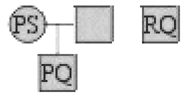
Paternity testing

The classic paternity question is to collect some genetic information (DNA profiles) and ask which of the two pedigrees below better explains the data. In the following, PS = mother, PQ = child and RQ = man.



Explanation 1
man is father

(2ps)(2qr)(¼) event



Explanation 2
man is not father, his Q is coincidence

(2ps)(2qr)(½q) event

The Likelihood Ratio for paternity is 1/(2q). If q=1/20, the data are 10 times more characteristic of the 'father' explanation

There are a few simple formulas, such as 1/q, for paternity analysis that are well known to people in the business.

Kinship testing

Ten years ago I decided to try solving a more general problem – given any two pedigree diagrams relating any set of people, to decide with DNA data which set of relationship is supported by the data (and by how much). It was difficult and I was only able to make progress on it when I hit on the idea of making all the computations algebraically.

Implementation of this idea is simple. Wherever the original program might add two or more probabilities, for example `+/P`, the new program simply used instead a defined function: `Plus/P`. The formulas in question turn out to be polynomials in n+1 variables (p, q, … for the frequency of each mentioned allele P, Q, …, plus one symbol, z, for the total frequency of all other alleles), hence the data structure to represent a polynomial is two arrays: a matrix of exponents with a column per variable and a row for each term, and a vector of coefficients for each term. It is easy to write `Plus, Times, Simplify` (collapse like terms), Over (divide two polynomials and remove common factors) and one function that turns out to have remarkable properties, `UnZ`, which removes all the instances of the variable z in favor of the equivalent expression 1-p-q-… . This is necessary because z should only occur internally; the user doesn't expect to see it. What is remarkable is that `UnZ` very much reduces the complexity of the polynomial. Dozens of terms reduce to a few. Still, the resulting expressions are funny-looking, wrapping to several lines when applied to an irregular collection of people who are hypothesised to be half-siblings, uncles, cousins, and inbred.

Body identification

It turned out that my toy found numerous practical applications. One of them is body identification: is the DNA profile for this corpse related or unrelated to reference father, sister, and aunt profiles? By 2001 the program and I had a sufficient reputation for this kind of problem that on September 12, when I managed to check my e-mail in London, I had an inquiry from the New York Office of the Chief Medical Examiner explaining that they foresaw a problem on which I would be able to help. Thus began a long saga including hundreds of difficult kinship identifications.

A new problem that arose was to sort out the thousands of DNA profiles arising from the World Trade attack – the victim profiles from anonymous bodies, versus the reference profiles from relatives or toothbrushes. Given a suspected identity it is all very well to use the Kinship program to test it, but kinship testing is inherently manual (the pedigree information is neither uniform nor accurate) so it is not practical to use the kinship program a million times with the wrong assumptions.

Disaster screening

Therefore it was apparent that a special 'screening' program would be needed. Thousands of profiles – maybe only hundreds in the beginning – uncertain information about the true family structures, toothbrushes shared, poor quality DNA leading to deficient profiles – clearly experimentation would be needed in order to discover heuristics that would usefully find probable candidate identities. You are ahead of me: an application tailor-made for APL.

As would you, I said, "Give me the data and I'll produce the identities in hours." And so it was, though it took a week or two to surmount bureaucratic hurdles and procure the data.

A work flow soon developed wherein new DNA profiles, obtained by various contracted laboratories, were funneled to me bi-weekly, I ran them through the constantly revised screening program, and passed the results to another software firm that installed them into a database at the OCME where caseworkers used Kinship and other tools to make final decisions on identification. This pattern continued for two years; eventually the flow of new DNA profiles, new information, and new identifications dried up.

Racial discrimination

I'll mention one more among the many interesting ideas that arise from DNA analysis, and that is the idea to determine race from DNA. The obvious idea may be to find the 'black' or 'Chinese' genes, but that is difficult, perhaps impossible, and entirely unnecessary.

Instead, several people besides myself noted that all of the minor or moderate variations of allele frequencies among populations are clues to source of origin. If a crime stain (e.g. semen of a rapist) includes an allele that is 20% more common among population X than population Y, then that is a slender clue supporting population X as the origin. Accumulated over 26 alleles this evidence is likely to be very powerful.

Particularly, 10 years ago I had an advantage over other researchers, thanks to my wide international customer base, of having handy lots of population data. Plus I speak APL. Consequently I was able, mainly by rapidly going through many computer experiments, to find the successful way (whose theoretical basis in population genetics I didn't then understand) to extract the racial inferences from DNA profiles. This is occasionally useful in police investigations, but more than that it is interesting in shedding light on human migratory history and also it turns out to have important application in regulating horse racing.

Programming notes and acknowledgements

I am grateful for the generosity and selfless collegiality of the APL community. In particular, Don Lagos-Sinclair completed my inept attempt to write my menu selection method in APL+WIN, Davin Church has shared his SQL tools and has worked hard to help me understand as have Morten Kromberg, Eric Lescasse and Adrian Smith. In addition there are many individuals who contribute to and encourage my learning simply by their contributions at the conferences.

I don't have a lot to offer in return. My days at the forefront of computer literacy probably expired about 1963. I would of course be glad to share the utilities or the ideas behind them that I have developed, though I don't have any code in shape to include as a workspace. Adrian’s wry comment a few years ago that Rex’s utilities are typically about 90% complete would need to be revised negatively for my stuff as it stands.

I would, though, like to recommend a package that John Walker pointed out a few years ago, namely a free but very professional installer package called Inno Setup and a companion macro preprocessor called ISTool. Examination of the script file [omitted] that I use to create an install file for DNA•VIEW will give you an idea of its capabilities, and to the extent that I have learned some of the ins and outs of using it I will be glad to answer questions from any APLer who wishes to call or e-mail to me.

APL: The next generation

by Ajay Askoolum

APLNext has a brand new APL product, Visual APL (VA). My initial exploration of VA is based on version 1.0 of the released product; this is also available for evaluation as a free time-limited version – see http://aplnext.com and http://apl2000.com for further details.

By way of clarification, the phrase 'next generation' in the title serves a dual purpose:

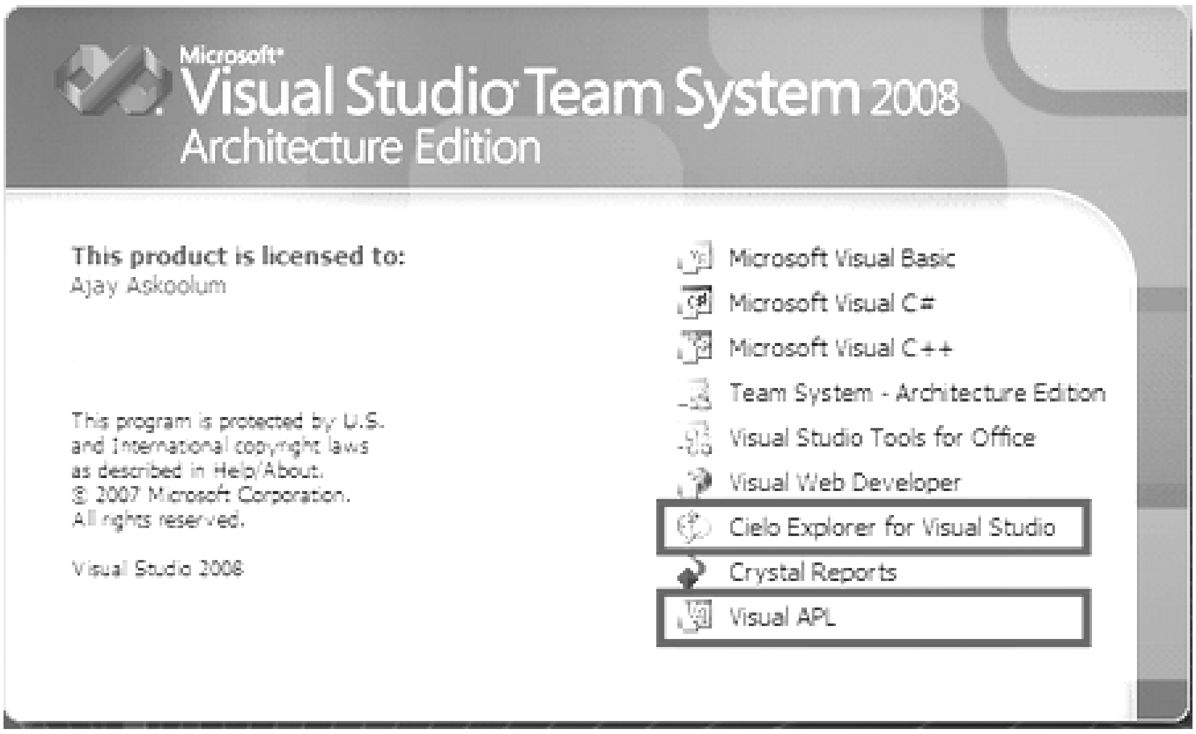
- It signals a radical new beginning for APL that does not lose sight of its origins.
- It heralds a new approach to APL application development, using managed code and the same integrated development environment (IDE) as the contemporary flagship development language C#.

VA heralds a revolutionary new generation of APL: its IDE is Microsoft Visual Studio 2008 (VS2008), it produces Common Language Runtime (CLR) assemblies, and, although it does not have a workspace, it retains mainstream compatibility with 32-bit Windows (legacy) APL+Win language semantics, including control structures. In short, VA takes to the .Net environment all the features of APL+Win except its special requirements such as the bespoke IDE, workspaces etc. At the same time, it confers the benefits of .Net to APL. VA is compliant with the ISO/IEC 13751 international standard for array languages.

The tag *legacy* is enigmatic since APL+Win is available with active support and undergoing further development; APL2000 released version 9.1 in June 2009.

Visual APL and VS2008

As Figure 1 shows, VA offers two pathways into VS2008 for .Net assemblies, Cielo Explorer for Visual Studio (CE) and Visual APL (VA); *Cielo* translates as heaven or sky.



Microsoft® Visual Studio Team System 2008
Architecture Edition

This product is licensed to:
Ajay Askoolum

This program is protected by U.S. and International copyright laws as described in Help/About.
© 2007 Microsoft Corporation.
All rights reserved.

Visual Studio 2008

- Microsoft Visual Basic
- Microsoft Visual C#
- Microsoft Visual C++
- Team System - Architecture Edition
- Visual Studio Tools for Office
- Visual Web Developer
- Cielo Explorer for Visual Studio
- Crystal Reports
- Visual APL

Figure 1: Visual APL – a native .Net language (Yes, this is APL!)

VA requires VS2008, which uses .Net Framework 3.5 by default. The VA installation automatically configures VS2008, including its menus and tools. VA requires activation via the Internet in order to enable code compilation to .Net assemblies. A permanent Internet connection is not a prerequisite after activation; this is a welcome feature, especially for laptop users.

There is a comprehensive set of help files on VA and VS2008; these are accessible via the help icon in the CE toolbar. The help files contain a tutorial on VA, which should appeal to existing and new APL developers alike. VA's customisation of VS2008 is non-intrusive; that is, it does not override any standard features such as shortcut keys etc. I can understand the complexity of redefining a standard VS2008 shortcut key such as F1.

VA is like C# and its solutions[1] can incorporate C# code. Any Component Object Model (COM) aware language can use an 'interop' class library (or Dynamic Link Library (DLL) written in VA; VS2008 projects can use assemblies (non-interop DLLs) directly.

It should come as no surprise that there is a new jargon to contend with; although this is not a tutorial, an elementary grasp of the recurring .Net/VS2008 jargon is relevant.

Namespace

A namespace is analogous to a directory but it contains other namespaces and classes. A VS2008 C# solution does not require a namespace but it is recommended as a means of organising a logical grouping of names or identifiers as this empowers the developer to keep control and to avoid name collisions. Unlike a folder, a namespace does not exist physically; it exists in memory at run time.

Class

A class is a child of a namespace. A class may contain other classes but always contains member elements. A class has properties, that is, data that the class exposes. A class also has behaviours, that is, functionality or methods/functions, and events.

Type

A class is a type; however, the term *type* describes modular units used to build an assembly, that is, classes, predefined or internal CLR types, delegates etc.

Object

An object is an instance of a class, that is, it exists at runtime only; an object cannot modify the class whose instance it is, rather, it uses the class as a black box.

Member

The properties and behaviours of a class are its members.

Solution

A solution is Microsoft's term for the collection of all the building blocks that produce an assembly. Erstwhile, the common terminology was project.

Assembly

An assembly is what a solution produces, an EXE or a DLL, and what the CLR uses to execute the application.

Scope

This term defines the lifetime of a variable: that is, the scope of a variable is the block of code in which it exists. In C#, this means within the braces that follow a function name. A variable defined within a class but outside of any functions within it has the whole class as its scope; this is also known as a field.

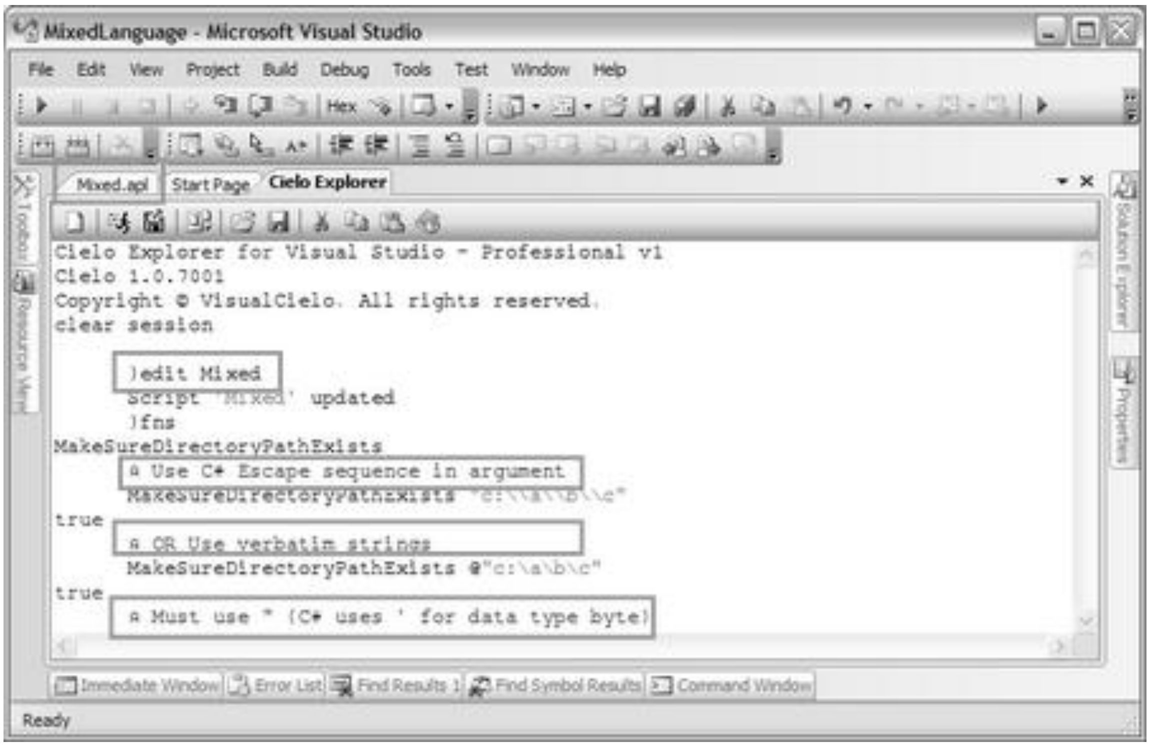
The Visual Studio factor

What does Visual Studio do for APL?

- It makes APL a mainstream programming language, in the same context as the flagship C# language.
- It removes the obstacle of the learning curve that a bespoke IDE imposes on developers of another .Net language who want to include APL in their arsenal of skills. There are more .Net than APL developers.
- It adds transparency to the management of APL source code: VS2008 deals with APL source code much as it deals with, say, C# code. This includes version management using any source code-management software that integrates with VS2008, including Visual Source Safe and Source Vault.
- It permits neither the saving of the runtime state of an application nor of arbitrary and complex data – in or out of scope. Notoriously, the APL workspace saves objects (such as namespaces), variables and even the dynamic execution stack – that is, the state of memory, which cannot be easily recreated.
- VS2008 has facilities for producing documentation in Extensible Markup Language (XML) format from inline code comments found in the code files; add-ins produce help (CHM) files from the XML files.

Is it APL?

The salient characteristics of APL are that it is a language based on symbols, is a dynamically-typed language, has an interactive development environment, and processes vectors and arrays, simple or nested, naturally. Yes, VA is APL in all these respects but there are crucial differences: it does not have a workspace, the localisation rules are reversed, that is, the inclusion of a name in the header makes it global, and index origin is zero by default. Remarkably, it is possible to mix APL and C# code in APL functions: mixed-language programming is a reality with VA – see Figure 2.

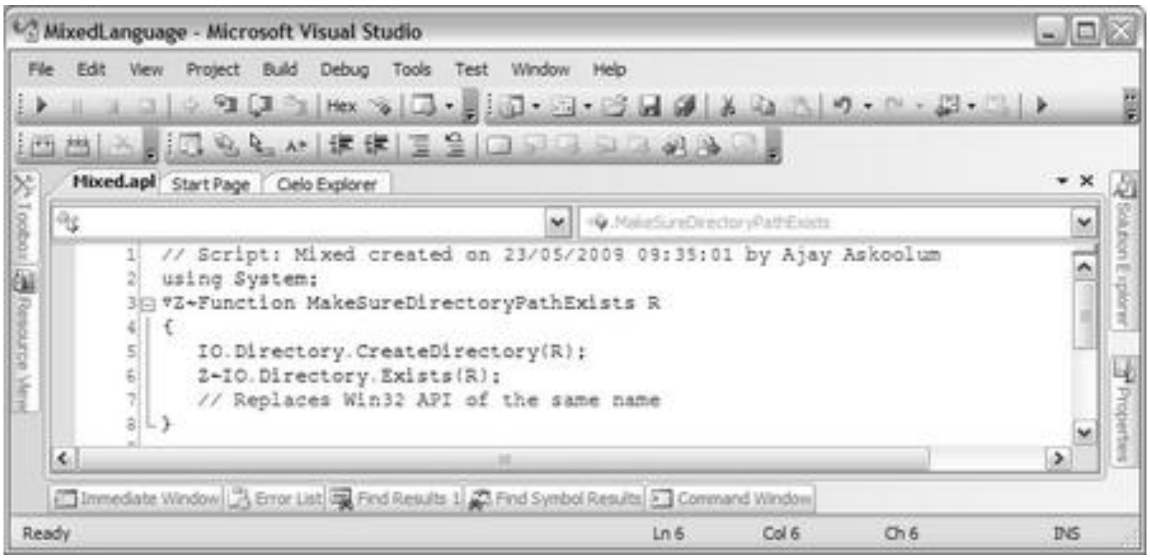


```
Cielo Explorer for Visual Studio - Professional v1
Cielo 1.0.7001
Copyright © VisualCielo. All rights reserved.
clear session

      )edit Mixed
      Script 'Mixed' updated
      )fns
MakeSureDirectoryPathExists
      ⍝ Use C# Escape sequence in argument
      MakeSureDirectoryPathExists "c:\\a\\b\\c"
true
      ⍝ OR Use verbatim strings
      MakeSureDirectoryPathExists @"c:\a\b\c"
true
      ⍝ Must use " (C# uses ' for data type byte)
```

Figure 2: Mixed-language programming

The code for the function `MakeSureDirectoryPathExists` is shown in Figure 3.



```
// Script: Mixed created on 23/05/2009 09:35:01 by Ajay Askoolum
using System;
∇Z←Function MakeSureDirectoryPathExists R
{
   IO.Directory.CreateDirectory(R);
   Z←IO.Directory.Exists(R);
   // Replaces Win32 API of the same name
}
```

Figure 3: Mixed code

The mixed-language reality applies to *both* the APL code and the semantics for calling the code.

VA places APL squarely into the mainstream software development arena and without the handicap of built-in proprietary functionality such as a bespoke development environment, runtime system, and hybrid quad functions that act as wrappers for system resources.

Legacy comparison

As mentioned, VA does not have a workspace; its code is stored in Unicode text files managed by VS2008. Native files are supported, albeit they are superfluous as the C# file management facilities are superior, and component files – renamed Share File System – are available but are incompatible with corresponding APL+Win component files. VA and APL+Win cannot read each other's component files.

VA uses the same keyboard mapping for APL symbols as APL+Win but the APL fonts are different – VA uses a Unicode font. The IDE provides support for pasting APL+Win code into VA projects – the menu option `Edit | Paste` APL+Win copies APL+Win code from the keyboard and remaps the fonts seamlessly. This menu option is doing something subtle and intricate – it copies the code from other APL interpreters mostly correctly too.

Legacy APL uses the semicolon for one purpose alone, namely to separate names in function headers. Names included in the header disappear as soon as the function goes off the execution stack; names not included in the header but created within the function persist. VA retains this convention but in reverse: names included in the header persist and all other variables are local. In other words, in VA the semicolon reverses the localisation rules.

In legacy APL, index origin, `⎕io`, is 1 by default. In VA, it is 0 by default, in common with C#. Variables are local by default; however, there is a new syntax – that is, deliberate coding is required – for creating global variables.

Contrary to what might be the initial reaction to these changes, the impact on migration can be minimal, depending on the quality of the existing code. There are tools provided when Visual APL is installed for migrating legacy applications, workspaces, and component files into the managed code environment.

APL Font

VA uses a Unicode font that supports all the APL characters; there are two new symbols, ≅ (*approximately equal to*) and ƒ (*guilder*). In addition, `=` (*equal to*) is no longer a comparison but an assignment operator – the comparison operator is `==`, as in C#. As far as I can see, there is a single instance where APL symbols create a conflict within VS2008. APL uses semicolon both to separate names in APL function headers and as a statement terminator in C#.

APL keyboard

VA uses the same keyboard layout and shortcuts as APL+Win and the APL symbols are accessible via the Alt key. The key combination `Alt+F` produces ƒ and `Alt+5` produces ≅.

Data typing

VA introduces a number of new data types – see Table 1 - and manages the interface between the dynamically- and strongly-typed arenas seamlessly; however, this presents a new learning curve for the developer accustomed to legacy APL.

Table 1: Visual APL data types

Code	Description	Comment
11	Boolean (true/false, not bit)	No longer indicates binary data
81	Bytes	
82	chars (compatible with 82 in existing system)	
83	String (compatible with 82 in existing system)	These types correspond to the CLR predefined types.
163	short (Int16, 16-bit integer)	
164	Ushort (UInt16, unsigned short)	
323	int (Int32, 32-bit integer, default)	
324	uint (UInt32, unsigned int)	
325	float (Single, 32-bit real)	
643	long (Int64, 64-bit integer)	

Table 1: Visual APL data types

Code	Description	Comment
644	Ulong (UInt64, unsigned long)	
645	double (Double, 64-bit real, default)	
1285	Decimal (128 bit real)	
807	Object (serialized object)	
99999	no code available for data type	This facilitates error trapping.

The type of a variable can be queried using the C# method `GetType()` or `⎕dr`; the latter is incompatible with its legacy counterpart. Consequently, there is another hurdle to porting legacy applications, namely, `'wrapl' ⎕dr` and `'unwrapl' ⎕dr` cannot be used to transfer data because the APL+Win serialisation is proprietary to the Win32 environment. However, VA offers the facility for representing its data, including numeric and heterogeneous arrays, in XML format; therefore, if migration is an issue, APL+Win needs to render its data using the same schema so that VA can read it. The XML form of variables is held in a file and can be edited within VS2008 and any changes are automatically reflected in the session. Refer to the `)xmlout` system command in the VA help file or research *serialization* in C# for background information.

Benefits of the radical departures from legacy APL

An experienced APL developer might instinctively condemn the design of VA as a betrayal of the legacy of APL. However, I take a very different stance – I applaud the design decisions for the following reasons:

- The direct implication of the design is that legacy applications cannot simply be migrated into the new environment; that is the catalyst for modernising legacy applications.

- For too long APL developers have indulged in creating applications that are a homogeneous tangle of the presentation, data, and business tiers which have proved notoriously difficult to maintain and modernise.
- APL developers need to learn to integrate standard solutions – written, debugged, and maintained by Microsoft and others at their own cost – into APL applications in order to give the applications a generic look and feel. It is time APL utility functions that re-invent readily available solutions were relegated to history.
- It is no longer economically viable for APL applications do things in a different way; modern applications have a transparent design that focuses on the ease of maintenance, or evolution, and the acquisition and transfer of data.
- The intrinsic credentials of legacy APL, strong and legitimate as they are, has long ceased to sway decisions in favour of APL.
- This APL fully acknowledges the prior experience of newcomers to the APL language. For example, .Net developers' understanding of VS2008, application design, scoping, and general experience of .Net classes transfers directly to VA.
- Equally, APL developers can use their experience of core APL with VA. The Cielo Explorer provides an interactive immediate mode for APL as a tool of thought. Support for legacy features such as component files and `⎕wi` means that APL developers can be readily productive in the new development environment. With experience, developers should gradually learn to favour and adopt platform solutions.
- With VA, applications have the same characteristics as any other VS2008 application; the specialist requirements of legacy APL, such as workspaces, component files and a dedicated runtime system, simply do not apply; there is no stick with which to bludgeon APL. VA assemblies require .Net Framework 3.5 and other DLLs.
- Perversely, the incompatibility of the Share File System with component files is also welcome. The .Net platform offers ADO.NET for access to

databases, which provide open access to application data whereas component files blocked open access; this will prompt a redesign of legacy APL applications. Although it is expedient to hold data as arrays within an APL, the nature of data is scalar in other languages that are growing in influence.

Getting started

VA is a new product working within an IDE that may also be completely new to traditional APL developers.

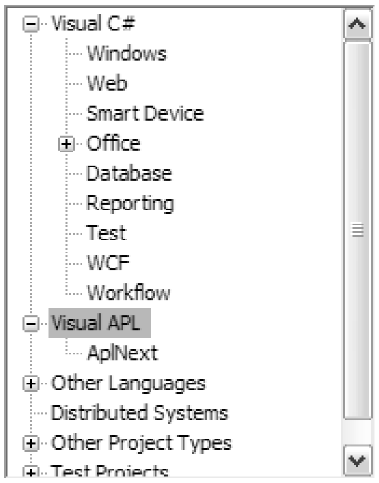


Figure 4: Language options

In VS2008, `File | New Project` offers the language options shown in Figure 4.

Installation places documentation for VA in a folder `Documentation`. The document `tutorial.chm` provides a general overview of VA.

Webcasts and other worked examples can be found on the APL2000 Forum. [2]

Visual APL pathways

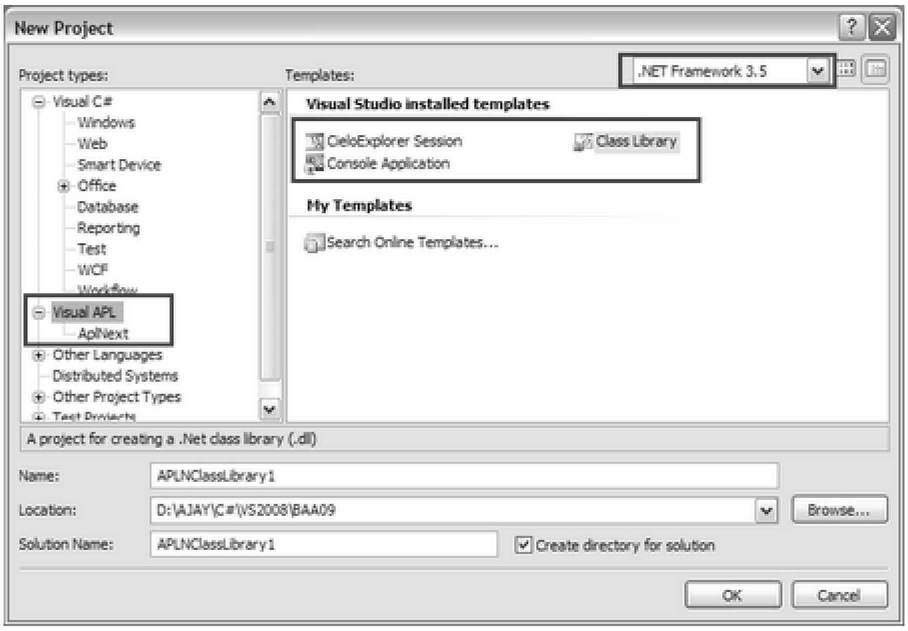
As far as I can make out, VA offers several options for using APL in VS2008.

- The migration of existing APL applications to a managed code environment with little disruption; this includes native and component files and Win32 Graphical user interfaces. VA supports all the quad functions, including `⎕wi`, with the same familiar syntax; however, such functionalities are implemented in new managed code class libraries – the Visual APL foundation class libraries.

- The Lightweight Array Engine (LAE) describes the core VA foundation classes that can be included in .Net projects; this makes APL array facilities directly available to non-APL VS2008 projects.

- Cielo Explorer is available from any VS2008 project with the menu option `View | Other Windows | Cielo Explorer`. This option provides interactive APL and the facility for producing script files. Code that is written in script files can be debugged and tested in CE and pasted into files belonging to other VA templates.

VA provides templates for types of projects; see Figure 5.



New Project

Project types:
- Visual C#
  - Windows
  - Web
  - Smart Device
  - Office
  - Database
  - Reporting
  - Test
  - WCF
  - Workflow
- Visual APL
  - AplNext
- Other Languages
- Distributed Systems
- Other Project Types
- Test Projects

Templates: .NET Framework 3.5

Visual Studio installed templates
- CieloExplorer Session
- Console Application
- Class Library

My Templates
- Search Online Templates...

A project for creating a .Net class library (.dll)

Name: APLNClassLibrary1

Location: D:\AJAY\C#\VS2008\BAA09

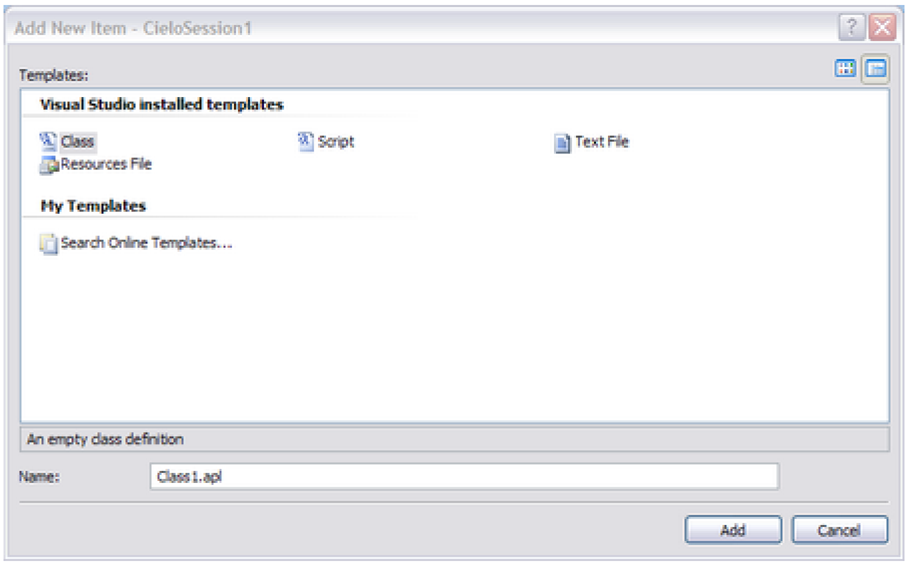
Solution Name: APLNClassLibrary1

[✓] Create directory for solution

OK Cancel

Figure 5: Visual APL templates

The CieloExplorer Session exposes another set of templates, see Figure 6. This is slightly confusing as it is another type of project: see Figure 4.



Add New Item - CieloSession1

Templates:

Visual Studio installed templates
- Class
- Resources File
- Script
- Text File

My Templates
- Search Online Templates...

An empty class definition

Name: Class1.apl

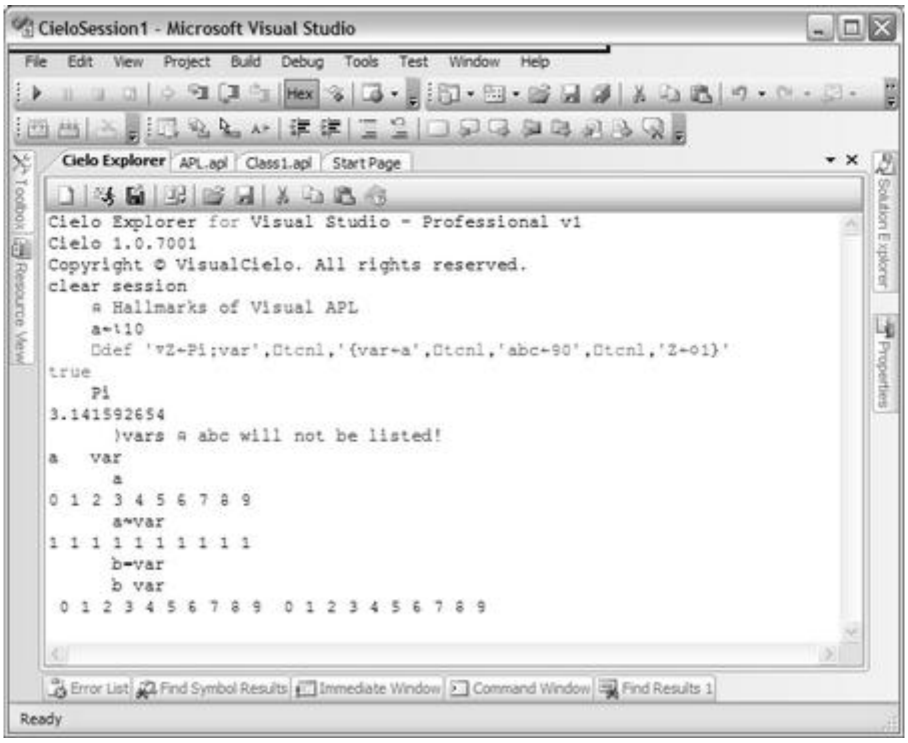
Add Cancel

Figure 6: CieloExplorer Session templates

Cielo Explorer

With an active installation (an Express version that has not expired or the commercial product) of VA, CE is available with `View | Other Windows | Cielo Explorer` in VS2008 at any time, even when the project is not based on VA.

CE is the closest thing that is available to an interactive APL session with an important difference: the session itself *cannot* be saved as the ubiquitous workspace but only as a Unicode text file. However, functions and variables may be defined in script files, which can be opened and saved, and with the use of the Visual APL Lightweight Array Engine (LAE), incorporated as late-bound executables in any .Net project. CE serves as an interactive APL workbench for developing and debugging APL code destined for incorporation into a class file, consistent with the way VS2008 works. Figure 7 shows an interactive session.



```
Cielo Explorer for Visual Studio - Professional v1
Cielo 1.0.7001
Copyright © VisualCielo. All rights reserved.
clear session
    ⍝ Hallmarks of Visual APL
    a←⍳10
    ⎕def '∇Z←Pi;var',⎕tcnl,'{var←a',⎕tcnl,'abc←90',⎕tcnl,'Z←○1}'
true
    Pi
3.141592654
     )vars ⍝ abc will not be listed!
a   var
     a
0 1 2 3 4 5 6 7 8 9
     a=var
1 1 1 1 1 1 1 1 1 1
     b=var
     b var
 0 1 2 3 4 5 6 7 8 9  0 1 2 3 4 5 6 7 8 9
```

Figure 7: Hallmarks of Visual APL

The major surprises are:

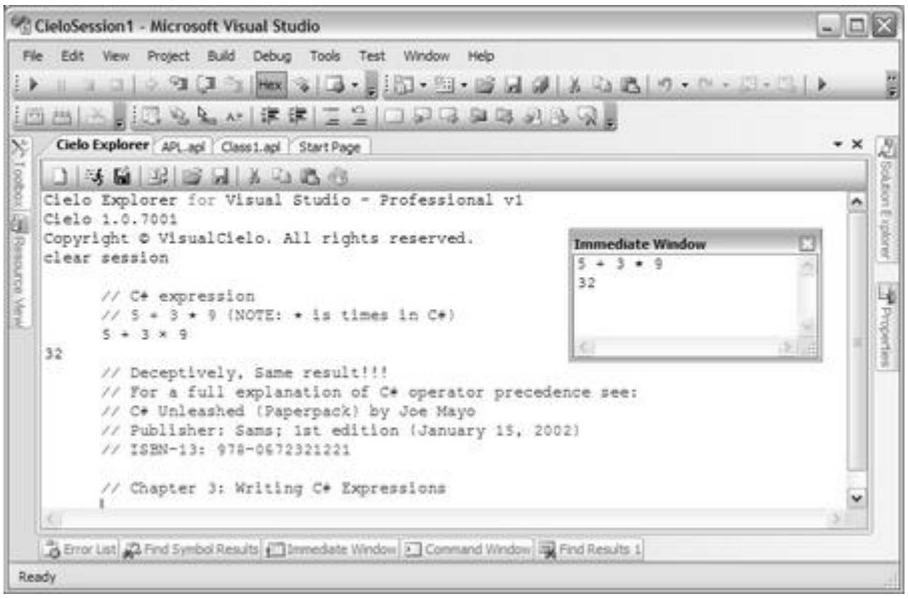
- Index origin `⎕io` is zero by default; see the value of variable `a`.
- Semicolon globalises variables; `var` appears in the header of function `Pi` but is available in the session after the function has run.
- A pair of braces following the header defines the scope of variables; thus, the scope of `abc` does not extend into the session.
- The comparison operator `=` serves a new purpose, see 'Assignment by value and by reference'.

A note on multi-language programming

Although VS2008 hosts a number of different languages, any project can incorporate literal code from just one language at a time; compiled code from another language can be incorporated either by including a reference to its assembly or by including the project itself.

This applies to VA too. However, VA shares the characteristics of C# and does allow the incorporation of C#-like code into VA script and project files. Although C# does not understand anything about APL, VA does integrate C# concepts.

- C# and VA are both case-sensitive.
- The order of execution is different. VA works from right-to-left but C# has a hierarchy of operator precedence and is more complex. Figure 8 shows the evaluation of the same expression in Cielo Explorer and the Immediate Window of VS2008: tempting as the conclusion is, VA and C# do not work in the same way.



```
Cielo Explorer for Visual Studio - Professional v1
Cielo 1.0.7001
Copyright © VisualCielo. All rights reserved.
clear session

      // C# expression
      // 5 + 3 * 9 (NOTE: * is times in C#)
      5 + 3 × 9
32
      // Deceptively, Same result!!!
      // For a full explanation of C# operator precedence see:
      // C# Unleashed (Paperpack) by Joe Mayo
      // Publisher: Sams; 1st edition (January 15, 2002)
      // ISBN-13: 978-0672321221

      // Chapter 3: Writing C# Expressions
```

```
5 + 3 * 9
32
```

Figure 8: Order of execution

- Both languages have reserved typewriter symbols (C# has `/*`, `@` etc) and VA has its own symbol set. In a VA project incorporating C#-like code, the APL symbols override the C# meaning. Thus, `/` is *scan/select* and never *divide* as in C#.
- C# uses `\` for starting an escape sequence; for example, `\r\n` embedded within a string will be interpreted as carriage return followed by linefeed.
- Verbatim strings – that is, strings declared with prefix `@`, see below – confer no intrinsic meaning to escape sequence characters.
- C# uses `//` for a comment; this works like the APL. Multi-line comments are enclosed within the `/* */` pair. With VA ensure that either pairs of

characters are followed by a space in order to avoid confusion with the corresponding APL operators.

- Both VA and C# use square brackets for indexing but VA uses semicolons to separate the indices, whereas C# uses comma.
- C# uses double quotes for enclosing literals of type *string* and single quote for enclosing data of type *char*. VA can use single and double quotes interchangeably within an APL context but in some contexts, it needs to follow the C# convention. Consider the following examples:

  C#

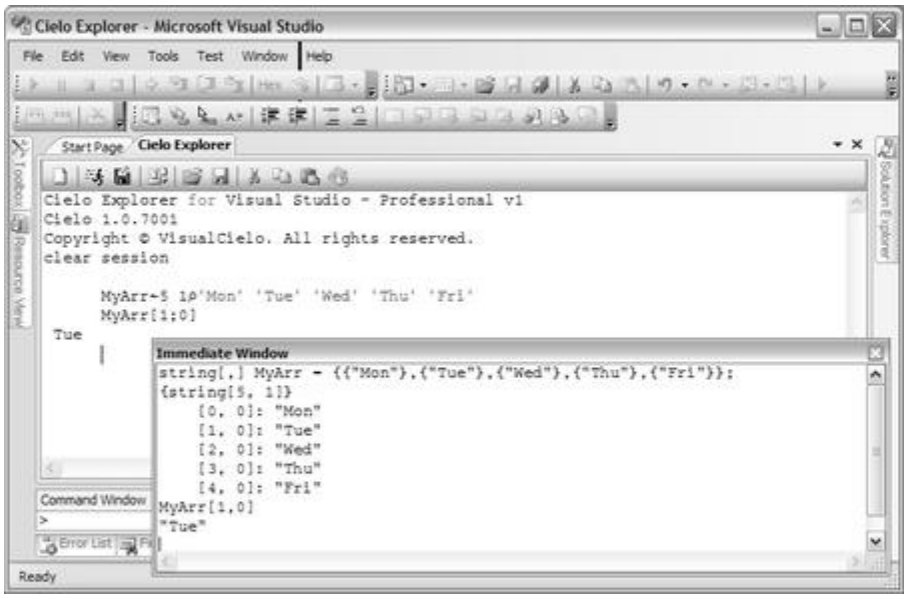
  ```
  System.IO.Directory.GetFiles("C:\AJAY","*.*",
                       System.IO.SearchOption.AllDirectories)
  ```

  Unrecognized escape sequence

  Errors because the first argument should be either `"C:\\AJAY"` or `@"C:\AJAY"`

  VA

  ```
  ⎕System.IO.Directory.GetFiles("C:\AJAY","*.*",
                       System.IO.SearchOption.AllDirectories)
  686
  ```
- C# has vectors (single-dimensional arrays), arrays (multi-dimensional arrays) and jagged arrays. Jagged arrays are arrays of arrays like VA's nested arrays: however, arrays in a C# jagged arrays must all be of the same type, although not necessarily the same shape or dimension, whereas VA's nested array can mix types. With an APL two-dimensional character array, the second index specifies a single column; with C#, the second index specifies the whole element: see Figure 9.


Cielo Explorer - Microsoft Visual Studio
File Edit View Tools Test Window Help
Start Page Cielo Explorer
Immediate Window
Command Window
Error List
Ready

```
Cielo Explorer for Visual Studio - Professional v1
Cielo 1.0.7001
Copyright © VisualCielo. All rights reserved.
clear session

      MyArr←5 1⍴'Mon' 'Tue' 'Wed' 'Thu' 'Fri'
      MyArr[1;0]
 Tue
```

```
string[,] MyArr = {{"Mon"},{"Tue"},{"Wed"},{"Thu"},{"Fri"}};
{string[5, 1]}
    [0, 0]: "Mon"
    [1, 0]: "Tue"
    [2, 0]: "Wed"
    [3, 0]: "Thu"
    [4, 0]: "Fri"
MyArr[1,0]
"Tue"
```

Figure 9: Simple or nested array?

Assignment by value and by reference

In legacy APL, the interpreter manages the process of assignment and reassignment internally. With VA, like C#, the developer can control whether an assignment takes place by value or by reference. In C#, value types derive from `System.ValueType` whereas reference types derive from `System.Object`: value types are managed on the stack and object types on the heap. A good grasp of the trade-offs between assignment by value and reference comes with experience and is highly relevant from the point of view of debugging and fine-tuning the performance of an application.

Although the consequences of this notion are far-reaching, for the moment, it will suffice to have a basic understanding:

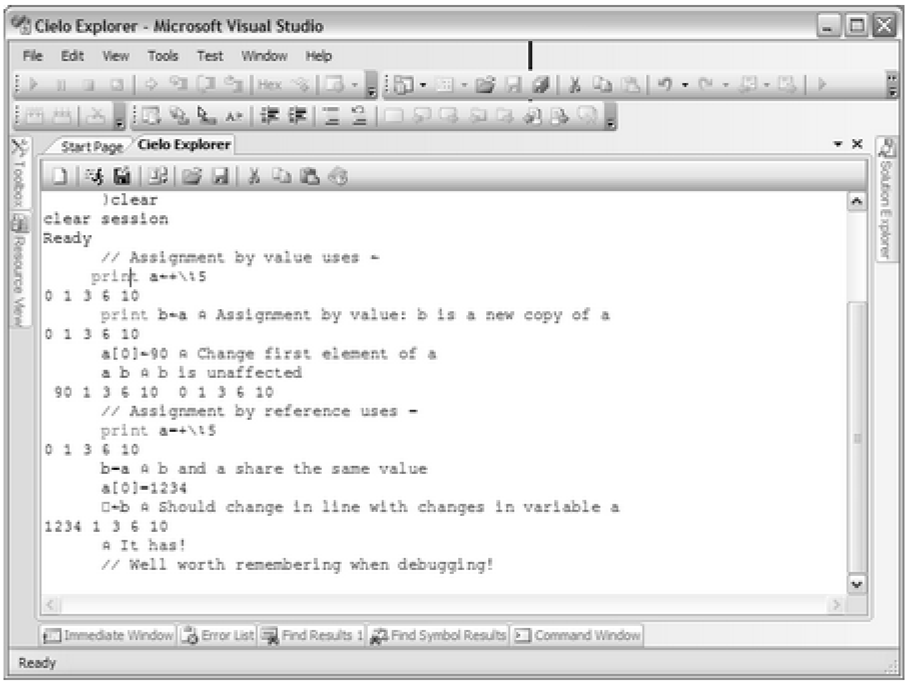
A variable of type value

contains a value and if that variable is reassigned to another variable of type value, the value is replicated: subsequent changes to one variable do not affect the other. This happens on the stack. With VA, ← makes an assignment by value.

A variable of type reference

points to (refers to) the memory location where the actual variable (object) is contained. And if that variable is reassigned to another variable of type reference, the second copy inherits the reference (held on the heap) to the same memory location. The memory location is not replicated, therefore, a change to one is reflected in the other and both copies remain identical. With VA, = makes an assignment by reference.

Figure 10 shows a simple example that illustrates the basic difference between the two assignment methods. Although APL+Win used assignment by reference for copies of variables, the difference is that APL+Win managed the transition to value implicitly but the developer has to manage this with VA.



```
      )clear
clear session
Ready
      // Assignment by value uses ←
     print a←+\⍳5
0 1 3 6 10
      print b←a ⍝ Assignment by value: b is a new copy of a
0 1 3 6 10
      a[0]←90 ⍝ Change first element of a
      a b ⍝ b is unaffected
 90 1 3 6 10  0 1 3 6 10
      // Assignment by reference uses =
      print a=+\⍳5
0 1 3 6 10
      b=a ⍝ b and a share the same value
      a[0]=1234
      ⎕←b ⍝ Should change in line with changes in variable a
1234 1 3 6 10
      ⍝ It has!
      // Well worth remembering when debugging!
```

Figure 10: Assignment by value and reference

New quad functions: division by zero

Besides a number of new keywords like print, VA adds a number of new quad functions to the language vocabulary and discards a number of others such as `⎕cr` and `⎕vr`.

By default, legacy APL yields `DOMAIN ERROR` on encountering a division by zero where the numerator is not `0`, and `0÷0` is `1`. In contrast, VA adopts the .Net behaviour: any division by zero returns `0` by default. However, this behaviour can be overridden. The legacy behaviour can be implemented by the following assignment `⎕dbz←1`; this system variable can take five possible values, which help to customize the output of division by zero (the help file shows the available options).

The CE toolbar

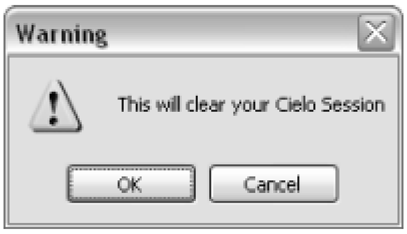
An acquaintance with the CE toolbar is necessary to be able to manage the log files with ease and to access the help files. The toolbar has ten icons, whose functions are explained in Table 2.

Table 2: The Cielo Explorer toolbar

	New	Clear the Explorer session. The system command `)clear` achieves the same purpose, as does the undocumented command `)off`.
	Run Cielo Script	Prompts for a script file and fixes its content in the current Explorer session. It does not add the file to the current VS2008 project.
	Load Cielo File	Like `Import Assembly` but it also adds several other using directives requiredby the assembly that is imported to the session.
	Import Assembly	Adds a reference to an existing assembly into the current session as follows: `refbyfile@"C:\Program Files\AplNext\VisualAPL\APLNext.APL.dll"`
	Load Session Log	Prompts for the log file name and brings its textual content into the CE session.
	Save Session Log	Prompts for the log file name and saves the textual content of the session to it.
	Cut	Copy highlighted section of the session to the clipboard and then delete selection.
	Copy	Copy the content of the clipboard at the cursor location.

	Paste APL+Win	Copy APL+Win code: transparently remaps the APL fonts.
	Invoke help files	Opens the VA help files. (This is also available as a menu option: `Start	Programs	APLNext…`

In addition, CE has a new set of system or session commands such as `)classes`, `)cd` etc.; these are documented in the help file which includes a chapter on CE. A particular command is noteworthy: if you are building a class library in VS2008 and testing it within CE; any attempt to rebuild the library will cause VS2008 to complain because the DLL will be in use. The command `)clear` frees the DLL for recompilation; you do not have to close the session as with APL+Win.


Warning

This will clear your Cielo Session

Figure 11: Clearing the Cielo Session

If it is desirable to clear the session at any time, the command `)clear` resets the active session with a warning, shown in Figure 11.

In fact, when working with script files, it is recommended that the CE session is cleared before pressing `Ctrl+E,E` to fix their content; this will ensure that the objects in the session are from the current script files; if a script file fails, the session will continue to hold the objects last created successfully.

If you are inclined to type `)off` by force of habit, CE supports the command: it provides the same warning as `)clear` but it is much more destructive as it closes the current project. All windows in VA projects work with the VS2008 IDE.

The Cielo Explorer or Editor windows are configurable as a Dockable, Floating, or Tabbed Document within VS2008, as shown in Figure 12; note that this is shown with a C# project open, as evident by the `.cs` file extensions.

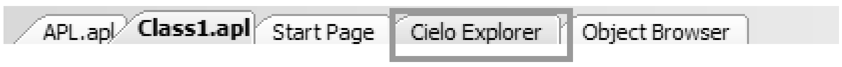
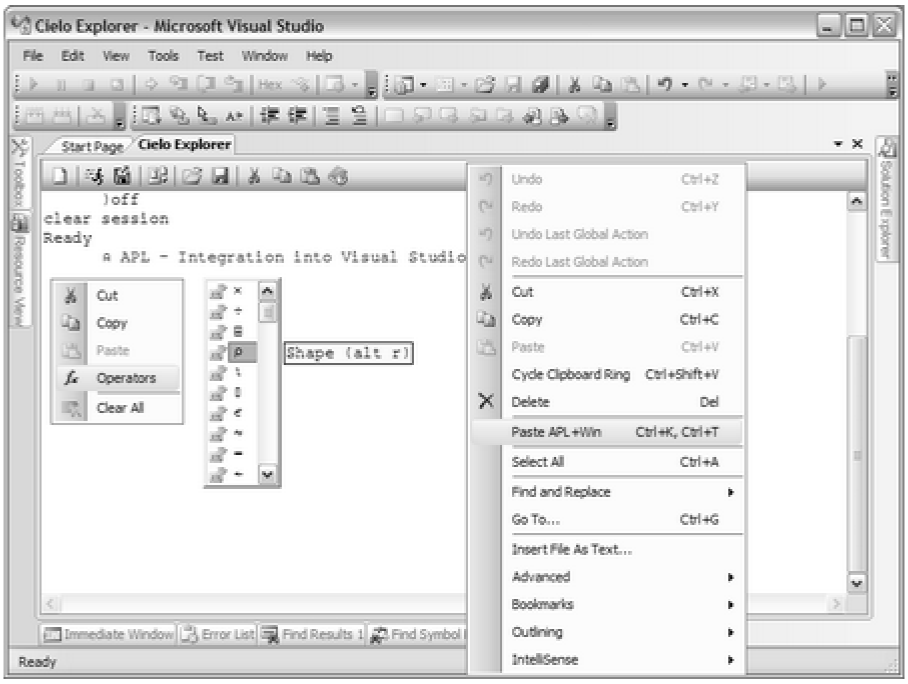


Figure 12: Windows in VS2008

Although it is surprising to see `Load` instead of `Open`, the extent of integration is very impressive: see Figure 13.



```
      )off
clear session
Ready
      ⍝ APL - Integration into Visual Studio
```

Figure 13: Integration in Visual Studio IDE

VA uses a Unicode font, which is not compatible with the font used by APL+Win; therefore, `Ctrl+C` will not paste APL+Win (or other APL's) code correctly: this facility is doing something subtle and getting it right! VA and APL+Win use the same keyboard mapping for APL characters.

Cielo script files

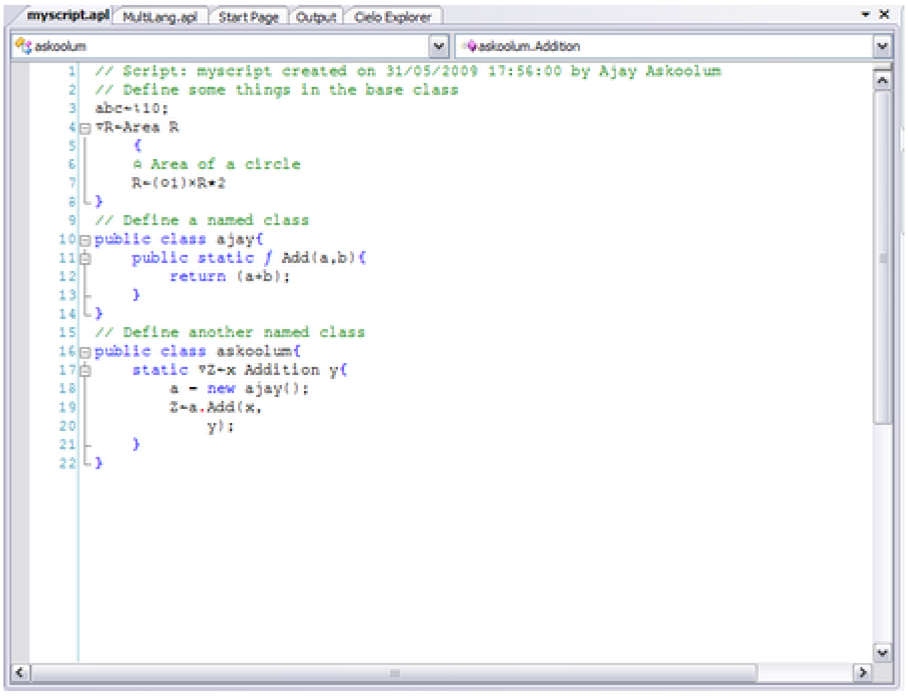
Although CE's inability to save sessions, except as log files, may appear highly restrictive at first, this is in fact a bonus for two sound reasons:

- Script files store variable and function definitions in the base or named classes independently of session activity. This creates a higher degree of transparency in application code and better documentation.
- The clutter of the session activity is not saved as might (or does!) happen with `)save`. In other words, the scope of a session cannot span across sessions.

Using script files

A script file has extension `apl`; this is recognised by VS2008 in that double-clicking the file within Explorer will launch VS2008 and open the file. Figure 14 shows a script file within VS2008.

- The code in a script file is fixed in CE by the keyboard shortcut `Ctrl+E,E`. This has the effect of overwriting synonymous functions and variables in the base or named classes; other session objects are unaffected.
- A script class is created using the session command `)ed MyScript` within CE; this either creates the file and opens it or simply opens it if it exists. The file name that is created has extension `apl`.
- The easiest way to grasp the concept of a class is to visualize it like a subfolder in the filing system. Functions and variables defined outside of a class are akin to being in the root folder. In the example, shown in Figure 14, the script defines `abc` and `Area` in the root and two other classes `ajay` and `askoolum`, where the latter cross references the former.
- Note the valence (signature in C# parlance) of the function `ajay.add` and `askoolum.Addition`; see 'Valence and signatures' for more details.
- Note lines `[7]` and `[19 20]`: the last has the C# statement terminator (;) and the first does not. Unlike C#, where they are mandatory, VA statement terminators are optional for statements on the same line. My own preference is to use the statement terminator, as it is good practice. The difference between a terminator and a separator (⋄ or diamond) is that a terminator denotes the end of a statement, shown in lines `[20]` and `[21]`, which may span several lines, and a statement separator denotes the start of another statement on the same line.



```
 1 // Script: myscript created on 31/05/2009 17:56:00 by Ajay Askoolum
 2 // Define some things in the base class
 3 abc←⍳10;
 4 ∇R←Area R
 5      {
 6      ⍝ Area of a circle
 7      R←(○1)×R*2
 8 }
 9 // Define a named class
10 public class ajay{
11      public static ƒ Add(a,b){
12           return (a+b);
13      }
14 }
15 // Define another named class
16 public class askoolum{
17      static ∇Z←x Addition y{
18           a ← new ajay();
19           Z←a.Add(x,
20                y);
21      }
22 }
```

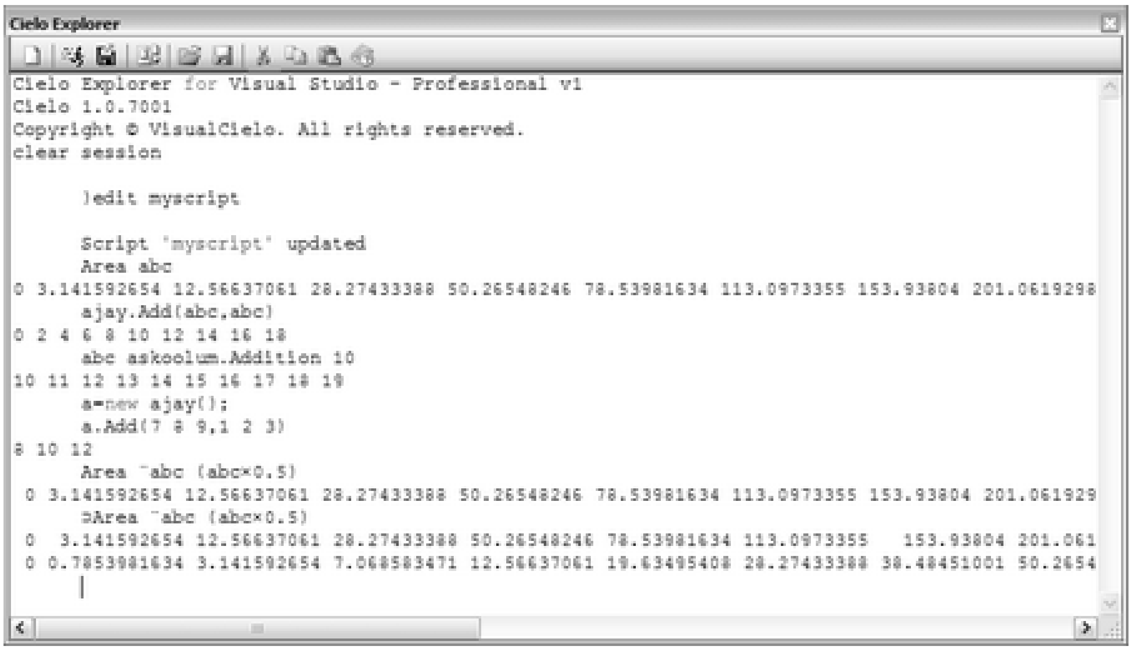
Figure 14: A simple script file

Watch out If, having successfully fixed the contents of a script file, you introduce new errors while making further changes, the definitions based on the older file remain in the session.

As with legacy APL, a system command cannot be followed by a comment; however, a command `)edit MyScript ⍝` treats the comment as part of the file name. This should be fixed, as it causes difficulty in referring to the file by name.

MyScript.apl in CE

The keyboard shortcut `Ctrl+E,E` fixes or updates the contents of the script file into the session; see Figure 15.



```
Cielo Explorer for Visual Studio - Professional v1
Cielo 1.0.7001
Copyright © VisualCielo. All rights reserved.
clear session

      )edit myscript

      Script 'myscript' updated
      Area abc
0 3.141592654 12.56637061 28.27433388 50.26548246 78.53981634 113.0973355 153.93804 201.0619298
      ajay.Add(abc,abc)
0 2 4 6 8 10 12 14 16 18
      abc askoolum.Addition 10
10 11 12 13 14 15 16 17 18 19
      a=new ajay();
      a.Add(7 8 9,1 2 3)
8 10 12
      Area ¨abc (abc×0.5)
 0 3.141592654 12.56637061 28.27433388 50.26548246 78.53981634 113.0973355 153.93804 201.061929
      ⊃Area ¨abc (abc×0.5)
 0  3.141592654 12.56637061 28.27433388 50.26548246 78.53981634 113.0973355   153.93804 201.061
 0 0.7853981634 3.141592654 7.068583471 12.56637061 19.63495408 28.27433388 38.48451001 50.2654
      |
```

Figure 15: `myscript.apl` in session

Some observations on what Figure 15 shows:

- Reference to classes can be relative, e.g. `ajay.Add`, or use an alias e.g `a = new ajay();`.
- The session (or root) objects are universally available, that is, in the session and to classes using relative reference.
- The APL functionality works exactly as expected with conformable scalar, vector, and nested arguments.

Valence and signatures

The valence of legacy APL functions classifies them into *niladic* (no arguments), *monadic* (one argument, always on the right), or *dyadic* (one on the left and one on the right).

Dyadic functions can be ambivalent: the left-hand argument can be omitted. The code must test this using `⎕nc`. VA also has `⎕monadic` and `⎕dyadic` and allows for an argument's absence by using a default value. This has proved restrictive for the following reasons:

- In an APL environment, multiple arguments are passed via a nested variable, thereby coercing a schema where the arguments are anonymous and positional.
- In a non-APL environment, functions calls never take a left argument; all arguments are specified on the right. Indeed if VA is used to build a class library for use with other languages, it is preferable to have all arguments on the right.
- With other languages, some arguments are optional. With C#, the concept is called *overloading*, whereby a function can be defined a number of times with different arguments – with the missing ones being given default values. Additionally, with VB.NET a calling function can supply an argument by name.

VA removes all the restrictions on function arguments and complies with the condition that arguments are specified either by position or by name, not a mixture. VA functions can use either the classic valence or the .Net-compliant signatures. This is a major enhancement in APL but can be quite confusing, even frustrating. The confusion (or is it excitement?) gets worse because:

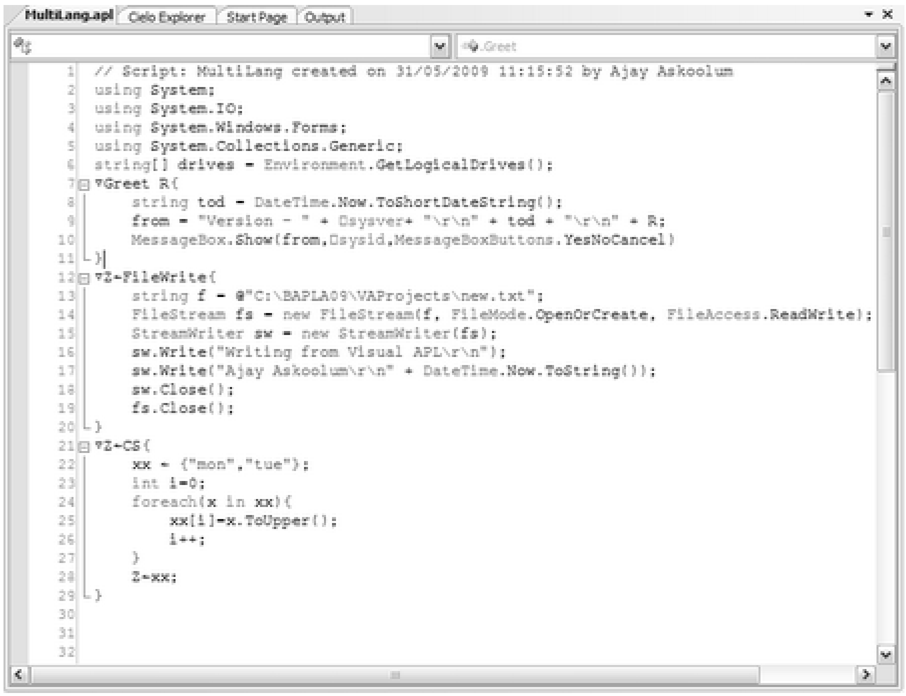
- It is possible to code a function using the classic valence and call it using the .Net signature!
- Primarily for the benefit of strongly-typed target languages, it is also possible to specify the types of arguments and return values of APL functions.

The following sections from the help files are vital reading:

Section	Topic
Visual APL Programming Guide	The AplFunction Attribute
Visual APL Tutorial	15 Defining Functions
	16 More about defining functions
	17 Typing Arguments to Functions
	18 Data Types and Collections

Multi-language coding

It is possible to mix APL and C# code within a class. In fact this is a critical advantage in using VA, namely, the whole of the .Net functionality is available. There are over 12 million C# developers – and this is growing – and but a handful of APL developers – and this is shrinking. Refer to Figure 16 for some examples.



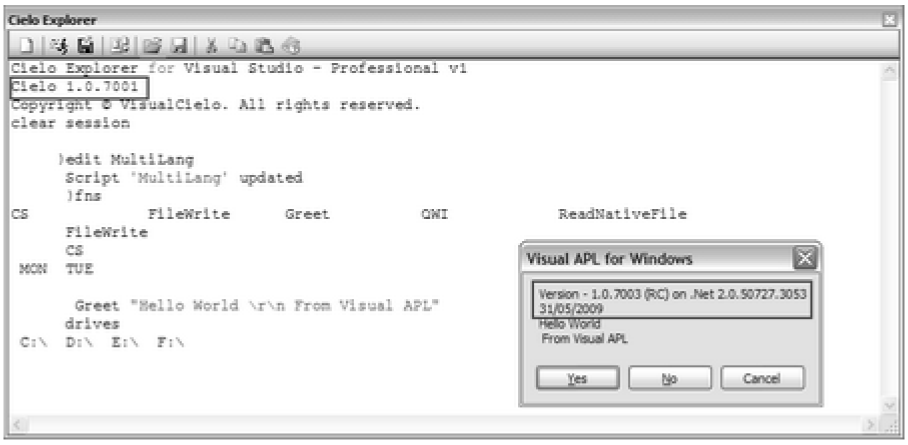
```
// Script: MultiLang created on 31/05/2009 11:15:52 by Ajay Askoolum
using System;
using System.IO;
using System.Windows.Forms;
using System.Collections.Generic;
string[] drives ← Environment.GetLogicalDrives();
∇Greet R{
    string tod ← DateTime.Now.ToShortDateString();
    from ← "Version - " + ⎕sysver+ "\r\n" + tod + "\r\n" + R;
    MessageBox.Show(from,⎕sysid,MessageBoxButtons.YesNoCancel)
}
∇Z←FileWrite{
    string f ← @"C:\BAPLA09\VAProjects\new.txt";
    FileStream fs ← new FileStream(f, FileMode.OpenOrCreate, FileAccess.ReadWrite);
    StreamWriter sw ← new StreamWriter(fs);
    sw.Write("Writing from Visual APL\r\n");
    sw.Write("Ajay Askoolum\r\n" + DateTime.Now.ToString());
    sw.Close();
    fs.Close();
}
∇Z←CS{
    xx ← {"mon","tue"};
    int i←0;
    foreach(x in xx){
        xx[i]←x.ToUpper();
        i++;
    }
    Z←xx;
}
```

Figure 16: Multi-language coding

Some observations on mixing code

Refer to Figure 17 for the actual CE session.

- Lines [2] – [5] show the `using` directives; in this case, only the ones used by the functions are included. However, there is no harm in including the typical class on libraries that a C# solution includes by default.
- Line [6] creates a global variable, which is a nested vector in APL.
- Lines [7] – [11] create a message and displays it; aside of the function header and the reference to APL system constants, the code in C#.
- Lines [12] – [20] illustrate how C# worked examples can be used with VA almost unchanged. On line [13], note the double backslash: backslash is an escape character in C#. On line [16], `\r\n` represents carriage return and linefeed. Unlike VA, which uses the delimiters interchangeably, C# uses double quotes to enclose data of type *string* and single quotes for data of type *char*. However, in order to avoid stray errors, follow the C# convention for passing data into mixed code.
- Lines [21] – [29] create a trivial example to illustrate how C# can manipulate and return what APL sees as a nested vector (and arrays with suitable modification). Note also that this APL function is using a control structure that belongs to C#. The control structure keywords in VA begin with colon as in APL+Win. (Did you notice the anomalies?)
- The version number shown in the message box is different from that shown in a new CE session. APLNext has confirmed that CE and VA projects use the same APL black box, so I would have expected the versions to match, as I am using the same version throughout.
- Although I am using VS2008 with the default framework set to 3.5, the message box suggests that VA is using Framework 2.0.



```
Cielo Explorer for Visual Studio - Professional v1
Cielo 1.0.7001
Copyright © VisualCielo. All rights reserved.
clear session

      )edit MultiLang
      Script 'MultiLang' updated
      )fns
CS              FileWrite      Greet          QWI            ReadNativeFile
      FileWrite
      CS
 MON  TUE

       Greet "Hello World \r\n From Visual APL"
      drives
 C:\  D:\  E:\  F:\
```

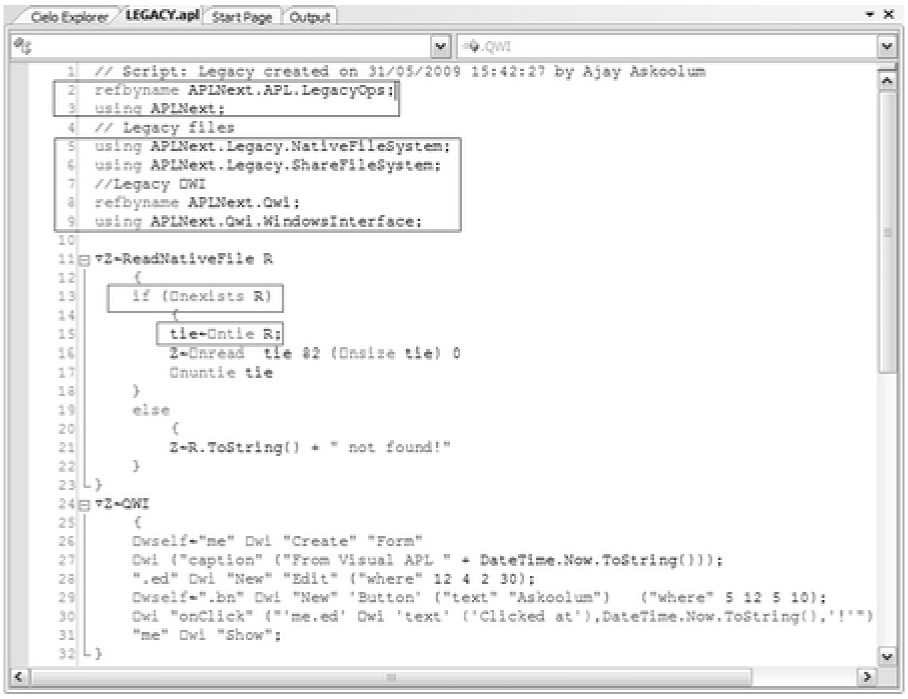
```
Version - 1.0.7003 (RC) on .Net 2.0.50727.3053
31/05/2009
Hello World
 From Visual APL
```

Figure 17: Mixed code running in Cielo Explorer

Legacy features

During the beta testing phase, APLNext made a concession, namely, added back support for native (`⎕n*`) and component (`⎕f*`) files, `⎕fmt` and `⎕wi` – at a price. These features are not available by default; express references to class libraries are required to enable them.

Figure 18 gives an insight into how legacy functionality can be deployed. The first function, `ReadNativeFile`, reads the file created by the method `FileWrite` (using C#) in 'Multi-language coding'.



```
// Script: Legacy created on 31/05/2009 15:42:27 by Ajay Askoolum
refbyname APLNext.APL.LegacyOps;
using APLNext;
// Legacy files
using APLNext.Legacy.NativeFileSystem;
using APLNext.Legacy.ShareFileSystem;
//Legacy ⎕WI
refbyname APLNext.Qwi;
using APLNext.Qwi.WindowsInterface;

∇Z←ReadNativeFile R
    {
    if (⎕nexists R)
        {
        tie←⎕ntie R;
        Z←⎕nread  tie 82 (⎕nsize tie) 0
        ⎕nuntie tie
    }
    else
        {
        Z←R.ToString() + " not found!"
    }
}
∇Z←QWI
    {
    ⎕wself←"me" ⎕wi "Create" "Form"
    ⎕wi ("caption" ("From Visual APL " + DateTime.Now.ToString()));
    ".ed" ⎕wi "New" "Edit" ("where" 12 4 2 30);
    ⎕wself←".bn" ⎕wi "New" 'Button' ("text" "Askoolum")   ("where" 5 12 5 10);
    ⎕wi "onClick" ("'me.ed' ⎕wi 'text' ('Clicked at'),DateTime.Now.ToString(),'!'")
    "me" ⎕wi "Show";
}
```

Figure 18: Legacy support

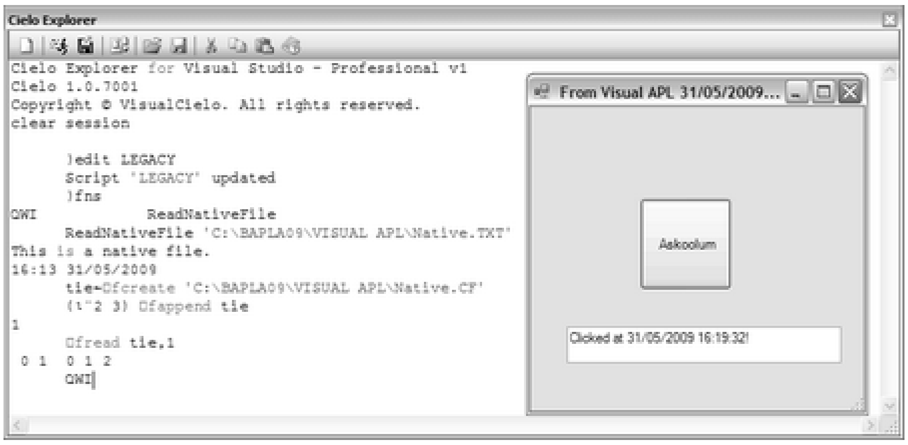
My personal view is that this was a retrograde step, a mistake – it shows how strong the legacy lobby can be – because:

- Much better corresponding functionality – with documentation – exists in the .Net platform (except perhaps for component files).
- There is no guarantee that existing code using these quad functions will work in the same way as these have been reverse engineered for the .Net platform.
- Component files hide data and create special bespoke requirements; this is bad news from the point of view of securing industry-wide peer endorsement.
- There is no automatic migration option for legacy APL applications.

- However, on the positive side:
- Some of these quad functions have been enhanced. For example, ⎕nexists is a new function that returns true if a file does not exist (long overdue although simply done in C#) and ⎕ntie has a new syntax that returns the next available tie number. The help files document all the changes and exceptions.
- The excesses of extended and colossal native and component file functions have not been implemented.
- The native and component files created by VA and APL+Win are not interchangeable across the two environments. It is time to embrace more modern data-tier handling using ADO.NET.

Might it have been more appropriate to court the legacy applications built with competing APLs than to provide legacy support?

Figure 19 shows the results of the code.



```
Cielo Explorer for Visual Studio - Professional v1
Cielo 1.0.7001
Copyright © VisualCielo. All rights reserved.
clear session

      )edit LEGACY
      Script 'LEGACY' updated
      )fns
QWI             ReadNativeFile
      ReadNativeFile 'C:\BAPLA09\VISUAL APL\Native.TXT'
This is a native file.
16:13 31/05/2009
      tie←⎕fcreate 'C:\BAPLA09\VISUAL APL\Native.CF'
      (⍳¨2 3) ⎕fappend tie
1
      ⎕fread tie,1
 0 1  0 1 2
      QWI
```

```
Clicked at 31/05/2009 16:19:32!
```

Figure 19: Working legacy features

Managed code support

Although VA supports APL+Win native and component file operations, the code that provides this support is managed code. The VA and APL+Win component files themselves are not compatible and the component file functions cannot read APL+Win component files. A freestanding utility is available to port the APL+Win component files to VA.

Note that there are some enhancements in the supporting functions; notably, the tie numbers of files can be determined automatically and there is a function for determining whether a file exists.

The VA ⎕wi function does not support the APL+Win COM interface, because .Net has its own (interop) method of interacting with legacy COM or ActiveX DLLs.

APLNext project templates

Among the several project templates that are available with this version, I'll consider just two, namely Windows Application and Class Library.

Windows applications would have a user interface. For a VA windows application, VA takes over the form design and runtime handling. For me, this decision is anomalous for the following reasons:

- On the one hand, the designers have settled for the standard VS2008 IDE with little adaptation for VA and on the other they have made the code generation for forms bespoke (or is it simply using VS2003?) – the actual process of putting together a form is driven via the IDE.
- One of the 'benefits' of VA handling the code behind forms is that statement terminators are omitted; for me this is of dubious value. A C# project puts the system-related and user-defined code for forms into separate files, e.g. `Form1.Designer.cs` and `Form1.cs`, respectively. VA puts all the code in the same file, e.g. `Form1.apl`. I think this makes the application design and maintenance processes much harder.

I concede that I have very likely missed the finer subtleties of integrating APL into VS2008 but I am inclined to believe APLNext have missed a couple of opportunities here.

First, there is no `APLNext Class` option in the rich list of options available from the `Project | Add Class…` menu item.

Second, I would have settled for the same form designer as C# – and lived with the default `cs` extension (VA uses `apl` as the extension) – and added an `APLNext` class with a reference to this class into the form's file. Why? Form handling essentially involves the handling of scalar data and I would not expect much call for VA's array facilities. This arrangement would have some significant advantages:

- It might entice C# developers more persuasively and externalise the support issues relating to form handling. The partitioning of an application into distinctive tiers has significant advantages in terms of the number of people who can work on it and the debugging process.

- It would permit form design and runtime handling using both native C# and APLNext code in the separate class.
- It would minimize the incidence of problems relating to data typing, bearing in mind that C# is strongly typed and VA is not.
- It would make the `APLNext` class available at all times within the same project; that means that all the debugging facilities are to hand. The alternative is to have an APLNext DLL for code handling from the form; while this is still a highly viable option, especially for core and debugged functionality, it would make the process of debugging the application that uses the DLL a rather disparate process.

APL+Win

APL2000 actively support APL+Win and provide ongoing development. In June 2009, they released version 9.1 of the product. At version 9, as you would expect, the maintenance cycle no longer includes the fixing of any significant flaws or bugs in the product. It consists primarily of an effort to maximize efficiency gains via better or clearer documentation, the provision of worked examples, and the fine-tuning of particular functionalities to make them faster. In other words, you would not expect radical new features in a product that has a large base of existing applications.

There are, however, some very interesting uses of existing functionality.

Table 3: APL+Win Data Type Support

VB Type Name	Scalar Type	Array Type
Empty	0	-
Null	1	-
Integer	2	8194
Long	3	8195
Single	4	8196

Table 3: APL+Win Data Type Support

VB Type Name	Scalar Type	Array Type
Double	5	8197
Currency	6	8198
Date	7	8199
String	8	8200
Object	9	8201
Error	10	8202
Boolean	11	8203
Variant	12	8204
DataObject	13	8205
Byte	17	8209

ActiveX interface

APL+Win has a robust ActiveX interface that permits its deployment both as a client and as a server. The client can be APL+Win itself or any other Component Object Model (COM) compliant software, including C#. APL+Win can be the server to APL+Win as a client.

Routinely, APL+Win copes with incoming and outgoing data types seamlessly; however, there are occasions when this does not quite work because some data types do not exist in APL. Many ActiveX objects use values that are typed; that is, a variable can hold a value, which has a special representation of the raw value. For example, one special type is currency. For such situations, there are means of translating the data explicitly.

Other examples include null, empty or missing values: the system object # can create such values.

APL+Win as COM Server and Client

APL+Win can act as both a COM client or as a server; in other words, it can work with itself in a COM configuration. For example:

```
      ⎕wself←'APLW' ⎕wi 'Create' 'APLW.WSEngine'
      ⎕wi 'XExec' '+/⍳10'
55
```

If the client is other than APL+Win, it will not be possible to pass APL expressions for evaluation because of the nature of the APL keyboard; however, there would be little point in using APL+Win as a COM server in immediate mode. The properties, methods, and events that are exposed are:

```
      (∊'x'=1↑¨x)/x←⎕wi 'properties'
xSysVariable xVariable xVisible
      (∊'X'=1↑¨x)/x←⎕wi 'methods'
XCall XExec XSetOrphanTimeout XSysCall XSysCommand
      (∊'X'=1↑¨x)/x←⎕wi 'events'
XNotify XSysNotify
```

ActiveX interface – using redirection

One feature of the APL+Win ActiveX interface is the ability to create objects using redirection.

Imagine that you have an existing session of Excel – perhaps one orphaned by a client that terminated abruptly – and you want to use that session as a COM server. How do you do it?

```
      'xl' ⎕wi 'Create' 'Excel.Sheet'
                                                   xl
      'xl' ⎕wi 'xApplication>xl'
```

Now `xl` is an instance of the oldest existing Excel session. Note:

- This technique requires error trapping as it will fail if there are no existing Excel sessions.
- It corresponds to the `GetObject` function that exists in Visual Basic.

ActiveX – events

APL+Win enables seamless event handling.

```
      'xl' ⎕wi 'onXSheetSelectionChange' '⍳10'
```

The event fires when another cell is selected; either an APL expression or an APL function may be specified as the event handler.

Two system variables are available to event handlers:

`⎕warg`

  contains the arguments passed by the event.

`⎕wres`

  contains the behaviour passed back to the server.

ActiveX – syntax

APL+Win uses a prefix of `?` to query the signature of the properties, methods, or events of ActiveX objects. For example:

```
      'xl' ⎕wi '?Range'
xRange property:
  Value@Object_Range ← ⎕WI 'xRange' Cell1 [Cell2]


      'xl' ⎕wi '?onXSheetSelectionChange'
onXSheetSelectionChange event:
  ⎕WEVENT ←→ 'XSheetSelectionChange'
  ⎕WARG ←→ Sh@Object Target@Object_Range
  ⎕WRES[2] ← Target@Object_Range
```

Note that in the latter example, the event passes an object to the client.

A prefix of `??` invokes the help file of the ActiveX object and displays the relevant topic; if this fails, the signature is returned.

ActiveX interface - passing objects as arguments

Usually, the `progid` of an ActiveX object has two levels, e.g. `Excel.Application` and the syntax for creating instances of such objects is straightforward. However, some properties expose child objects; for example:

```
      'xl' ⎕wi 'Range()' 'A1:F5'
88430676
```

For such properties, it is necessary to create an instance of the object returned; however, all that is available is an object pointer and not a progid. APL+Win can create an object from the object pointer. The following two techniques return the same result:

```
      'xl.rng' ⎕wi 'Create' ('xl' ⎕wi 'Range()' 'A1:F5')
xl.rng
      'xl' ⎕wi 'Range()>xl.rng' 'A1:F5'
```

APL+Win, as client, creates instances of objects passed by events in the same manner.

```
      'xl' ⎕wi 'onXSheetSelectionChange' 'MyFn'
```

The syntax query indicates that two objects are returned by the event; hence, the handler may need to create two objects.

```
    ∇ MyFn
[1] ('obj1' 'obj2') ⎕wi¨ (⊂⊂'Create'),¨⊂¨⎕warg
    ∇
```

Win32 API

A tradition that started in the halcyon days of the Disk Operating System (DOS), APL secured a unique competitive advantage by providing quad functions for accessing operating system resources. With operating systems becoming more complex, the APL strategy has simply failed as it is not possible to provide a quad function for everything. Microsoft provides an Application Programming Interface (API) that is much more comprehensive and is widely adopted by developers of other languages. APL can deploy the same techniques.

APL+Win can deploy Win32 API too and it does so in a unique manner: the bridge to the API calls is independent of the workspace. Therefore, APL2000 has been able to ship APL+Win with a large number of popular API calls with the interpreter. Workspace independence also implies that means that any newly definition becomes universally available to all workspaces.

Defining new API calls

Developers are able to define API calls if they are found missing in the predefined set supplied by APL2000. One of the requisites for defining any particular API call is the ability to query whether that definition exists already. The following expression, if true, indicates whether a definition is available:

```
0≠⍴⎕wcall 'W_Ini' '[Call]MakesureDirectoryPathExists'
```

This API is documented as follows:

```
Declare Function MakeSureDirectoryPathExists Lib "imagehlp.dll"

             (ByVal lpPath As String) As Long
```

It can be defined conditionally as follows:

```
   ∇ API

[1] ⍝ Define MakesureDirectoryPathExists conditionally

[2] :if 0=⍴⎕wcall 'W_Ini' '[Call]MakesureDirectoryPathExists'

[3]     ⎕wcall 'W_Ini' '[Call]MakesureDirectoryPathExists=L(*C
lpPath)

             ALIAS MakeSureDirectoryPathExists LIB imagehlp.dll'

[4] :endif

   ∇
```

This API call is capable of creating a hierarchical directory in a single pass: for example,

```
⎕wcall 'MakeSureDirectoryPathExists'
'c:\ajay\askoolum\Finance\Qtr1\'
```

API calls are efficient.

API callbacks

Some API calls involve callback functions. For example,

```
Declare Function EnumWindows Lib "user32.dll"
          (ByVal lpEnumFunc As Long, ByVal lParam As Long) As Long
```

The parameters are:

`lpEnumFunc`
: Points to an application-defined callback function.

`lParam`
: Specifies a 32-bit, application-defined value to be passed to the callback function.

Which applications are running?

This API can return a list of application that are running:

```
  EnumWindows
C:\PROGRA~1\AVG\AVG8\avgtray.exe
C:\Program Files\API-Guide\API-Guide.exe
C:\Program Files\APLWIN8\APLW.EXE
C:\Program Files\APLWIN8\aplw.exe
C:\Program Files\Creative\SBAudigy2\DVDAudio\CtdVDDet.EXE
C:\Program Files\Creative\SBAudigy2\Surround Mixer\CTSysVol.exe
C:\Program Files\Dell\Media Experience\PCMService.exe
C:\Program Files\IBM\SQLLIB\BIN\db2systray.exe
C:\Program Files\Microsoft Office\Office12\EXCEL.EXE
C:\Program Files\Microsoft Office\Office12\GrooveMonitor.exe
C:\Program Files\Microsoft Office\Office12\WINWORD.EXE
C:\Program Files\Microsoft SQL Server\80\Tools\Binn\sqlmangr.exe
C:\Program Files\NetMeeting\conf.exe
C:\Program Files\ScanSoft\PaperPort\pptd40nt.exe
C:\WINDOWS\BCMSMMSG.exe
C:\WINDOWS\Explorer.EXE
C:\WINDOWS\System32\Ati2evxx.exe
C:\WINDOWS\System32\DSentry.exe
C:\WINDOWS\system32\CTHELPER.EXE
C:\WINDOWS\system32\ctfmon.exe
C:\WINDOWS\system32\dla\tfswctrl.exe
```

```
C:\WINDOWS\system32\rundll32.exe
C:\WINDOWS\system32\wscntfy.exe
```

The functionality is defined as follows:

```
   ∇ Z←EnumWindows;ptr;hdc
[1] Z←''
[2] ⍝ the filter appends the name to Z
[3] ptr←⎕wcall 'W_CreateFilter'
                   ('EnumWindows' 'Z←Z,⊂EnumWindowsCallback2')
[4] →(ptr=0)/0                      ⍝ unable to create the filter
[5] 0 0ρ⎕wcall 'EnumWindows' ptr 0  ⍝ make the call
[6] 0 0ρ⎕wcall 'W_DestroyFilter' ptr     ⍝ free the ptr
[7] Z←((Z⍳Z)=⍳ρZ)/Z                      ⍝ remove duplicates
[8] Z←⊃Z                                 ⍝ convert to a matrix
[9] Z←Z[⎕AV⍋Z;]                          ⍝ sort alphabetically
   ∇
```

An alternative callback function that might be used to return, say, Windows captions etc. – is defined thus:

```
   ∇ Z←EnumWindowsCallback2;procid;proc_hwnd
[1] procid←2⊃⎕wcall 'GetWindowThreadProcessId' (⍬ρ⎕warg) ⍬
[2] proc_hwnd←⎕wcall 'OpenProcess'
          'PROCESS_QUERY_INFORMATION PROCESS_VM_READ' 0 (↑procid)
[3] Z←↑↑/⎕wcall 'GetModuleFileNameEx' proc_hwnd 0 (256ρ⎕tcnul) 256
   ∇
```

The API definitions are stored in an INI file, typically `aplw.ini`; that file can also store predefined constants such as the ones used in `EnumWindowsCallback2[3]`.

APL+Win and .Net

A frequent request in the support forum is for a ⎕NET functionality for harnessing .Net classes. I have no idea what APL2000 plans to do in the future.

My own view is that the deployment of such a function – that is, mixing managed and unmanaged code – would make applications harder to maintain.

ActiveX

The alternative route is to build Interop ActiveX components in .Net and then use them with APL clients. From a personal point of view, this approach has merit for the following reasons:

- ActiveX promotes code re-use.
- Although there are murmurs about the continued use of ActiveX technology, it remains viable for the near future because of it prevalence. The Windows operating system, including Vista, uses this technology extensively.
- Separating ActiveX components, that is, servers, from clients (APL) makes it easier to maintain both, using developers with corresponding skill sets.

NetAccess

APL+Win now offer NetAccess, a user interface that simplifies the task of building the elements of a .Net ActiveX component, from lescasse.com. [3]

Current APL2000 subscribers can acquire NetAccess free.

APL and .Net – options

As far as I can see, there are three (possibly four) options for bringing APL and .Net together.

- APL+Win – Use Visual C# 2005/2008 Express (free) and APL+Win. The possible arrangements are either to build the application in C# and use APL code as a black box or to build DLLs using C# to make .Net facilities such as ADO.NET available to APL+Win.
- Visual APL – requires Visual Studio 2008; I believe the version for Visual Studio 2010 is in preparation.

VA has significant advantages.

- It has greater wider appeal to other .Net developing communities because it shares the same IDE and especially to C# developers because it is not only C#-like but can also integrate C# code. In case you are inclined to dismiss this, imagine what an uptake of APL by just 1% from the growing 12 million C# developers would mean for APL.

- It makes it possible to adapt worked examples from the Internet and other printed material into APL projects, almost without change. This APL does not lock its developers into a closet.
- It is a modern and up-to-date product and part of the flagship range of development tools; it will benefit directly from enhancements that Microsoft makes to Visual Studio in the future.

Conclusions

VA is a completely new APL for contemporary software development; it is hosted by the flagship IDE of today. I have participated in the beta and Release Candidate cycles of the development of VA. It has been exciting to see the product develop to the current release. For ongoing success, the vendor must provide a hefty manual with worked examples for the types of application that can use VA; this will not only provide a means of exposing what VA can do (i.e. training) but also provide a template for software developers.

Would I use VA to build applications?

The answer is emphatically in the affirmative. VA takes APL to .Net, in my opinion, very successfully. This has significant advantages, especially the opportunity to adopt worked examples from the .Net world. However, the designers seem a little reticent when it comes to GUI-based applications: if the designers are going to adopt the C# form design approach, that is better done before the product has a legacy of applications.

Would I migrate applications to VA?

Yes! Legacy APL has a lot of clutter, which makes maintenance very expensive:

- This includes the myriad utility functions developed over 40 years. Most of them can be easily replaced either by improved nested APL functionality or by access to platform resources such as API calls.
- It is dependent on highly specialised infrastructure that includes component files and a bespoke GUI designer and code editor (cannot seriously call it an IDE). Such dependencies make APL inaccessible and deny the acquired experience with standard platform tools and resources.

VA overcomes the stigma: it offers a modern APL that is free of clutter. More tellingly, it does *not* offer an automated migration path – which I applaud. The .Net facilities are very comprehensive.

Opportunity to re-engineer applications

Therefore any migration has to be a manual process that may seem onerous and expensive but it also heralds a new opportunity, namely, to re-engineer applications that, in many cases, are decades old. My own experience is that some 70% of a typical APL+Win post-V3.5 application can be migrated very quickly but this is a highly subjective assessment and will depend on the nature of the code. The remaining 30% usually raises fundamental issues relating to scope, which can be tricky.

I would recommend the re-engineering process take the opportunity to separate the application into tiers, at least, into three tiers, namely, presentation, business and data. ADO.NET can be deployed easily from VA.

Is there anything that I do not like about VA?

This is a highly subjective criterion. Being accustomed to C#, I notice the differences with VA. An example is that with C#, scope delimiters, i.e. braces, are always on new lines with the possible exception of properties; VA puts only the closing brace on a new line. This is highly perceptible when switching between the two. I prefer the C# arrangement. The VA font is less pleasing than the standard VS2008 fonts; the VA font is required for IDE pop-ups, such as code completion, to display correctly. Can VA switch this on or off dynamically depending on whether VS2008 is opening an VA application or not?

Although VA has a forum site and online documentation, the documentation is not adequate for a new product that calls on knowledge acquired from legacy APL but that works in a completely different way. The webcasts during the development cycle proved an invaluable supplement and still have a role to fulfil; unfortunately they seem to have stopped. Moreover, the vendor has to address the fact that there is little or no information on the deployment of VA applications; this was not a problem before the commercial release but it has some urgency now.

.Net competition

I think the critical difference is that VA is .Net: it works in tune with .Net as

Figure 20: The revolution starts now?

opposed to tuning .Net to work like APL; the latter is an expensive ongoing endeavour always playing 'catch up'.

Therefore, VA can concentrate on enhancing the language whereas the competition also needs to bring new developments into the workspace.

I believe the VA approach holds a better promise for the future of APL: an APL without workspaces is a valiant start.

Notes and references

21. A *solution* is Microsoft's term for a collection of projects. A project contains a namespace definition and the classes associated with that namespace. Each project can produce a .Net assembly.
22. http://forum.apl2000.com/viewforum.php?f=4 &sid=ad9188db7f262590715157d0c34ad0be
23. "Interface APL+Win and .Net (C#)", Eric Lescasse, http://www.lescasse.com/InterfaceAPLandCSharpA4.pdf

Further reading

24. Visual APL electronic help files and public newsgroups
25. "Building C# COM DLLs for APL", Ajay Askoolum, *Quote Quad* **34**, 4
26. *System Building with APL+Win*, Ajay Askoolum, John Wiley & Sons Ltd, 2006, ISBN 10-0-470-03020-8
27. *APL An Interactive Approach*, 3rd edition, Leonard Gilman & Allen J Rose, John Wiley & Sons, 1984, ISBN-10-0-471-09304-1
28. *Les APL étendus*, Bernard Legrand, MASSON, 1994, ISBN 10-2-225-84579-4
29. *Professional C# 2005*, Nagel, Evjen, Glynn, Skinner, Watson & Jones, WROX, 2006, ISBN 10-0-7645-7534-1

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 1 May to 30 April.

Name	________________
Address	________________
	________________
Postcode/Zip and country	________________
Telephone number	________________
Email address	________________

UK private membership	£20	__
Overseas private membership	£22	__
+ airmail supplement outside Europe	£4	__
UK corporate membership	£100	__
Overseas corporate membership	£110	__
Sustaining membership	£500	__
Non-voting UK member (student/OAP/unemployed)	£10	__

Payment should be enclosed with your membership as a sterling cheque payable to *British APL Association*, or by quoting your Visa, Mastercard or American Express number:

I authorise you to debit my Visa/American Express/Mastercard (mark which)

account number ________________ expiry date ____ /______

for the membership category indicated above

_____ annually, at the prevailing rate, until further notice

_____ for one year's subscription only

Signature ________________ Date ________________

Send this completed form to:
BAA, c/o Nicholas Small, 12 Cambridge Road, Waterbeach, Cambridge CB25 9NJ