VECTOR

100+ pages of APL News & Views

Including ...

The Journal of the British APL Association

A Specialist Group of the British Computer Society

ISSN 0955-1433

Vol.12 No.4 April 1996

Contributions

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

Please supply as much material as possible in machine-readable form, ideally as a simple ASCII text file on an IBM PC compatible diskette (any format). APL code can be accepted as camera-ready copy, in workspaces from I-APL, APL*PLUS, IBM APL2/PC or Dyalog APL/W, or in documents from Windows Write (use the Vector TrueType font, available free from Vector Production), and Winword-2.

Except where indicated, items in VECTOR may be freely reprinted with appropriate acknowledgement. Please inform the Editor of your intention to re-use material from VECTOR.

Membership Rates 1995-96

Category	Fee	Vectors	Passes
UK Private	£12	1	1
Overseas Private	£14	1	1
(Supplement for Airmail, not needed for Europe)	£4		
UK Corporate Membership	£100	10	5
Overseas Corporate	£135	10	
Sustaining	£430	10	5
Non-voting Member (Student, OAP, unemployed)	£6	1	1

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

Corporate membership is offered to organisations where APL is in professional use. Corporate members receive 10 copies of VECTOR, and are offered group attendance at association meetings. A contact person must be identified for all communications.

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

Advertising

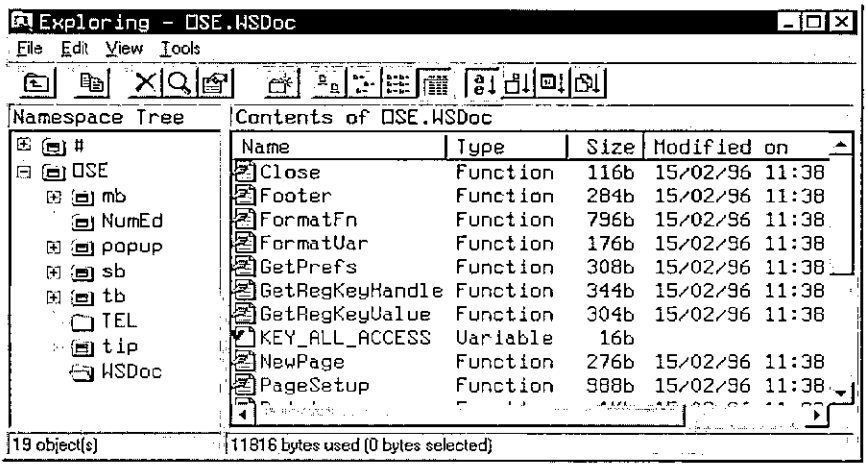
Advertisements in VECTOR should be submitted in typeset camera-ready format (A4 or A5) with a 20mm blank border after reduction. Illustrations should be photographs (b/w or colour prints) or line drawings. Rates (excl VAT) are £250 per full page, £125 for half-page or less (there is a £75 surcharge per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

Deadlines for bookings and copy are given under the Quick Reference Diary. Advertisements should be booked with, and sent to: Gill Smith, Brook House, Gilling East, YORK YO6 4JJ. Tel: 01439-788385 CompuServe: 100331,644

Contents

dyalog APL

The Definitive APL for Windows™



Dyalog APL Namespaces let you ...

- **Organise** your workspace into self-contained sub-systems
- **Encapsulate** functions and variables *within* GUI objects
- **Isolate** utilities from your application code

Plus The new Version 8 **Explorer** lets you browse namespaces, drag-drop objects from one to another, and a whole lot more.

That's why Dyalog APL/W remains the **professional** choice. For further information, contact Dyadic or your local distributor today.

MICROSOFT® WINDOWS™ COMPATIBLE

Dyadic Systems Limited, Riverside View, Basing Road, Old Basing, Basingstoke, Hants. RG24 7AL, United Kingdom.
Tel:+44 1256 811125 Fax:+44 1256 811130 Email: sales@dyadic.com

EDITORIAL

by Anthony Camacho

I am handing over the Editor's chair to Duncan Pearson from Volume 13 onwards. I shall continue helping him as a member of the Vector Working Group and, I hope, subsequent editors. For the record, the roll of Vector's editors is: Robert Bittlestone, David Preedy, Adrian Smith, Jonathan Barman, me and now Duncan Pearson.

The working group is the thing that keeps Vector going. That, far more than the editor, deserves your thanks for twelve years of the best in APL. How it came about is hard to explain. It wasn't planned, with the social dynamics carefully worked out. It was started by David Preedy and still has some of the original members. Current membership is: Adrian and Gill Smith, David Ziemann, Jonathan Barman, Richard and Adam Weber, Duncan Pearson, John Searle, Ray Cannon & Jon Sandles. Editors for education Vector have not always attended the working group meetings. Nevertheless, Norman Thomson, Alan Sykes, Alan Mayer and Ian Clark also deserve your thanks.

And while I am thanking people I also thank the authors, especially those who I, or others, bullied into writing.

I just looked at the Vectors and editorials I've been responsible for. There are quite a lot of editorials about conferences and how I think they should be run — so many that I have been accused of riding a hobby-horse. One good thing to note was the reduction in price (in Vol.10 No.3) which should have put Vector into everyone's reach.

The two best editorials (my choice) are in Vol.11 (Nos 1 & 2).

Now I shall have (as they say of Tory ministers that get caught) more time to spend with my family. Alison gave birth to Gemma on 24th February and we expect a second grandchild later this year.

High Energy APL Developer

Saladin

Saladin is a successful and rapidly growing company with a niche market in the supply of information services and software to the energy trading business. Our business is truly global, with 200 customer organisations supported from our offices in the UK, USA and Singapore.

Saladin's most widely used product, PAWS, is set for a new period of intensive development following its successful conversion to MS Windows using the Dyalog APL/W environment, and we are now looking for a talented and enthusiastic individual to complete the product development team at our headquarters in Walton-on-Thames, Surrey.

The successful candidate will have extremely good APL skills and, ideally, experience of developing in a Windows environment. Knowledge of other programming languages (particularly C/C++) would also be welcome. Applicants should have a track record in APL development of at least two years, but candidates with a substantially greater level of experience should not hesitate to apply.

With its excellent road and rail links (25 minutes from Waterloo), Walton-on-Thames is in comfortable commuting distance of both London and the M4 corridor.

Salary will be dependent on experience and ability, but is likely to be highly competitive. Benefits include private medical cover and life assurance.

Please call Hugh Hagan on 01932 243233 to discuss the position, or send full career details to Hugh at *Saladin Ltd, Walton Court, Station Avenue, Walton-on-Thames, Surrey, KT12 1NT.*

Saladin – experts in energy information solutions

Quick Reference Diary 1996

Date	Venue	Event
17th May 1996	London Royal Statistical Soc.	**AGM and Vendor Forum** Provisional Programme ...
		1.30pm Annual General Meeting 2.00pm Bloomsbury Software 2.40pm Causeway Graphical Systems 3.20pm --- Tea break --- 3.40pm MicroAPL 4.20pm Dyadic Systems 5.00pm Close
24-25 June 1996	Toronto Canada	**J User Conference** For more information, contact Anne Faust email: amfaust@aol.com
July 28 - August 2 1996	Lancaster University UK	**APL96** - Designing the Future See *Preliminary Programme* and *Abstracts* on pages 6-9

The Annual General Meeting and Vendor Forum will be held at **The Royal Statistical Society** on Friday 17 May 1.30 p.m. to 5.0 p.m. Details will be sent to all members as an independent mailing.

Dates for Future Issues of VECTOR

	Vol.13 No.1	Vol.13 No.2	Vol.13 No.3
Copy date	24th May 96	6th Sept 96	6th Dec 96
Ad booking	31st May 96	13th Sept 96	13th Dec 96
Ad Copy	7th June 96	20th Sept 96	20th Dec 96
Distribution	July (at APL96)	October 96	January 97

APL96

Programme Highlights and Abstracts of Accepted Papers

report on Programme Committee Meeting by Adrian Smith

APL96 Programme Committee Meeting at Lancaster Saturday 16th March 1996 at the Lancaster House Hotel

Present were:

Adrian Smith (Causeway)
Philip Benkard (IBM, retd)
John Scholes (Dyadic Systems)
Dr Alan Sykes (University of Swansea)
Morten Kromberg
Gitte Christensen (both of Insight Systems)

Note that this followed a preliminary meeting in New York between Adrian Smith, Mike Kent, Philip Benkard and Lynne Shaw on Saturday 2nd March, where the general shape and content of the conference were also discussed.

Programme Outline

The meeting began by discussing the basic layout of the programme grid, and agreed on a standard format for each day of a morning plenary, followed by a Vendor forum, followed by a short session when each workshop leader would spend no more than 15 minutes introducing the material to be covered that afternoon. This takes the morning through to coffee break, after which we allocated two parallel tracks for submitted papers. Where possible, these papers will be programmed to support the material in the afternoon workshops.

Lunch was intentionally left clear, as several of us had received feedback from San Antonio that delegates had missed out on valuable 'networking' time because of the complete loss of lunchtime to vendor forums.

In the afternoon we worked with four parallel tracks: two of these were allocated to 'hands-on' workshops, one to a more traditional 'classroom' tutorial, and one to submitted papers and 'APL Success Stories' run sequentially. Not all the afternoon paper slots are currently filled with accepted material, so we would encourage potential authors to contact us with any 'late-breaking' APL stories.

The afternoon programme is scheduled to run until 7.00 (with a half-hour break for tea) with the option for enthusiasts to resume in the hands-on labs later in the evening. If workshops are heavily over-subscribed, we will need to look at the option of hiring more lab space and repeating sessions as required.

In general, the programme committee did not react favourably to the suggestion of 'panels' or 'debates', although the possibility of some 'panel-led' workshops (for instance on portability issues) was left open if suitable topics are suggested.

Strategy for Proceedings

It was agreed to provide delegates with a ring-binder containing refereed papers in the normal proceedings format, and workshop notes as they became available during the conference. The final proceedings will appear as the Autumn or Winter QuoteQuad, and will include additional material arising during the conference.

Programme Highlights

APL96 is planned to be the best training opportunity this year for anyone involved in designing, migrating or implementing the APL systems of the future. We have major 'hands on' workshops on GUI programming in Dyalog and J3 from Dyadic, Strand and Causeway. We have Timo Laurmaa running a full afternoon session on TCP/IP programming (APL over the Internet); Morten Kromberg will be running tutorials on ODBC access and setting up APL as an ODBC server; Norman Thomson and Philip Benkard are sharing a tutorial stream on nested-array programming.

Heinz Roggenkemper of SAP AG will explain why one of the world's largest software companies has chosen APL for graphical configuration tools, and Jack Rudd will show us how APL2 has been used to achieve unheard-of accuracy in civilian GPS systems. With submitted papers on everything from whale songs to economic modelling in Russia to the use of J as a teaching language - we plan to have something for everyone at APL96.

Abstracts of Accepted Papers

At the time of writing, we still have a number of papers under review. This list probably represents just over two-thirds of the papers which will be presented at APL96. The sequence numbers have been allocated in no particular order.

[002] Stephen M. Mansour
How to Write an APL Utility Function (Workshop)

In today's business climate, re-usable code is essential. But many programmers often don't use existing utility functions because they find them difficult to use or not general enough. Also, they may not know that such functions exist. Instead, programmers often clone lines of code from other functions. This results in sloppy, undocumented code which is full of errors. In order to avoid this, the author of a utility function must make an extra effort to ensure that his function is designed properly.

APL is easy to learn because its primitives behave consistently, work on arrays as well as scalars, can handle edge conditions, often use default values, and are totally encapsulated from the user. We can learn from this by designing utility functions in the same way, allowing them to become an extension of APL and its set of primitives. This workshop will show some design techniques and examples. Attendees will be encouraged to bring in their own examples.

[003] Manuel Alfonseca
Representation of Fractals by means of L-Systems

Fractals can be represented by means of L-systems (Development Grammars), together with a graphic interpretation. Two families of graphic interpretations have been used: turtle graphics and vector graphics. This paper describes an APL2/PC system able to draw fractals represented by L-systems, with both graphic interpretations. A theorem is proved on the equivalence conditions for both interpretations. Another point shown is the fact that supposed deficiencies in L-systems that have prompted proposals of extensions are really deficiencies in the graphic translation scheme.

[004] S.M.Obraztsov et al
Joint Deterministic/Adaptive Method for Economic Forecasting

Economic forecasting is of great importance when some economic mechanism is changed rapidly as in Russia nowadays. Any economic system is a deterministic/stochastic entity of great complexity. Because of this, informative models which offer the interplay of the most significant factors are inadequate for satisfactory long-term forecasting.

The paper describes a forecasting procedure based upon the joint use of formalized method (numerical simulation) and adaptive method (simulation with neural network) when the model structure is formed by incoming information. Combination of forecasts selected by experts allows to make the most likely forecast from the "fan" of probable tracks. APL-implementation of this procedure has been used for the forecasting of municipal expenditures.

[006] Alan Graham
0: a Simple Modern Array Programming Language System

0 is a programming compiler/interpreter and environment for Microsoft Windows 95 and NT. The session manager is written in Microsoft Visual Basic. Microsoft Word or WordPad can be used as a program editor. The 0 compiler is written in 0 and the runtime interpreter is J.

0 is a descendent of APL, APL2, and APL0. 0 preserves the qualities that made original APL great: fast array-based computation, compact symbolic mathematical notation, simple expression-oriented functional language, scalars (character and numeric) are abstract (internal details are hidden), computer management (declarations, space management, compile and link,

etc.) is automatic.

0 eliminates deprecated and redundant features. Rank and Depth are merged: a vector of vectors is identical to rows of matrix. Many modes are eliminated: no selectable index origin, print precision, comparison tolerance, or print width. Shared variables and auxiliary processors are removed: files are arrays, asynchronous processes are controlled by function calls.

0 extends the simplified APL2. New classes of scalars: expression, string, fault, and nil. New operators: apply, axis, compose, power. New functions: count, convert. New library functions: if, else, do, while, until, for take expressions as arguments and provide, using C-like syntax, control-flow functions. An APL/ASCII one-to-one mapping is built in for transfer and for devices that do not support alternate character sets, for example _i is a verbose form of iota.

[007] John E. Howland
Using J as an Expository Language in the Teaching of Computer Science to Liberal Arts Students

APL and J are seldom, if ever, used in the teaching of college or university courses. As a result, students rarely experience the benefits of learning and using these languages which are well known to expert practioners of APL and J. One outcome of this alarming fact is that APL and J are destined to be nothing more than, perhaps, obscure languages used only by a small number of experts in a few selected fields. Since few new people are being trained, uses of APL and J may cease completely because companies cannot afford the risk of systems based on a technology which has no skilled labour force.

Recently, the author has developed a new laboratory-based computer science course for liberal arts students in which students are introduced to 13 core computer science topics. Programming language is used in an expository fashion to describe each topic by building simple working models of each topic. These models are then used as the basis of laboratory experiments in a co-requisite laboratory course. Students are not taught programming in this course, but rather, are taught just enough of the syntax and semantics of the language to be able to read and understand the exposition and models. Initially, Scheme was used in the lecture notes and laboratory materials developed for this course. Recently, however, an experiment is under way to replace the use of Scheme in this course by J. The development of this course and laboratory was funded by the Meadows Foundation and NSF grant DUE 9452050.

[010] Linda Alvord
The Derivative is for Dancing

This paper uses J to illustrate the meaning of the derived function or a derivative which is a fundamental concept in the study of calculus. To make the idea more understandable we can demonstrate the notion using visual representations. First fill polygons and consider them as graphic objects. Then present them at successive intervals along the curve. Next, rotate each polygon by an angle obtained from the derivative. This will cause the object to appear to turn in as it appears to move along the curve. Using animation techniques of drawing and erasing the figure, it will appear to move appropriately along the curve. When the derivative is incorrect, the figure will appear to spin aimlessly along the curve. Hopefully this aspect of the "slope of the curve" will motivate and enliven the study of the derivative. In addition, it may provide an interesting twist to a study of animation.

[012] Johann Mitlöhner
Classifier Systems and Economic Modeling (possible workshop)

Human economic decisions are characterized by a number of factors which make them difficult to model with standard mathematical tools. Decisions can be more easily described by a set of rules, and some of them may be 'rules of thumb'. Economic behaviour is adaptive, in that people are able to adjust to a changing environment. It is argued in this paper that the classifier system framework is a suitable means of modelling human economic decisions. A case of a simple economic decision of finding an optimal price is discussed, which is later made more complex by introducing an input variable that affects the optimal price. It is shown that classifier systems can

be used in both tasks, and their performance is compared to human decisions in the same set of circumstances.

[013] Jack Rudd
Real-time APL Prototype of a Wide Area Differential GPS System

The power of APL2 was applied to design and prototype a wide area differential GPS system in a matter of months. This system is designed to have all the capabilities of the GPS control segment, but with dramatically more accurate geolocation estimates for civilian users. The prototype was implemented on a single workstation and demonstrated in real time with live GPS satellite signals.

An APL2 simulation of the main elements of the system was created in two labour months. This included precise simulation of satellite orbits, sensor measurements and the random walk of atomic clocks, as well as measurement processing and state estimation. A full APL simulation, including user algorithms, was completed in two more labour months. Then the simulation was evolved to process recorded GPS satellite measurements instead of simulated measurements. Particularly complex processing of the actual measurements was found necessary. This evolution was completed in less than three months.

An effective graphical demonstration using AP207 was created on a laptop computer for marketing the system to potential customers in the international market. Real-time processing of measurements simultaneously gathered from five GPS receivers was achieved, demonstrating user location accuracy of a very few metres.

[014] Manual Alfonseca
User Interfaces with Object-Oriented Programming in APL2

The power of general arrays is used to provide APL2 with object-oriented capabilities, which are used to generate user interface object classes such as windows, menus, dialog boxes and messages, among others, all of which can be created as persistent objects. This makes very straightforward the development of user interfaces for real applications.

[018] Robert Bernecky
APEX: The Peak of APL Parallel Performance

APEX, the APL Parallel Executor, is a high-performance parallel compiler for an extended subset of ISO Standard APL. APL programs compiled with APEX run up to 1300 times faster than interpreted APL, and are often competitive with FORTRAN and C. APEX also automatically parallelizes code for execution on multi-processor platforms including the CRAY C90 and the Silicon Graphics SGI Power Challenge.

This tutorial will be given in two 1 1/2 hour segments. The first half be of general interest. The second half will be more technical, and explore APEX internals. If all goes well, attendees will receive a floppy containing compiled versions of popular APL utilities. If time permits, we will compile a few simple applications.

Who should attend?
- Managers who are responsible for large APL projects
- APL programmers who are interested in high-performance APL
- Power APL users who need parallel computing capability
- Interpreter and compiler designers interested in compiled APL

What will I learn?

Part 1:
- What is APEX?
- How does APEX perform compared to interpreted APL?
- How does APEX perform compared to FORTRAN and C?
- How much parallel speedup does APEX get?

- Why does APEX-generated code run so fast?
- What can APEX do for me?
- What can't APEX do for me?
- How do I link APEX code with C or FORTRAN?
- How do I build DLLs with APEX?
- What platforms does APEX run on?
- What is the future of APEX?

Part 2:
- How does APEX work?
- Why is holistic design required?
- Where does all that speed come from?
- The role of SSA and SISAL in APEX
- APL design issues — semiglobals, functional control structures, value error
- The role of synergy and loop fusion

[019] Phil Chastney
Multi-dimensional Databases at Eurostat

At Eurostat we are developing a Multi-dimensional Database using APL for the front-end with an SQL interface to a relational DB at the backend. There will be a presentation of the concept of "dimension" and the underlying ideas and techniques of MDDs.

[020] Gérard Langlet
The Least-Action Principle (LAP) in APL

The least-action principle, one of the fundaments of Physics, has never been given a definition for computer science. APL is the best notation to try to reformulate it in modern terms.

[021] Per Gerlov
Song of the Whale (Parallel Processing using APL2)

At the Technical University of Denmark APL2 has often been used in a credit course. The objective of the course exercise has been to analyse the vocalisations of Sperm Whales. In January 1996 the course was run again, this time using the IBM SP/2 supercomputer to solve the problem. An additional objective in this course was to utilize the possibilities of this powerful parallel computer.

During the three weeks of the course the students learned the APL language, solved a nontrivial problem, used a very powerful computer for the solution and wrote a detailed report.

News from Sustaining Members

Compiled by Gill Smith

APL2000 Inc (Bloomsbury Software Ltd)

We are busily working on release 2.0 of APL+Win and expect to release it at the end of March 1996. This new release will allow your APL+Win applications to be used under Windows 3.1, Windows 95 or Windows NT so as a developer, you only need worry about one set of code for your application. We are adding new controls that will allow your applications to have the new "look and feel" that users have come to expect, and give it to them in any Windows environment. This means less complexity, reduced documentation, simpler training and easier support.

Among the controls we are adding is a single selector control that will enable you to easily create Tabbed Property Sheets, Application "Wizards", and button driven input forms. You will like the "ease of use" of this particular feature. In addition we are adding ListView, TreeView, Progress bar, TrackBar, and more to enhance your applications and speed development. You will also enjoy speedups and enhancements to the interpreter (such as Take and Drop along Axis). Applications written using 32 bit DLLs will work fine in all environments and 16 bit DLLs can be accessed in Win95 or 3.1. A second release providing OCX support is expected to be demonstrated at the international APL96 conference in July. With this group of APL enhancements your development tools will be expanded to include virtually all of the widely available programming and utility packages, and allow you to smoothly integrate APL with other tools and environments.

We said our mission goes beyond keeping the interpreter up-to-date. With this in mind, we're pleased to announce the creation of some workspaces called APL+Widgets. These are workspaces of tools or application fragments that can help you add features to your application, or perhaps see new approaches to using APL+Win. The collection currently includes a calendar control (which looks remarkably like the calendar in Quicken for Windows) and a video control for showing AVI files. Over time, we'll add additional workspaces to showcase the features of the language.

We believe APL+Unix is important to your long-term Client-Server strategy. We will be improving our UNIX product to make it a better tool for Server applications. APL+Unix is being upgraded to incorporate control structures and

to consolidate all of the bug fixes and changes given out on a per request basis for the product. You will find the addition of control structures to be especially useful in developing Server applications. In addition to this major change, we are adding Take and Drop along Axis to bring us closer in line to the APL2 language. This release is expected to be ready for beta distribution in the second quarter of 1996.

To ease the flow of code among all three platforms, we are adding control structures to APL+DOS in the third quarter. This will allow computational routines to be easily moved without having to be rewritten.

Any license purchased in 1996 will be eligible for free (with the exception of shipping charges) upgrades to all releases of that product during the calendar year. This will include all improvements and fixes to that product. That means that if you purchased APL+Win in January, you will receive all of the APL+Win improvements as they are available. To order any product in the UK call Bloomsbury Software on 0171 436 9481.

To help in that communication effort we are now on the World Wide Web located at **http://members.aol.com/APL2000**. Our site has information about our products and services and innovative ways to use them. In addition, a section on technical tips will give you examples you may find useful in improving your business application.

Insight Systems / Adaytum Software

Adaytum Planner for Windows

Development of version 2.0 of our multi-dimensional business planning product, previously known as the Kunzle Planning System (KPS), is finally completed. Rolling the product out to 200 customers will take up a considerable amount of our time over the next several months, but even so we can now schedule new releases of our Client/Server product range.

SQAPL Client 3.0

Development of a new version of the SQAPL or APL Link client will be completed in the 2nd quarter of 1996, with general availability planned for August/September. The primary goal is to take advantage of enhancements to SequeLink (version 3.0), and to ODBC drivers. In particular, we would like to:

- Remove the object size limitations of previous versions of MiddleWare products, by supporting "partial bind" calls. The only remaining limits on object size will be workspace size and limits imposed by the host database.

- Support SequeLink 3.0 functionality, making ODBC-style dictionary calls and data type support available under Unix and OS/2. This will remove most of the differences which exist between the SequeLink and ODBC versions of SQAPL.
- Improved "Data Set" functions, supporting multiple keys.
- Support for ODBC calls to interrogate the list of installed drivers, driver options, and to support interactive logon panels provided by ODBC drivers.
- Implement workarounds for errors in popular ODBC drivers, in particular the Microsoft Office drivers.

We invite all our existing customers to write to us to suggest features to be included in the new release.

SQAPL Server 3.0

No new release of the SQAPL Server is planned for the immediate future. We do expect that our own use of the product in the Client/Server version of Adaytum Planner will lead to the development of a number of new features, which will find their way into a new version of the SQAPL Server later this year.

APL Pipes 1.0

We finally have the time to complete the packaging and documentation of APL Pipes, our TCP-based communications product. We will also add support for non-APL applications, so Pipes can provide access to APL services from Visual Basic and other application development tools, and access to TCP services like FTP, telnet and the World Wide Web from APL.

APL96

This year's APL conference, to be held at Lancaster at the end of July, is promising to be an APL training event of the highest calibre. Judging from the APL consulting work we have been asked to do during the last 6-12 months, downsizing of APL applications is now really happening. At APL96 you can learn about all the relevant technologies: using enclosed arrays, moving from one APL system to another, using remote databases and building distributed systems using TCP/IP and other communication tools.

Together with APL 2000, Dyadic, IBM and Soliton, we will be there to show you how to use our software in combination with their APL systems. See you in Lancaster!

Dyadic Systems Limited

Dyadic is pleased to announce that Dyalog APL/W Version 8.0 is **finished** and should be shipping by the time this edition of Vector is printed. The last few pieces of the jigsaw, including Control Structures and support for 32-bit OLE Controls, were added during February and issued to Preview Program Subscribers in March. The new OLE Control support is worthy of special mention.

Version 7 of Dyalog APL has, for a long time, supported Visual Basic Custom Controls (VBX). Whilst a *well behaved* VBX is indeed a valuable tool in the Dyalog APLer's armoury, VBX technology is (a'hem) not exactly a rigorous science, and has now all but been abandoned by Microsoft. Its replacement, *OLE Controls*, provides similar functionality, but is based upon sound object-oriented technology. OLE Controls are constructed from a properly defined set of function calls and a prescribed set of rules and protocols. In Microsoft terminology, Dyalog APL/W Version 8 acts as a fully fledged *OLE Control Container* and can therefore access any OLE Control that adheres to the rules. The good news is that so far, Dyadic has not found an OLE Control that doesnÆt work with Version 8.

When an OLE Control is installed on your computer, all of the information needed to access and manipulate the Control is recorded in the Windows Registry and Type Library. This information includes a list of Properties, Events and Methods provided by the Control together with information about the data structures they use. For example, not only does the system record that the *Crystal Reports* OLE Control has a property called *Destination*, but it also records that it may be set to 0, which means "To Window", or 1 which means "To Printer" or 2 which means "To File". Pointers to on-line help topics are also available and all this information is directly accessible in Dyalog APL.

One test of a true OLE Control Container is whether or not it can act as a browser. The proof is a Version 8 workspace named OCXBROWS which lists all of the OLE Controls installed on your system. Select one and it will create an instance of the Control together with a hierarchical view of its Properties, Events and Methods. Click on a property name and it will give you a brief description of the property, its data type and, if appropriate, a list of the values it may take. Click again and you will get full on-line help. You can also experiment with the Control directly through a standard property sheet interface.

OLE is in fact a set of technologies of which OLE Controls is just a part. In terms of its implementation, OLE Controls utilises all of the other technologies and is in a sense the most difficult to achieve. With the hardest part completed, Dyadic plans to add support for the other OLE technologies in future releases.

Causeway Graphical Systems Ltd

The highlight of this quarter was our trip to the USA and Canada, where we had the opportunity to address keen and interested APLers in Toronto and New York. At both meetings the emphasis was on the principles behind the Causeway environment, and we were able to show how our approach helps developers to manage complexity, even in very advanced Windows applications. We will also be talking at the GSE/APL Club of Germany meeting on April 23rd in Frankfurt, as well as participating in the BAA's Vendor Forum in May.

We were very encouraged by the positive news from APL 2000. We have resumed development of Causeway for APL+Win, this time with the goal of producing a much lighter-weight implementation which is very interpreter-specific. This gives us speed at the expense of cross-platform compatibility. So far, the results are quite startling — on a P90 laptop a collection of 20 forms, all watching the same text variable, can be set to notify/refresh **on keystroke**. All 20 keep up with normal typing speed.

We shall, of course, support the existing Causeway 'shareware' platform on both Dyalog 7.2 and 8 — the Windows 95 version includes many of the new Gui controls (trackbar, tree etc.) as Causeway objects.

The *NewLeaf* printing namespace is now functionally complete, and is undergoing pre-release testing at a number of Dyalog APL sites around the world. By the time you read this, it will also be available for APL+Win users to download and test, as the code has been written for maximum cross-platform compatibility. The beta programme has resulted in several useful additions, for example NewLeaf tables now support tree-structured column titles and HTML-style tags for bold, italic, sub- and super-script text both in the titles and in the table cells. Decimal alignment was also introduced for numeric columns as a result of a specific user-request, as was spooling to file for large reports (several hundred pages). NewLeaf will be priced at £400 per developer, with unlimited runtime and discounts for major APL sites. Watch for details on our web site at www.causeway.co.uk.

We are now working with a number of companies who are beginning to migrate existing APL code to the Windows platform. So far we have been able to help with advice on choice of platform, migration issues for mainframe or APL*PLUS II code, and detailed interface design. We are also continuing to work with SAP to extend the 'Organisational Architect' configuration tool to cover the entire process of setting up a company structure in the SAP R/3 system. If you would like to know more, please call Adrian or Duncan at +44 (0) 1653 696760 or Email us via 100265.1564@compuserve.com and ask for a copy of our brochure.

THE EDUCATION VECTOR

April 1996

Editor Ian Clark

This Education Vector has been reprinted from VECTOR Vol.12 No.4. VECTOR is the Quarterly Journal of the British APL Association. For more information about the British APL Association, please contact: Sylvia Camacho, 11 Auburn Road, Redland, BRISTOL, BS6 6LS. Tel: 0117-973 0036.

Contents

Ian Clark
IAC/Human Interfaces,
9, Hill End, Frosterley,
Bishop Auckland,
Co. Durham DL13 2SX.

Tel: 01388-527190
Email: 100021.3073@compuserve.com

Editorial

Not long ago, all teaching activity ceased at our village primary school. Songs were practised, decorations were cut out and pasted, flowers were planted and the place generally spruced-up by an assortment of willing little hands. Better than boring old reading and writing and 'rithmetic. A visit of royalty? A princess, perhaps, or some other endangered species?

Not at all. The school was due for an OFSTED visit. In deference to the dangerous power of these unpredictable people, the preparations were painfully OTT. In the event one teacher, the best in the school by common agreement among the consumers (the pupils and their parents) decided she couldn't be bothered fooling about at her age. She had more important things to do. Like teaching the children. Lacking the lily-white paperwork on That Day, she was bound hand and foot and cast into the exterior darkness. That is, she was declared inefficient and promptly took early retirement in an attitude of disgust tinged with relief.

I recall Hemingway's account in *"For Whom The Bell Tolls"* of the dying days of the International Fifth Brigade. According to Ernest, what really finished it off was not Franco and his merry chums pounding hell out of it, for all their support from Hitler with the latest weaponry and aircraft, but an inspector sent round by the high-ups to report on people and practices which were sapping the fighting power of the embattled volunteers. The fellow had the eye of a kite-hawk for growing tips, strong fibre and firm loyalty. Wherever he came across it he made absolutely sure it was lopped off, stripped out and rooted up.

But of course there are those who do, and those who inspect. Belief in the efficacy of more and more voluminous paperwork is a measure of the gap of confidence between the two. I remember, when I was part of a Home Office inspection team, encouraging police forces to pursue their plans to have all beat-officers' reports phoned-in to a team of audio typists. The idea was to enhance their effectiveness by relieving them of inessential paperwork. No sign of any comparable insight by the architects of OFSTED and the National Curriculum, even post-Dearing.

Talking about paperwork reminds me of the passport application I've had to fill in for my daughter recently. Not at all onerous, as official forms go, until you come to the bit where it says "...Section 11 should be completed by a Member of Parliament, Justice of the Peace, Minister of Religion ... and your heart sinks. You begin to wish you'd accepted that invitation to come *On The Square* all those years ago, when — ah! — it goes on to say "Doctor, Engineer, Lawyer, Teacher".

So that's all right then. Even I know one or two of those. But observe! There it is, in black and white. Her Britannic Majesty's Government giving formal recognition to the fact that a Teacher is a respected professional person, of similar standing to M.P., J.P., Revd.,... and a cut above Tinker, Tailor, Soldier, Sailor.

Observe something else, too. Nowhere on the form does it say "OFSTED inspector". When a teacher acquaintance saw that, she whooped with joy. She realised she could knock an OFSTED inspector into the gutter, because they're all just nuttin's, aren't they? Is there a professional qualification leading to OFSTED inspector? — let alone a national procedure for measuring their efficiency, feeding into published league-tables? I haven't heard of one. No more than I have for Members of Parliament, much less Government Ministers. But just imagine the exams they'd have to go through. The dissertations they'd have to sweat over. All those neatly-clipped folders full of photos of school lockers, and lists and lists of things to be snooped into.

Now I imagine that some OFSTED inspector will read this and cry "Unfair! My team is hardworking and dedicated. It consists of respectable individuals all highly experienced in their own various fields." All right, simmer down. I'm sure you and your colleagues do the job to the best of your ability. But what is the real nature of your job? Does the teaching profession want its fruits? Even (or especially!) if it is performed with zeal and dedication? Is OFSTED a sort of spiritual leaven to improve the quality of the nation's teaching? Or is it a political commissariat?

It cannot, by its very nature and constitution, be the former. Teaching in our (state) schools is performed, as HMG itself admits, by highly qualified professionals, on a par with Doctors, Engineers and Lawyers. Like these they merit exclusive accreditation and review of their competence by independently constituted professional societies of their peers. Not to be paraded and inspected by official appointees, graded and marshalled worse than the hapless infants they're supposed to find time to be teaching. That is treatment which even soldiers and sailors wouldn't stand for, let alone tinkers and tailors. And other local businessmen, who, by the way, are having a say in the running of schools out of all proportion to their professional standing. If they want to help, use them for what they're good for — putting their hands in their wallets, not their noses outside their own businesses.

Some people have told me that this page is too strong on politics and not strong enough on IT, APL in particular. Now I'm not responsible for the politicisation of education. That agenda has been set by others. So what has all this got to do with IT? What's it got to do with APL? Or J? Quite a lot, as I hope to show next time.

Jot-Dot-Min

an outer product of rock-bottom APL matters

In a previous JDM I let slip that one of the joys of APL for me was replacing loops by bit-arrays. Bitting instead of knitting, you might say. Throughout the ages philosophers have built calculating engines, talking heads and other logical machines, but the real father of commercial data processing isn't any of these – it's the Jacquard Loom, an engine which can weave beautiful intricate pictures in silk like the one I possess of the Kinkakuji Temple in Kyoto. Visitors think it is a photograph, until they examine it with a magnifying glass. The loom is driven by a deck of punched cards and considerably predates the Hollerith punched card popularised by IBM before electronic computers came along. Joseph Marie Jacquard produced his design in 1805; it was in some sense just a mechanical improvement on Basile Bouchon's 1725 concept, which draws an endless loop of perforated slats across the tops of the so-called needles which raise each individual warp thread, letting some rise and others fall.

You can illustrate the principle like this. Let *WARP* stand for the threads and *CARD* stand for a card (or a Bouchon slat). Choose a string of 25 pretty APL symbols to stand for the different coloured threads (if you've never used them all, now's your chance):

```
WARP←'⍟⍣⍬⍞⍟⍣⍬⍞⍟⍣⍬⍞⍟⍣⍬⍞⍟⍣⍬⍞'
```

Make *CARD* a string of 25 0s and 1s, as many elements as there are in *WARP*. A quick way to do this is to generate them at random by putting *?* in front of a vector of 25 2s. But this ends up with a string of 1s and 2s. So we make a comparison out of it (a so-called 'logical' expression) by putting `1<` in front. That gives us a sequence of FALSE and TRUE results, which happen to be 0 and 1 in APL.

```
      CARD←1<?25⍴2
      CARD
0 0 0 1 0 1 0 1 1 1 1 1 0 1 0 0 0 1 0 1 0 0 0 1 0
```

or something like that (it will be different each time).

Now we can apply *CARD* to *WARP* to raise all those threads where *CARD* is 1 and lower them where *CARD* is 0. The raised threads show on top of the cloth, the lowered threads get hidden underneath.

```
        WARP
±▼⊖⍞⊛±▼⊖⍞⊛±▼⊖⍞⊛±▼⊖⍞⊛
      CARD\CARD/WARP
  ⍞ ± ⊖⍞⊛±▼ ⍞   ⊖ ⊛   ⍞
```

CARD/WARP would simply knock out the 0-positions and close up the gaps. *CARD\...* puts the gaps back again.

Let's make a function out of the expression which makes *CARD* (just cut and paste from the session, if your APL lets you). We call it: *nextcard*, because it acts like it's getting the next card in the endless belt. This function is clever enough to count the number of threads in *WARP* so that you don't have to fix it at 25.

```
      ∇nextcard
[0]   nextcard
[1]   CARD←1<?(⍴WARP)⍴2
      ∇
```

Now there must be loads of ways of turning a sequence of numbers (or characters, or threads, or anything!) into a sequence of 0s and 1s. That's the principle of Bitting. We like it the way we've shown because you can tinker with the numbers 1 and 2 to get any proportion of 1s and 0s you like in the random stream. How? Just try some examples...

```
          ,CARD←5<?25⍴10
1 0 1 0 1 1 0 0 1 1 0 1 1 1 0 0 1 0 0 0 1 0
        ,CARD←1<?25⍴10
1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1
```

Get the idea? It's like tossing a weighted coin. The 'weight' is that first number, 1 or 5, in the expression.

The comma in front is just a lazy way to kid APL it's computing an uncommitted expression (which it outputs) rather than an assignment (which it doesn't). It simply saves us typing *CARD* each time. Be careful what you use it on, because it strings the result out flat.

Now we write a short function to behave like the working loom:

```
      ∇weave
[0]   weave
[1]   nextcard
[2]   CARD\CARD/WARP
[3]   →1
      ∇
```

It's a good use for a loop here (uncommonly, for APL). As you see, the function:

[1] Gets the next card *CARD*

[2] Applies it to *WARP* to weave a row of cloth

[3] Goes back and does it again

So here we go:

```
      weave
±  ⍞⊛±∇   ⊛±  θ⍞⊛±∇  ⍞⊛
± θ⍞⊛±∇θ   ±    ⊛   θ   ±∇θ  ⊛
±∇ ⍞⊛±∇θ⍞   ∇θ  ⊛±∇  ⍞⊛±  θ⍞
±∇θ⍞⊛  ∇θ⍞    θ   ±  θ⍞   ∇   ⊛
   ⍞  ±  θ⍞⊛±∇  ⍞    θ  ⊛    ⍞
...
```

Press the Interrupt key (<Ctrl+Break> perhaps, or <Esc>) when you've had enough. Then type a branch to clear out the suspended function:

```
      →
```

There are cleverer ways to stop a function, but they just clutter the discussion here.

Now let's change the pattern of the cloth it weaves. If we change the function: `nextcard`, that's like changing the endless belt on the loom. Make up a collection of 'nextcard' functions, `nextcard1`, `nextcard2`... and alter the function `nextcard` to simply run any one of them:

```
      ∇nextcard
[0]   nextcard
[1]   nextcard2
      ∇
```

```
      ∇nextcard1
[0]   nextcard1
[1]   CARD←¯1+?(⍴WARP)⍴2
      ∇

      ∇nextcard2
[0]   nextcard2
[1]   CARD←1⌽CARD
      ∇
```

So, to change the endless belt on the loom, edit `nextcard` to slot in a different `nextcard*` function. The version: `nextcard2` is a bit of a cheat, because it needs a valid result in `CARD` to work, which you can get by running `nextcard1` once. It simply alters the existing pattern by 'rotating' `CARD`, ie. moving the first element to the end. Here's what it looks like with `nextcard2`...

```
      weave
  ⍟ ⍣ ⍒⍟⍞⍣± ⍟   ⍒ ⍞   ⍟
 ⍒ ⍞ ±⍒⍟⍞⍣ ⍒   ± ⍟   ⍒
± ⍟ ⍣±⍒⍟⍞ ±   ⍣ ⍒   ±
 ⍒ ⍞⍣±⍒⍟ ⍣   ⍞ ±   ⍣     ⍣
± ⍟⍞⍣±⍒ ⍞   ⍟ ⍣   ⍞     ⍞
 ⍒⍟⍞⍣± ⍟   ⍒ ⍞   ⍟     ⍟ ⍣
±⍒⍟⍞⍣ ⍒   ± ⍟   ⍒     ⍒ ⍞
±⍒⍟⍞ ±   ⍣ ⍒   ±     ± ⍟ ⍣
±⍒⍟ ⍣   ⍞ ±   ⍣     ⍣ ⍒ ⍞⍣
±⍒ ⍞   ⍟ ⍣   ⍞     ⍞ ±⍟⍞⍣
± ⍟   ⍒ ⍞   ⍟     ⍟ ⍣ ⍒⍟⍞⍣
 ⍒   ± ⍟   ⍒     ⍒ ⍞ ±⍒⍟⍞⍣
±   ⍣ ⍒   ±     ± ⍟ ⍣±⍒⍟⍞
   ⍞ ±   ⍣     ⍣ ⍒ ⍞⍣±⍒⍟ ⍣
  ⍟ ⍣   ⍞     ⍞ ± ⍟⍞⍣±⍒ ⍞
...
```

(Once again, press Interrupt when you're tired of it.)

Well, it's not the Kinkakuji Temple, but Rome wasn't built in a day. Read on...

Not long after the Jacquard loom revolutionised weaving (and nearly got its inventor drowned in the river Rhône by furious Lyonnaises, who thought it would put them out of work) the theory of Cellular Automata arose in mathematicians' minds. Don't let anyone tell you that mathematicians dream up new things all the time. Often they just sweep up the philosophical droppings of technological innovation. The original challenge was this (the Napoleonic Wars not long over): imagine a line of musketeers in the mist. Each man is only able to communicate with his two neighbours, which he does once a second. Each man is an automaton and goes into a different state as a result of giving or getting

messages. Thanks to parade-ground drilling, all the automata are identical. (Remember that the drill for loading and firing a musket had something like 131 steps, so the National Curriculum had nothing on the common soldier.) Devise a system of messages and states which allows the line to get messages from one end, and this to result (several seconds later) in them all firing at the same time.

Now this problem is actually quite difficult. At least it is if you aren't allowed to build in the number of soldiers into the system, or let the end-soldier recognise he is at the end. But just to show how you might tackle the problem, let's give each soldier only two states, called 0 and 1, and a repertoire of one message which he sends exclusively to the soldier on his right. He switches to his other state when he receives the message, and he sends the message when he's in state 1 and has just been switched to state 0. Speak up at the back there... Yes, we've just turned him into a Binary Flip-Flop and constructed a Shift-Register, but you're in the wrong class, my lad.

Now let's be clever and use our Jacquard loom to model the situation. Here's the 'program' to do it, called *nextcard3*. But now it's not the 'next card' but the 'next state' for each of the soldiers, i.e. what's to happen one second later. People who write simulations call this a 'discrete simulation' and the step of one second they call an 'epoch'.

```
      ∇nextcard3
[0]   nextcard3
[1]   CARD←CARD≠(¯1↓0,CARD)
      ∇
```

Not-equals behaves like 'exclusive-or' in this context, or 'non-carrying addition' as the pioneers used to call it. Line [1] says: "Add *CARD* to itself shifted along one position using non-carrying addition". That corresponds to each soldier passing the message to the right.

Now let all the soldiers start in state 0...

```
      CARD←25⍴0
      weave
```

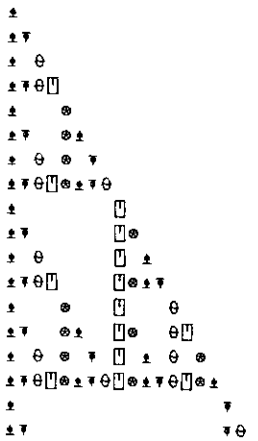
And there they'll stay... until we give the first soldier the order (i.e. the one message he understands), by setting *CARD* to 1 for that position.

(...endless blank lines, until you press Interrupt...)

```
...
weave[2]
```

So give the first soldier the order and re-start the simulation...

```
      CARD[1]←1
      →⎕LC
```



...etc.

Very pretty. A Binary Tree. Still not the Kinkakuji Temple, but we've only just started. Computers themselves may have started with Jacquard looms, but what nobody tells you is that a Jacquard loom with self-generating cards could in principle handle any computing task. Tripping over boxes and boxes of printout at my old place of work, I came to that conclusion myself, and I'm not Alan Turing. If they had only used a loom and some silk instead of forests of paper and a lineprinter, it could have made someone a nice kimono.

A less well-known fact about Alan Turing was that he was an expert at knitting (as well as bitting). He once got the whole department at Manchester trying to knit a Riemann Surface. This piece of intelligence was told me by my mother-in-law who worked there as a Computer.

No joke! 'Computer' was a job-title, like 'navvy', until the steam-navvy came along. Ma-in-law was an expert knitter too, but she never went as far as drowning Alan in the Mersey. Meanwhile, back at the Battle of Waterloo, if you want to give your soldiers more than 2 states, as you'll need to, you'll have to replace the bit-string: *CARD* with a string of integers and make the loom a little more elaborate. I'll publish the most interesting solution I receive in a future issue.

But until then, as you play with the loom, remember... you are experimenting

with Cellular Automata, the forerunner of the fashionable Neural Network. Yes, if you consider each of your soldiers to be a neurone (a nerve cell), it can 'fire' (i.e. send a message) when it's leaving a particular state, and its next state depends solely upon its current state and the messages it receives from other neurones.

In 1968, in my first job and faced with my first computer (an IBM 1130 — remember it? — and it sported APL, too!) I wrote a simple neural network simulation. Then I sat and stimulated it with the sense-switches. (Why doesn't my PC have sense-switches any more?) I was fascinated by the wallpaper that came off the lineprinter. Young as I was, I even wondered eerily if it might be 'thinking' in there, and whether it might be wrong to switch it off. Happily a perusal of the printout showed the activity gradually descending from conscious to comatose, so I reluctantly switched off the machine and caught the late train home.

You think I'm being funny? What if I had been an embryologist?

In the classical situation, each cell can only receive messages from next-door neighbours. In animals, this is probably only true for jellyfish (which have neurones almost like ours). The more advanced applications of neurones in the animal world collect bundles of fibres into cables for making trunk calls to distant parts of the network. This is the 'white matter' beneath the 'grey matter' of the human brain.

But there's plenty of mileage in a nearest-neighbour topology. If you abandon bit-strings in favour of bit-squares, you can play Conway's Game of Life (which we're coming to in the next issue). And of course, if you go to higher dimensions in the form of an n-dimensional hypercube, you can make every one of n cells a next-door neighbour of every other...

```
n←15
z←(n⍴2)⊤1
```

Or you can put 3 instead of 2 and try developing cellular automata to play Tic-Tac-Toe.

How high does your version of APL let n go until it complains: *LIMIT ERROR*? Dyalog APL lets n go up to 15. The celebrated Connection Machine goes up to 32767, I understand. Just imagine 32767-dimensional Tic-Tac-Toe. You'd have to have your wits about you.

J-ottings 9

by Norman Thomson

This is the most powerful article yet in this series; I can say this with confidence because the starting theme is the power conjunction `^:` .

The sequence "verb conjunction noun" as in `+^:2` defines a new verb. A very simple example is that of finding the *n*th term of the series 1,3,7,15,31,63,... This can be obtained either using the power verb:

```
   p=.<:@(2&^)       NB. (y.th power of 2) - 1
   p 5
31
```

or by using the power conjunction:

```
   step=.>:@+:       NB. 2y. + 1
   (step^:5)0        NB. Iteration using power conjunction
31
```

A more sophisticated use of `^:` is in standard algorithms for evaluating Fibonacci terms and series.

First the verb `fa` describes the essential Fibonacci operation, which transforms the number pair (*a*,*b*) into the pair (*b*,(*a*+*b*)):

```
   fa=.{: , (+/)     NB. Update rightmost terms in
                         Fibonacci series
```

The power conjunction can be used to execute it any required number of times:

```
   (fa^:15)0 1       NB. terms index 15 and 16 in series
                         starting 0 1
610 987
```

If it is the whole series which is required, define

```
   fb=.] , +/@(_2&{.)   NB. Join sum of last 2 terms onto
                            the series
   (fb^:15)0 1          NB. Add 15 terms to the series
                            which starts 0 1
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

These examples show clearly how strongly the power conjunction ^: is associated with iteration.

Now compare the above with a recursive approach. The Fibonacci problem is amenable to the technique which Hui and Iverson call "tail recursion", which they discuss extensively in [1], and which is also discussed in *J-ottings 3*. It is worth thinking initially about general techniques for converting iterative verbs to recursive ones, for which purpose return to step which is used below both iteratively and recursively.

```
   step=.>:@+:                NB. 2y. + 1
   (step^:5)0                 NB. Iteration using power
                                  conjunction
31
   (0:`(step@$:@<:)@.*)5      NB. Recursion using gerund and
                                  agenda
31
```

This simple case can be used as a general prototype. For fb, if the left argument is the number of terms required, the stopping condition is determined by the hook ~:$. To obviate a rank error following agenda, a conversion from a 1-item vector to a scalar must be included, leading to the stopping function

```
   stop=.{.@(~:$)
```

and thence to the recursive form

```
   fbr=. ] ` ([$:fb@]) @. stop
   5 fbr  0 1
0 1 1 2 3
```

In a similar way the recursive equivalent of fa can be obtained as:

```
   far=. ] ` (<:@[$:fa@]) @. (*@[)
   15 far 0 1
610 987
```

Another illustration of the power conjunction is Sylvester's algorithm, which takes a vulgar fraction in the range (0,1), and expresses it as a sum of best-fitting unit fractions. For example, 0.67296 is well approximated by

$$\frac{1}{2} + \frac{1}{6} + \frac{1}{159} + \frac{1}{248438}$$

The result of the algorithm can thus be expressed as a series of increasing integers, in general of indefinite length. In the above example, the series would

be (2,6,159,248438, ...). In the realisations which follow, the result series has at its head the error term, that is, the remaining fraction which has to be decomposed, so that successive results are:

0.17296, 2	(0.17296 = 0.67296 - ½)
0.00629, 2, 6	(0.00629 = 0.67296 - ½ - ⅙)
.....	

APL and J renderings of this algorithm were published by Donald McIntyre in the *IBM Systems Journal APL Anniversary edition* [2], and a further APL implementation of this algorithm is given in a *Vector* Technical Note [3]. An iterative approach in J using `^:` is significantly different in style. Taking *f* as a fraction, first define a verb `nig` to mean "next integer greater than 1/*f*", so that, for example `nig 0.333` is 4, and `nig 0.334` is 3.

```
nig=.>.&%      NB. Next integer greater than reciprocal
```

The next steps in the algorithm are to use the hook `-%` to derive `x.-1/y.`, and to apply this following (atop) `nig`, which in turn is applied to the current error term. This leads to the verb

```
srr=.(-%@nig)@{.      NB. Subtract reciprocal from
                          outstanding remainder
```

which is the core of the algorithm.

Finally adjust the result series to its standard form, ready to start all over again:

```
syl=.srr,{.,nig@{.)  NB. Update remainder, and add new
                         integer to right
```

Thus the original problem is solved by

```
   (syl^:4)0.67296
8.10094e_12 2 6 159 248438
```

Here is the result of applying three Sylvester steps to the fractional part of pi:

```
   z=.(o.1)-3              NB. Fractional part of pi
   (syl^:3)z               NB. Numerical error, 1st 3
                               Sylvester terms
7.77959e_9 8 61 5020
```

— and so 3 + ⅛ + 1/61 + 1/5020 gives pi correct to 7 decimal places.

The Technical Note [2] also describes a somewhat similar algorithm which reduces a decimal fraction to its equivalent in a different number base. Again the power conjunction suggests an iterative approach in J. The arithmetic required to find the fractional digits of 0.67296 in base 5 is:

```
  | 67296
3 | 3648
1 | 824
4 | 12
0 | 6
3 | 0
```

At every step, the decimal fraction to the right of the vertical bar is multiplied by 5, and the integer and fractional parts of the result are written to the left and right of the bar on the following line. The required digits in base 5 emerge down the column to the left of the bar. Take the number base as left argument and as right argument the fraction to the right of the bar, joined to the series so far.

The actions left and right of the bar imply that two verbs are required to supply fractional and integer parts. These are `1&|` and `<.` respectively which combine as a gerund following (atop) the multiplication:

```
g=. 1&|`<./.@*      NB. fractnl and integer parts
                        following multn by base
```

The verb `/.` makes the gerund to its left executable, and can also be written `:0.`

```
   5 g 0.67296
0.3648 3
```

For steps after the first, it is necessary to pick off the fractional part using `{.` (head), so define the fork

```
   f=. [ g {.@]

   5 f 0.67296
0.3648 3
   5(f^:3)0.67296
0.12 4
```

In order to complete the goal of carrying forward the digits as they are found, amend f to the final form

```
   fdiv=.([ g {.@]),}.@]        NB. apply g to fraction and
                                    append digits so far

   5(fdiv^:3)0.67296
0.12 4 1 3
   5(fdiv^:5)0.67296
0 3 0 4 1 3
```

Conversion to recursive forms can readily be applied to both syl and fdiv to obtain

```
   sylr=. ]`(p@(<:@$:]))@.(*@[)        NB. Recursive syl; x.
                                           is countdown parm

   4 sylr 0.67296                      NB. cf. (syl^:4)0.67296
8.10094e_12 2 6 159 248438

   fd=.(({:@[)g {.@]),}.@ ]            NB. Countdown parm
                                           to left of x.

   fdivr=.]`([fd-&1 0@[$:])@.(*@{.@[)  NB. Recursive fdiv

   3 5 fdivr 0.67296                   NB. cf. 5(fdiv^:3)0.67296
 0.12 4 1 3
```

In the illustrations above the verbs conjoined to ^: have all been monadic. Dyadic verbs raise some interesting questions, such as:

(1) + is a commutative verb, that is (a+b)-:b+a for all numeric a, b. Is +^:2 a commutative verb?

(2) Does 2(+^:2)3 mean 2+2+3 or 2+3+3?

The answers depend on the general property of dyadic verbs that the left argument "binds" more strongly than the right. This means that 2+3 means (2+)3, that is the noun left argument acts like an adverb and creates an intermediate verb "addtwo", say, which then works on the right argument. This device is known as "currying" after the American logician H.B. Curry, and it makes it easy to answer both the above questions:

(1) No, since "addtwo" applied twice to 3 is 3+2x2, which is not the same as "addthree" applied twice to 2, which is 2+2x3.

(2) <2(+^:2)3 means "addtwo" twice to 3, so the first of the two alternatives is correct.

References

[1] *Representations of Recursion*, Roger KW Hui & Kenneth E Iverson, Proceedings of APL95 Conference, pp.91-97

[2] *Language as an Intellectual Tool: From hieroglyphics to APL*, D.B.McIntyre, IBM Systems Journal, Vol.30 No.4, 1991, pp.562-563

[3] *Technical Note: Fractions as Sequences of Integers*, Norman Thomson, Vector Vol. 12, No. 3, pp. 130-131

Towers of Hanoi from 15 Years Ago

by Adrian Smith

For historical amusement, here is a rerun of the Hanoi Tower solution from *Crab APL*, Issue 10, page 11, March 1981. This recognises the binary pattern in the moves, and builds a boolean array, which it then adjusts and indexes to get the sequences of moves. Note that in modern APLs, the use of the ~~ sequence should not be necessary, as they do this sort of thing for you.

```
    ∇ R←HANOI N;DISC;PEG;INT
[1]   ⍝ NON-RECURSIVE SOLUTION TO THE HANOI TOWER PUZZLE
[2]   ⍝
[3]   ⍝ FIRST SET UP THE MOVE NUMBERS IN BINARY FORM; ONLY THE
[4]   ⍝ TOPMOST <ONE> BEING LEFT ON. (THE ~~ SPEEDS THINGS UP A BIT!)
[5]    INT←<⍀0~~(N⍴2)⊤¯1+⍳2*N
[6]   ⍝ THE DISC TO BE MOVED IS GIVEN BY THE POSITION OF THE ONES
[7]    DISC←(⍳N)⌈.×INT
[8]   ⍝ MAKE ALTERNATE ROWS +/- TO FORCE DISKS TO
[9]   ⍝ CYCLE WITH OPPOSITE HANDEDNESS
[10]   INT←INT×⍉((1+⍴INT),N)⍴N⍴ 1 ¯1
[11]  ⍝ THE POSITION OF EACH DISK AT ANY TIME IS THE CUMULATIVE
[12]  ⍝ VALUE OF THIS, BUT EXPRESSED IN MODULO-3.
[13]  ⍝
[14]  ⍝ TO GET THE NEW PEG AT EACH MOVE, WE INDEX BY DISK NO. FOR
[15]  ⍝ THE ROWS; MOVE NO. FOR THE COLUMNS. THIS EFFECT CAN
[16]  ⍝ BE ACHIEVED EITHER WITH A SCATTERED-PT SELECTION FUNCTION
[17]  ⍝ OR (AS HERE) WITH A 1 1⍉ TO GET THE DIAGONAL OF AN
[18]  ⍝ INTERMEDIATE OBJECT.
[19]   PEG← 1 1 ⍉(1+3|+\INT)[DISC;]
[20]   R←DISC,[0.5]PEG
    ∇
```

APL Product Guide - Updates

compiled by Gill Smith

VECTOR's exclusive APL Product Guide aims to provide readers with useful information about sources of APL hardware, software and services. We welcome any comments readers may have on its usefulness and any suggestions for improvements.

Pressure on space occasionally prevents us from printing the complete guide, however updates will always be listed. We do depend on the alacrity of vendors to keep us informed about their products. Anyone who is not included in the Guide should contact me to get their free entry — see address below.

We reserve the right to edit material supplied for reasons of space or to ensure a fair market coverage. The listings are not restricted to UK companies and international suppliers are welcome to take advantage of these pages.

For convenience to readers, the product list has been divided into the following groups ('poa' indicates 'price on application'):

- Complete APL Systems (Hardware & Software)
- APL Interpreters
- APL-based Packages
- APL Consultancy
- Other Products
- Overseas Associations
- Vendor Addresses
- World Wide Web and FTP Sites

Every effort has been made to avoid errors in these listings but no responsibility can be taken by the working group for mistakes or omissions.

We also welcome information on APL clubs and groups throughout the world.

All contributions and updates to the APL Product Guide should be sent to Gill Smith, at Brook House, Gilling East, York, YO6 4JJ. Tel: 01439-788385, Email: 100331.644@Compuserve.com

APL INTERPRETERS

COMPANY	PRODUCT	PRICES(£)	DETAILS
Dinosoft OY	Dyalog APL/W for Windows	poa	Finnish distributor of Dyalog APL products.
	Dyalog APL for Unix	poa	See Dyadic's listing for product details.

APL PACKAGES

COMPANY	PRODUCT	PRICES(£)	DETAILS
UNIWARE	Execucalc	--	Withdrawn
(for Dyalog 7.2 & 8)	DLL Parser version 2.0	FF1450	Auto-generates ⎕NA instructions for DLLs and gets constant values and structures.
	DELPHI Forms Generator	FF950	Takes a Delphi form and turns it into a Dyalog APL program which builds an identical form.
	Uniware Toolkit/W Vn. 2.4	FF39000	Fast relational database system, and complete set of Windows tools for RAD in Dyalog APL/W.

APL CONSULTANCY

COMPANY	PRODUCT	PRICES(£)	DETAILS
Dinosoft OY	Consultancy	poa	Specialised in very large databases.

OVERSEAS ASSOCIATIONS

GROUP	LOCATION	JOURNAL	OTHER SERVICES	Ann.Sub.
SE APL Users Grp	Atlanta, Georgia	SEAPL Newsletter	Quarterly meetings	$10

WORLD WIDE WEB SITES

ORGANISATION	URL
APL 2000	members.aol.com/APL2000
APL-385	www.demon.co.uk/apl385
Causeway	www.causeway.co.uk
Dyadic Systems Ltd	www.dyadic.com
IBM APL2	www.torolab.ibm.com/ap/apl/apl2.html
Mackay Kinloch Ltd	ourworld.compuserve.com/homepages/Alastair_Kinloch
MicroAPL Ltd	www.microapl.co.uk
SIGAPL	www.acm.org/sigapl
Rex Swain	www.pcnet.com/~rhswain
Toronto SIG	www.sigapl.mtnlake.com/sigapl/welcome.html
Jim Weigang	www.chilton.com/~jimw/welcome.html

FTP SITES

ORGANISATION	DOMAIN NAME
IBM APL2	ps.boulder.ibm.com/ps/products/apl2/
Toronto toolkit	see Toronto SIG home page
Waterloo Archive	archive.uwaterloo.ca/ftparch/languages/apl
APL-to-ASCII	archive.uwaterloo.ca/languages/apl/workspaces/aplascii

VENDOR ADDRESSES

COMPANY	CONTACT	ADDRESS, TELEPHONE, FAX, EMAIL etc.
ACM/SIGAPL	Donna Baglio	ACM, 1515 Broadway, New York, NY 10036 USA Tel:+1 (212) 626-0606 Email: baglio@acm.org
Active Workspace Ltd	Ross D Ranson	Moulsham Mill Centre, Parkway, Chelmsford, Essex, CM2 7PX, UK. Tel: 01245-496647; Fax: 01245-496646.
Adaptable Systems	Lois & Richard Hill	49 First Street, Black Rock 3193, Australia. Tel: +61 3 589 5578 Fax: +61 3 589 3220
Adaytum Systems		13 Great George Street, BRISTOL BS1 5RR UK Tel: 0117-921 5555
Adfee	Bernard Smoor	Dorpsstraat 50, 4128 BZ Lexmond, Netherlands. Tel +31-3474-2337, Fax: +31-3474-2342
Andrews	Dr Anne D Wilson	23, The Green, Acomb, YORK YO2 5LL, UK Tel: 01904-792670
APL-385	Adrian Smith	Brook House, Gilling East, York YO6 4JJ UK. Tel: 01439-788385 Fax: 01439-788194 Email: 100331.844@compuserve.com
APL Bay Area Users Group APLBUG	Lewis H. Robinson (Sec)	1100 Gough St, Apt 14A, San Francisco, CA 94109, USA Tel: +1 (415) 928-2058 Email: frgp21a@prodigy.com
APL Club Austria	Erich Gall	IBM Österreich, Obere Donaustrasse 95, A-1020 Wien, Austria
APL Club Germany	Dieter Lattermann	Rheinstraße 23, D-69190 Walldorf, Germany. Tel: +49 6227-63469 Compuserve: 100332,1461
The APL Group Inc	Stuart Sawabini	644 Danbury Road, WILTON, CT 06897 USA. Tel: +1 (203) 762-3933 Fax: +1 (203) 762-2108
APL Interest Group, South Africa	Mike Montgomery	Private Bag X11, Rivonia 2128, South Africa Tel: +27 (11) 803-7200 Fax: +27 (11) 803-9134 Email: mikemont@spi.co.za
APL People / Software	Jill Moss	The Old Malthouse, Clarence St, BATH, BA1 5NS UK. Tel: 01225-462602
Association Francophone pour la promotion d'APL	Dr. Gérard Langlet	SCM, C.E. Saclay, F-91191-Gif sur Yvette, France. Fax:+33 1 69-08-79-63
Atlantis Software	Arthur Whitney	1105 Harker Avenue, Palo Alto, CA 94301 USA
BACUS	Joseph de Kerf	Roolnberg 72, B-2570 Duffel, Belgium. Tel: +32 15 31 47 24
Beautiful Systems, Inc.	Jim Goff	308 Old York Road, Suite 5, Jenkintown, PA 19046, USA Tel: +1 (215) 886-2636; Fax: +1 (215) 886-4888
The Bloomsbury Software Co Ltd	Peter Day	3-6 Alfred Place, Bloomsbury, London WC1E 7EB UK. Tel: 0171-436 9481; Fax: 0171-436 0524; CompuServe: 100010,1467
Camacho	Anthony Camacho	11 Auburn Road, Redland, Bristol BS6 6LS UK. Tel: 0117-9730036. email: acamacho@dx.compulink.co.uk Reutemet (Sharp): ACAM
Ray Cannon		21 Woodbridge Rd, Blackwater, Camberley, Surrey GU17 0BS UK Tel: 01252-874697 Email: 100430.740@compuserve.com
Paul Chapman		51B Lambs Conduit Street, London WC1N 3NB UK. Tel: 0171-404 5401. Compuserve: 100343,3210
Causeway Graphical Systems Ltd	Adrian Smith, Duncan Pearson	5 The Maltings, Castlegate, MALTON, North Yorks YO17 0DP UK Tel: 01653-696760 Fax: 01653-697719 Compuserve: 100265,1564
Chicago SIG	Larry Mysz	836 Highland Drive, Chicago Heights, IL 60411 USA C'serve:73040,3032
Cinerea AB	Rolf Kornemark	Skyttegatan 25, S-193 00 Sigtuna, Sweden.
CODEWORK	Mauro Guazzo	Corso Cairoli 32, 10123 Torino, Italy. Tel: +39 11 885168 Fax: +39 11 812 2652
ComLog Software	Jeff Pedneau	PO Box 5570, Derwood, MD 20855 USA Tel: +1 (301) 990-7063 Email: jeff@softmed.com
CPCUG	Lynne Startz	Capital PC User Group, 51 Monroe Street, Suite PE-2, Rockville, Maryland 20850-2421, USA. Tel: +1 (301) 762-9372 Fax: (301) 762-9375.
David Crossley		187 Le Tour du Pont, Quartier Le Mourre, 84210 ST DIDIER, France Tel: +33 90-66-08-87
CYBEX AB	Lars Wentzel	Gruvgatan 35B, S-421 30 V. Frölunda, Sweden. Tel: +46 31-45 37 40. Fax: +46 31-45 24 23.
Peter Cyriax Systems	Peter Cyriax	22 Hereford Road, London W2 4AA UK. Tel: 0171-229 5344
Danish User Group	Per Gjerlof	Email: gjerper@inet.uni-c.dk

Datatrade Ltd.	Ian Tomlin	1 & 2 Sterling Business Park, Salthouse Road, Brackmills, Northampton, NN4 0EX UK. Tel: 01604-760241
Dinosoft OY	Pertti Kalliojärvi	Lönnrotinkatu 21C, 00120 Helsinki, FINLAND. Tel: +358 0 70028825 Fax: +358 0 70028824
Dogon Research	Dick Bowman	2 Dean Gardens, London E17 3QP UK Tel: 0181-520 6334 Email:bowman@api.demon.co.uk
Dutch APL Association	Bernard Smoor (Sec)	Postbus 1341, 3430BH Nieuwegein, Netherlands. Tel: +31 3474-2337
Dyadic Systems Ltd.	Peter Donnelly	Riverside View, Basing Road, Old Basing, Basingstoke, Hants RG24 0AL UK. Tel: 01256-811125 Fax: 01256-811130
E & S Associates	Frank Evans	19 Homesdale Road, Orpington, Kent BR5 1JS UK. Tel: 01689-824741
Evestic AB	Olle Evero	Bertelisvagen 12A, S-146 38 Tullinge, Sweden Tel&Fax: +46 778 4410
FinnAPL		Suomen APL-Yhdistys RY, FinnAPL RF, PL 1005, 00101 Helsinki 10, Finland
General Software Ltd	M.E. Martin	22 Russell Road, Northholt, Middx, UB5 4QS UK. Tel: 0181-864 9537
Greymantle Associates	George MacLeod	Bartrum House, Ravens Lane, Berkhamsted, Herts, HP24 2DY UK Tel: 01442-878065 Email: 100412,1305@compuserve.com
Hartford CT Group	Bob Pomeroy	Mass Mutual Life, 1295 State St, Maildrop F465, Springfield, MA 01111 USA Tel: +1 (413) 788-8411x2838
H.M.W.Trading Systems Ltd	Stan Wilkinson	Hamilton House, 1 Temple Avenue, Victoria Embankment, London EC4Y 0HA UK. Tel: 0171-353 8900; Fax: 0171-353 3325; Email:100020.2532@ compuserve.com
HRH Systems	Dick Holt	3802 N Richmond St, Suite 271, Arlington, VA 22207 USA Tel: +1 (703) 528-7624; Email: dick.holt@acm.org
Michael Hughes		28 Rushton Road, Wilbarston, Market Harborough, Leics. LE16 8QL UK. Tel: 01536-770998
IAC/Human Interfaces	Ian A. Clark	9 Hill End, Frosterley, Bishop Auckland, Co. Durham DL13 2SX UK Tel: 01388-527190. Compuserve: 100021,3073
I-APL Ltd	Anthony Camacho (for queries, order forms)	11 Auburn Road, Redland, Bristol BS6 6LS UK. Tel: 0117-9760036 email: acamacho@cix.compulink.co.uk Reutemet (Sharp): ACAM
	J C Business Services (for pre-paid orders only)	56 The Crescent, Milton, Weston-super-Mare, Avon, BS22 8DU UK Tel: 01934-625181
IBM APL Products	Nancy Wheeler	APL Products, IBM Santa Teresa, Dept M46/D12, 555 Bailey Avenue, San Jose CA 95141, USA. Tel: +1 (408) 463-APL2 (=2752) Fax: +1 (408) 463-4488 Email: APL2@vnet.ibm.com Cserve: GO IBMAPL2
Impetus Ltd	Cedric Heddle	Rusper, Sandy Lane, Ivy Hatch, SEVENOAKS, Kent TN15 0PD UK Tel: 01732-885126
INFOSTROY	Alexei Miroshnikov	3 S. Tulenin Lane, St. Petersburg 191186 Russia. Tel:+7 812-312-2673 Fax:+7 812-311-2184 Email:aim@infostroy.spb.su
Insight Systems ApS	Morten Kromberg	Nordre Strandvej 119A, DK-3150 Hellebæk, Denmark Tel:+45 42 10 70 22 Fax: +45 42 10 75 74 Email: insight@inet.uni-c.dk
Intelligent Programs Ltd	Mike Bucknall	9 Gun Wharf, 130 Wapping High St, London E1 9NH Tel: 0171-265 1120
Interprocess Systems Inc.	Stella Chamberlain	11660 Alpharetta Highway, Suite 455, Roswell, Georgia 30076, USA Tel: +1 (404) 410-1700. Fax: +1 (404) 410-1773 Cserve: 70373,2676
Iverson Software Inc.	Eric Iverson	33 Major Street, Toronto, Ontario, Canada M5S 2K9 Tel: +1 (416) 925-6096; Fax: +1 (416) 488-7559
JAD Software	David Crossley	580 Eyer Drive, #81 Pickering, Ontario, Canada L1W 3B7 Tel: +1 (905) 837-1895 Fax: +1 (905) 831-5172
Japan APL Association		23-2-302 Hiromichi, Adachi-ku, Tokyo 120, Japan
Kestrel Consulting	Mark Harris	Business & Technology Centre, Bessemer Drive, Stevenage, Herts SG1 2DX UK. Tel: 01438-310155 Fax: 01438-310131
Lingo Allegro USA Inc.	Victoria H. Fl	113 McHenry Road, Suite 161, Buffalo Grove, IL 60089 USA Tel:+1 (708) 459-7529 Fax:+1 (708) 459-8501 Email: info@lingo.com
Mackay Kinloch Ltd	Alastair Kinloch	519 Webster's Land, Edinburgh EH1 2RX, Scotland, UK Tel/Fax/Answerphone: 0131 228 3580 Pager/Voicemail: 01426 98 3858 Compuserve: 100010,33
MasterWork Software Ltd	Fraser Jackson	PO Box 56-036, Tawa, Wellington, New Zealand. Tel and Fax: +64 (4) 232-4440 Email: 100242.2535@compuserve.com
Melbourne APL Group	Harvey Davies	CSIRO Div Atm Res, Private Bag No.1, Mordialloc, Victoria 3195, Australia Tel: +61 3 586 7574 Fax: +61 3 586 7600 Email: hld@dar.csiro.au

Mercia Software Ltd.	Gareth Brentnall	Holt Court North, Heneage Street West, Aston Science Park, Birmingham B7 4AX UK. Tel: 0121-359 5096. Fax: 0121-359 0375
MicroAPL Ltd.	David Eastwood	South Bank Technopark, 90 London Road, LONDON SE1 6LN UK Tel: 0171-922 8866 Fax: 0171-928 1006 Email: MicroAPL@microapl.demon.co.uk
Ellis Morgan	Ellis Morgan	Myrtle Farm, Winchester Road, Stroud, Petersfield, Hants UK. Tel: 01730-263843
New York SIG APL	Nestor Nelson	PO Box 138, NY 10185-0002, USA
Oasis Systems	Theo Zwart	Lekstraat 4, 3433 ZB Nieuwegein, Holland Tel: +31 30 60 66 336 Fax: +31 30 60 65 844 Email: 100447.431@compuserve.com
Optima Systems Ltd	Paul Grosvenor	Airport House, Purley Way, Croydon, Surrey CR0 0XY UK. Tel: 0181-781 1812 Fax: 0181-781 1999
Potomac APL SIG	John Martin	51 Monroe Street, Plaza East Two, Rockville MA 20850-2421 USA Tel: +1 (301) 762-9372 Fax: +1 (301) 762-9375 Email:jam@acm.org
QB On-Line Systems	Philip Bulmer	5 Surrey House, Portsmouth Rd., Camberley, Surrey, GU15 1LB UK. Tel: 01276-855880 Fax: 01276-855301
Renaissance Data Systems	Ed Shaw	PO Box 421, Georgetown, CT 06982, USA. Tel: +1 (212) 864-3078
RE Time Tracker Oy	Richard Eller	PO Box 363, FIN-00101 Helsinki, Finland. Tel: +358-0-400 2777
Rochester APL	Gary Dennis	Soliton Associates, 1100 University Avenue, Rochester, NY 14607 USA Email: gsd@ipsalab.tor.soliton.com
The Rochester Group Inc.	Robert Pullman	50 S.Union St., Rochester NY 14607-1828, USA. Tel: +1 (716) 454-4360. Fax: +1 (716) 454-5430
Rome/Italy SIG	Mario Sacco	Casella Postale 14343, 00100-Roma Trullo, Italy Email: marsac@vnet.ibm.com
SE APL Users Group	John Manges	413 Comanche Trail, Lawrenceville, GA 30244, USA Tel: +1 (770) 972-3755 Email: seapldoc@aol.com
Shandell Systems Ltd.	Maurice Shanahan	Chiltern House, High Street, Chalfont St. Giles, Bucks HP8 4QH UK.
Snake Island Research Inc	Bob Bernecky	18 Fifth Street, Ward's Island, Toronto, Ontario M5J 2B9 Canada Tel: +1 (416) 203-0854 Fax: +1 (416) 203-6999 Email: bernecky@eecg.toronto.edu
SOCAL (South California)	Roy Sykes Jr	Sykes Systems Inc, 4649 Willens Ave, Woodland Hills, CA 91364-3812 USA Tel: +1 (818) 222-2759 Fax: +1 (818) 222-9250
Soliton Associates	Laurie Howard	Soliton Associates Ltd, Groot Blankenberg 53, 1082 AC Amsterdam, Netherlands Tel: +31 20 646 4475 Fax: +31 20 644 1206 Email:sales@soliton.com
SovAPL	Alexander Skomorokhov	PO Box 5061, Obninsk-5, Kaluga Region, Russia Email:askom@apl2.obninsk.su
Strand Software Inc	Anne Faust	19235 Covington Court, Shorewood MN 55331 USA Tel: +1 (612) 470-7345 Email: amfaust@aol.com
Rex Swain	Rex Swain	8 South Street, Washington, CT 06793 USA. Tel: +1 (860) 868-0131 Fax: +1 (860) 868-9970 Email: rhswain@acm.org
SWAPL	Stuart Yarus	PO Box 210367, Bedford, Texas 76095, USA Tel: +1 (817) 577-0165 Compuserve: 73700,2545 Email:syarus@unicomp.net
SwedAPL	Christer Ulfhielm	Novator Consulting Group AB, Svärdvägen 11C, S-182 33 Danderyd Sweden Tel: +46 8 622 63 50 Fax: +46 8 622 63 51 CServe: 100341,404
Swiss APL User Group		Swiss APL User Group, CH-3001, Bern 1, Switzerland Email: sl@ifi.unizh.ch
Sydney APLUG	Rob Hodgkinson	PO Box 1511, Macquarie Centre, NSW 2113, Australia Tel:+61 2 257 5313
Sykes Systems Inc	Roy Sykes Jr	4649 Willens Ave., Woodland Hills, CA 91364, USA Tel: +1 (818) 222-2759 Fax: +1 (818) 222-9250
Toronto SIG	Ben Best	PO Box 55, Adelaide St. Post Office, Toronto Ontario M5C 2J5, Canada Email: benbest@io.org http://www.sigapl.mtnlake.com/sigapl/welcome.html
Uniware	Eric Lescasse	Tour Neptune, Cedex 20, 92086 Paris la Defense 1, France. Tel: +33 (1) 47-78-78-00. Fax: +33 (1) 40-90-04-11
Wickliffe Computer Ltd	Nick Telfer	76 Victoria Rd, Whitehaven, Cumbria, CA28 6JD UK. Tel: 01946-692588
Warwick University	Prof. Jeff Harrison	Dept of Statistics, University of Warwick, Coventry, CV4 7AL UK Tel: 01203-523369
Zark Incorporated	Gary A. Bergquist	23 Ketchbrook Lane, Ellington CT 06029, USA. Tel: +1 (203) 872-7806

AP119/W - a First Look

by Timo Laurmaa (100316.3367@compuserve.com)

Introduction

AP119/W is a product which hooks Dyalog APL/W to TCP/IP, today regarded as *the* communications protocol to connect computers together. It was written by Andrei Kondrashev of Lingo Allegro, Inc. The product sells for US$300, but early birds attending APL95 were able to get a price of just below $80. Prices as low as these are not likely to generate significant revenues for Lingo Allegro, but that does not necessarily create a problem: Lingo's main business is in consulting, and the primary purpose of tools like AP119/W or GDDME (see *Vector* 10.2 and 10.3) is to leverage their consulting business, in which connections to the mainframe APL2 world are valuable.

AP119/W was apparently designed to be fully compatible with AP119, IBM's TCP/IP auxiliary processor, with a target of providing the communication between APL user interfaces in Windows and APL2 legacy applications on the mainframe. However, judging by the enhancements that have been implemented in 8 months between version 1.0 beta and 2.1 beta (my current version), AP119/W is developing into a sophisticated but easy to use package, which connects APL/W with other APL versions and non-APL environments such as Internet servers and browsers.

Installation and Documentation

The installation and invocation of AP119/W are almost identical to those of GDDME, a GDDM emulator by the same author (*Vector* 10.2 pp. 45–48). *ap119.exe* is an independent task, and it uses DDE to communicate with APL/W. It is enough to copy ap119.exe into the directory of your choice, indicate the location by the *PATH* variable and call an APL function such as:

```
      ∇ TCPSHARE;R
[1]    ⍝ Sign on to AP119/W using TcpIp
[2]     →(2=⎕SVO'TcpIp')/0
[3]    ⍝Set initial value
[4]     TcpIp←''
[5]     R←'DDE:'⎕SVO'TcpIp'
[6]    ⍝ Load and run AP119
[7]     ⎕CMD(PATH,'AP119 -',⎕WSID,' -TcpIp ')''
[8]    ⍝ Be sure that AP119 has answered
[9]    Wait:→(1≠2⊃⎕SVS'TcpIp')/Wait
[10]   ⍝ Set full interlock
[11]    R←1 ⎕SVC'TcpIp'
      ∇
```

Provided that a TCP/IP stack and the file *winsock.dll* are present, a TcpIp icon pops up (unless hidden by a startup parameter) and the shared variable `TcpIp` can be assigned AP119 commands. The use of a cover function is recommended:

```
      ∇ Z←TCP X;APRC;TCPRC
[1]    ⍝Pass a request to TCP/IP via AP119
[2]     TcpIp←X
[3]     Z←TcpIp
[4]     (APRC TCPRC Z)←Z
[5]     →(APRC=0)/0
[6]     ('AP119 error: ',⍕APRC TCPRC)⎕SIGNAL 666
      ∇
```

The *Programmer's Reference* follows Lingo Allegro's tradition of high quality documentation. It is a useful combination of a 30-page function reference and a 20-page introduction to TCP/IP and sockets. I learned the basics of socket programming by reading IBM's *APL2 System Services Reference*, and I am convinced that the AP119/W manual is at least equally suitable reading for TCP/IP beginners.

Working with Sockets

In TCP/IP, a *socket* is the fundamental concept onto which all protocols (such as FTP or HTTP) are based. AP119/W supports stream sockets and datagram sockets, of which I will only discuss the former. The following AP119 call allocates a new socket:

```
      ⎕←TCP 'SOCKET' 'STREAM'
0
```

Since sockets are used for two different purposes, communicating with a known partner, or waiting for new connections initiated by so far unknown users, I found

the following piece of code to be useful, because it combines the AP119 calls that are always required:

```
      ∇ Z←L TCPSOCKET V;A;IA;PN;PT;S
[1]    ⍝ Get a new socket
[2]    ⍝ For listening if L=1
[3]     (IA PN)←V                      ⍝ IP address, port nr
[4]     Z←S←TCP'SOCKET' 'STREAM'       ⍝ Get a new socket
[5]     →L/LB                          ⍝ Jump if LISTENing
[6]     A←TCP'BIND'S 0 '0.0.0.0'       ⍝ Bind the socket
[7]     A←TCP'CONNECT'S PN IA          ⍝ Address IA, port PN
[8]     →0
[9]    LB:A←TCP'BIND'S PN'0.0.0.0'     ⍝ Bind to port PN
[10]    A←TCP'LISTEN'S 5                ⍝ Socket for listening
      ∇
```

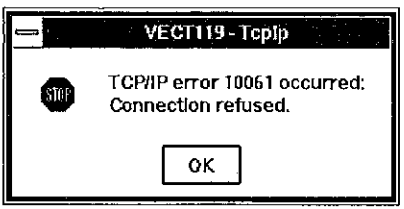
If you want to create a client connecting to the World Wide Web server of Statistics Finland, the IP address of which is 193.166.0.71, the connection to their WWW port 80 is established by:

```
      □←0 TCPSOCKET '193.166.0.71' 80
1
```

The result 1 indicates that socket number 1 is successfully connected to port 80 of *www.stat.fi*. If the IP address or port number are incorrectly specified, AP119 blocks for about half a minute and then returns an error code:

```
      AP119 error: 1 10061
```

AP119 has an optional error display facility which shows the error codes together with a description in a message box:



Message boxes are very useful when designing and testing an application, but the possibility to turn them off is invaluable in production systems, particularly servers, where the application is intelligent enough to trap non-zero error codes and take the necessary action.

If instead of connecting to a known server you want to create a server of your own, you allocate a socket by:

```
      ⎕←1 TCPSOCKET '' 3850
2
```

The result 2 indicates that socket number 2 is now listening to port 3850 (a not-so-well-known port of your choice). A client can now connect to this socket by knowing your IP address and your port number 3850. There are several ways to proceed and find out if a client wants to connect:

```
      ⎕←TCP 'ACCEPT' 2
3 1256 158.152.217.93
```

The server has accepted the connection from socket 2 and created *a new socket* number 3 for the communication that will take place between your server and the client, whose IP address is `158.152.217.93`. The original socket 2 remains listening for new connections. The problem with *ACCEPT* is that the system blocks until a client requests a connection. This is where the *SELECT* command will be useful:

```
      1⊃TCP 'SELECT' (2 3) '' '' 15
0 1
```

AP119 will wait at most 15 seconds (the last item in the vector is timeout) for data to be read from sockets 2 or 3, and returns a 1 for each socket where an event has taken place. In this case, there has been no activity in socket 2, but data can be read from socket 3, which was created above for the communication between this server and user `158.152.217.93` . *SELECT* can be set to wait indefinitely (timeout 0) or return immediately (timeout negative).

The data sent by user `158.152.217.93` can now be read:

```
      ⎕←TCP 'RECV' 3 0 'N'
Hello, are you there?
```

and a response can be sent:

```
      ⎕←TCP 'SEND' 3 0 'N' 'Yes I am. Who are you?'
22
      ⍝ 22 characters were sent
```

These examples show that to get going with AP119/W is quite easy. Building full scale applications with multiple users, possibly working on different platforms, is more challenging but by no means terribly difficult.

Building Real Applications

Considering that AP119/W is an APL interface to a standard *winsock.dll*, the application designer needs to be aware of pitfalls such as:

- Data is sent and received as simple character strings. Unless only APL/W users are involved, the application must convert numeric, nested and multidimensional to and from a transfer form. I developed a simple `⎕TF` emulator for APL/W in order to exchange data between APL2/370 and APL/W.
- If APL characters are sent across different platforms, the application must convert between different `⎕AV`s.
- TCP/IP chops the data into chunks of 2 kb or less. Even though the SEND command may accept very long strings, on the other side the RECV command only returns one chunk at a time, and the application must catenate them together and know when end-of-data has been reached. If only APL/W users are involved, AP119/W goes quite a bit further (in release 2.1).

AP119/W Release 2.1

For some weeks now I have had a beta version of the new release of AP119. Some of the enhancements make the application designer's life just a little bit easier, some of them are bound to revolutionise the way of APL programming with sockets.

- The new conversion option `W` allows for an automatic conversion from any APL array into a character string (when sending) and vice versa (when receiving). This option also caters for catenating small chunks together before returning the full APL variable.
- Andrei has been clever enough to notice that this approach may have its downside: if AP119 receives a large variable using the `W` format, the execution of the `RECV` command does not terminate until the whole variable has been built from possibly hundreds of small chunks. Network failures or problems on the sender's side may even lead to a case where AP119 never receives the entire data. The solution is to receive the data without conversion as character strings and at the end apply a new command `CONVERT` which returns an APL array.

- The commands *GETHOSTID* and *GETHOSTNAME* have been extended to use Internet name servers to convert between host names and Internet addresses. You can now query IP addresses by specifying host names such as *apl-385.demon.co.uk* or *www.stat.fi*.

- The gem of release 2.1 is the introduction of event driven socket programming.

Event Driven Socket Programming

The traditional approach to detect – typically in a server – if any of the active sockets has pending client requests is to call *SELECT* with a reasonable timeout. The longer the timeouts, the less the application can do other things. With the availability of the new *NOTIFY* command AP119 is now able to make the TCP/IP kernel interrupt the APL application when an event of interest occurs in the network.

The TCP/IP events get to the APL workspace via an invisible form, from which APL/W is able to catch the information related with the events. Therefore, the invocation of AP119 should be followed by statements like

```
'TCPF'⎕WC'FORM' 'APL-119' ('COORD' 'PIXEL')
'TCPF'⎕WS('EVENT' 5 'TCPEVENT')('VISIBLE' 0)
```

The sockets allocated later on will notify this form about their TCP/IP events if a statement like this is called for socket *S*:

```
TCP'NOTIFY'S('TCPF'⎕WG'CAPTION')(+/1 8 32)
```

Whenever socket S is ready for *RECV* (1), *ACCEPT* (8) or *CLOSE* (32), the callback function *TCPEVENT* is executed, for example:

```
     ∇ TCPEVENT MSG;A;EVENT;ERROR;IA;P;S;⎕IO
[1]   ⍝ A simple TCP/IP event handler
[2]    ⎕IO←1
[3]    (ERROR EVENT)←0 256⊤3↓MSG          ⍝ Get event nr
[4]    SN←4↓MSG                           ⍝ Socket number
[5]    →(EVENT=1 8 32)/READ,ACC,CLS       ⍝ Jump to event
[6]   READ:A←TCP'RECV'SN 0 'N'            ⍝ Receive chunk
[7]    ⍎'DATA',(⍕SN),',←A'                ⍝ Append to data
[8]    →0
[9]   ACC:(S2 P IA)←TCP'ACCEPT'S          ⍝ Get new socket
[10]   ⍎'DATA',(⍕S2),'←IA'                ⍝ New data variable
[11]   →0
[12]  CLS:A←TCP'CLOSE'SN                  ⍝ Close socket
[13]   STORE⍎'DATA',⍕SN                   ⍝ Store data
     ∇
```

This is all it takes to implement a simple TCP/IP protocol, which in the example above collects data sent by clients and, when the client closes the socket, calls *STORE* to save the collected data further to a suitable place. The beauty in the functionality of *NOTIFY* is that when *ACCEPT* creates a new socket *S2* based on the listening socket *S*, *S2* inherits the properties of *S*, including the *NOTIFY* settings.

Suggestions for Improvement

The new version leaves little to be desired. Nevertheless, a full-blown TCP/IP product might well offer additional goodies such as:

- standardisation of the TCP/IP solutions of different APL vendors would be necessary to allow for the effortless transmission of APL arrays between different APLs.
- a built-in data compression facility could reduce transmission times dramatically. Standardisation would be an issue here, too.
- it seems that AP119/W does not work well with all versions of *winsock.dll*. An additional warning in the documentation about this might be useful.

Who is AP119/W for?

For any company or institution, where TCP/IP is in place, AP119/W is an easy and inexpensive starting point for solving APL connectivity problems. It is not (and is not intended to be) a turnkey solution, but it provides all facilities for the rapid development of cross-platform connectivity between APL2 and APL/W – or within a network of APL/W PCs.

AP119/W is particularly attractive for APL2 shops, where Windows has been selected as the standard PC operating system and where the unavailability of OS/2 excludes the use of IBM's built-in co-operative processing facilities across different platforms. Instead of APL2/2 and its AP119, Dyalog APL/W with AP119/W can provide a nice path to program-to-program communication. This typically means accessing mainframe data from PCs, but a less traditional approach would be to let mainframe programs have real time access to PC or LAN based data.

AP119/W offers attractive possibilities for home users, too. I installed the new WinCim 2.0 for integrated CompuServe and Internet access, and was excited to see that the setup allowed me to establish an AP119/W connection from my home PC in Switzerland to an APL2/370 workspace that Jaakko Ranta had)*LOAD*ed in Finland. Using the same concept, any computer connected to Internet can be used

from almost anywhere in the world, for a price close to what a local telephone call costs. Quite a tool for a travelling APL consultant who can use, design and maintain applications far away from the customer.

Conclusion

AP119/W release 1.0 was a solid piece of programming, which provided the means for building client/server applications across different APL platforms. Release 2.1 goes several steps further by introducing the concept of event driven socket programming and by facilitating the exchange of APL arrays across the network. Unlike GDDME, which was Lingo Allegro's solution for the declining market of old APL2/370 applications run on PCs , the company now has a product for an exploding (I hope) market of Internet and intranet connectivity. The ideal place, however, for AP119/W would be as an integral part of Dyalog APL/W.

JAD SMS - An Object Management Environment for APL*PLUS II V5

reviewed by Jon Sandles

Introduction

In the last *Vector* I wrote about problems I had maintaining objects within the Causeway environment for Dyalog APL. There seemed to be a need for some sort of object management system and version control management system to allow maintenance of objects across different releases of Causeway.

Well over a year ago *Vector* was sent a copy of JAD SMS - "System Management Software for APL" - to review. For some reason, which now escapes me, I found problems getting the software going, but then I decided to look again into the whole topic of "Object Management", and as a result I mailed the creator of JAD SMS - David Crossley - and asked him for the latest version. Fortunately I had no problems getting the software going this time, so I began to explore what it could do for me .....

What Does JAD SMS Do?

JAD SMS is written in and used to manage functions/variables (referred to in the rest of this article as *objects*) in APL*PLUS II v5 , although there are plans to port it to other APLs. The evaluation copy came bundled with the Toronto toolkit, which is a useful place to start in seeing how the software works. The system is made up of a bunch of APL functions (unfortunately locked) which can be `)copy`'ed into your workspace when you need them. They can be run interactively or embedded into your own functions. Also supplied is a DOS-based pop-up windows interface which calls all the lower-level functions for you. In this review I concentrate on this full screen interface, but bear in mind a great strength of this system is that all the lower-level code is well documented and you can build your own interface/routines based on it.

Why do we need JAD SMS? You've probably tried to organise your workspaces before. Namespaces have been helpful for this purpose. Indeed, the way JAD SMS stores functions in a hierarchical tree structure is much the same as storing your functions in a bunch of namespaces - so what more does it give us? Having stored your functions in namespaces, you've also probably used that same structure in many different workspaces - indeed all you needed to do was `)copy` the

namespace tree to proliferate whole chunks of working systems from one application to another. But what happens when you find a bug right in the core of one of these proliferated functions? All of a sudden you need to update all the different occurrences of the function - remembering where they all are could be quite tricky. In a multi-developer environment the problem is even worse - there's always someone who likes to 'fiddle' the utilities to work the way they want them to. If you do a wholescale update of every workspace you can lay your hands on with the fixed function you could create three different problems .... (and probably more).

(i) You fix the bug - but maybe introduce new problems since the original developer had already worked around the problem - the work-around could now have undesirable effects.

(ii) You fix the bug - but you lose any fudges other developers have built into specific applications over time.

(iii) You fail to fix the bug - none of your workspaces work anymore!

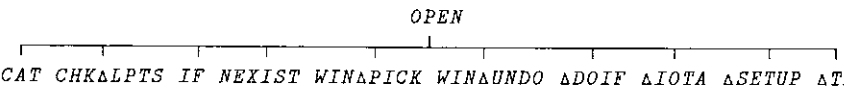
Thus, implementing the bug-fix could be very time consuming, and testing the bug-fix could be even harder. An object management utility should enable the development team to work together and share the same utility set - but be flexible enough to allow developers to adjust them in a controlled manner. It does this by providing information about the status of utilities in different applications, and the status of the master utilities. This information allows developers to update common code across a number of applications in such a way that they can have a high degree of confidence that the change will not introduce new bugs.

A Test Example

To illustrate how far JAD SMS takes us towards this goal I developed a test example. I have a set of functions which provide reasonably advanced print capabilities from DOS - allowing the user to select printers and define the printer type and select the paper orientation etc. (if the printer is capable). The functionality is quite simple - `OPEN 'FRED'` opens a channel for printing, `∆PR MAT` sends data to the open print channel - and `CLOSE 'FRED'` closes the print channel and spools the print to the printer. The functions - although quite simple - rely on quite a lot of standard utility code and when I tried to isolate them from the rest of the workspace I was surprised that I actually needed a lot more than just the 3 functions to make them run. In fact I needed a monstrous 37 functions! Only two of these were specific to the print - all the others were utility functions which are used hundreds of times in hundreds of workspaces.

This should illustrate how dependent code is on the standard utilities that you take for granted. Not only that, but if a bug was to crop up in one of these utilities (it **does** happen) how important it would be to identify where you need to upgrade your workspaces.

To help illustrate further here is the calling tree of the function *OPEN* .....



Some of these functions have their own calling trees calling other lower-level utilities.

JAD SMS allows us to store our utilities in a structure very similar to the DOS directory structure. A top-level utility library is in effect an APL sharefile (called a *database* in JAD SMS) storing all the utilities, but within that you can break the utilities up into a hierarchical structure made up of directories of objects. This is not the only way of grouping utilities in JAD SMS but it is the most important as directories are the level at which security is defined – and hence the level at which most of the JAD SMS utility functions work.

Security is an important aspect of JAD SMS and it works using an access matrix approach similar to many APL sharefile systems. It means you can give read-only access to utilities to all but the few who are actually responsible for the utility code, but also create new 'development' areas where you give access to completely different people. You may even want to prevent people from accessing areas of the utility set completely. Security is very well defined in JAD SMS and it seems to provide all the functionality you would want from such a system.

So what does it look like ....?

(Remember you don't have to use the full screen interface – all the lower-level functions are provided and fully documented – but I found the full screen interface very useful to get the hang of what the system can do.)

```
TEST Tree
Esc Quit   F1 Help    F2 Users   F3 Files   F4 Search  F5 PermTab  f6 □Fdup
F8 Print   F9 Export  F12 Ver    Pg ↑/↓
JS    Dir Actions:  Make  Delete  Rename  Move  Attributes  Enter

 TEST     GEN
          PRINTER
          WIN
```

Here I have created a database file called TEST in which I created 3 directories called GEN, PRINTER and WIN. This was simple enough to do - press <F3> 'Files' and add the file 'TEST'. Then press <M> 'Make' and type in the name GEN etc. I made these 3 directories to store my printing utility code. In GEN I was going to put any general utilities that come in useful all the time, in PRINTER the utilities specific to printing, and in WIN the utilities specific to windowed output. I then loaded my printer workspace and copied in JAD SMS and populated the directories with the appropriate functions.

This was also very simple to do - I selected the directory I wanted to populate and hit Enter - this takes you into the directory screen (shown later) - hitting F2 <Add> brought up the additions screen. Here you can manually type in the names of the objects you want from your workspace adding into this directory. Far easier, as long as the workspace is not too big, is hitting F4 <AllWs> to get all your functions/variables from the workspace and deleting all the ones you don't want in this directory. (It also only picks up those objects that aren't already in the library.)

I did this for all three directories and the PRINTER directory screen looked like this ....

```
TEST.PRINTER Directory                                    OPEN edited

Esc Cancel F1 Help   F2 Add     F3 OK      F4 Search F5 Action F6 Restore

F7 Sort    F8 Print  F9 Attribs F10 Select F11 Cmpre F12 Tree  Pg ↑/↓

 1 of 10    JS        Keys:  ALT+A..Z, B C D E H L O P R S
```

Name	Type	Date	Time	Class	# of Ver	Ver #	Rel #	User
CHKLPTS	f	96-02-29	21:16		1	1	0*	JS
CLOSE	f	96-02-29	21:16		1	1	0*	JS
OPEN	f	96-02-29	21:16		1	1	0*	JS
OPTIONS	f	96-02-29	21:16		1	1	0*	JS
ΔPDESC	v	96-02-29	21:16		1	1	0*	JS
ΔPR	f	96-02-29	21:16		1	1	0*	JS
ΔPTYPE	v	96-02-29	21:16		1	1	0*	JS
ΔSETUP	f	96-02-29	21:16		1	1	0*	JS
ΔΔΔprdest	v	96-02-29	21:16		1	1	0*	JS
ΔΔΔpreset	v	96-02-29	21:16		1	1	0*	JS

This screen shows all my functions and variables that are directly related to printing - the *v* and the *f* for functions and variables. You then have the timestamp of when that object was last changed. On the right hand side is the user who last made a change to the object. The 3 columns of 1s and 0s represent the number of versions - the version number and the release number of that particular object. *"Why do you need all those?"* you are probably thinking.

Remember what we are trying to do here - to manage a bunch of functions that are in use in a variety of systems. When the utility developer upgrades/fixes a utility you don't always get your system administrator to pull in the new version immediately. So, although you have fixed the bug, some of your users may well be running old versions. If you do not keep a copy of the old version you might get very confused when they re-discover the bug you have just fixed in 5 years' time. JAD SMS allows you to maintain both old production versions of code and new development versions of code all in the same database directory.

By Way of Illustration ...

To illustrate let's assume I have need to fix the top-level function *OPEN*. After editing, JAD SMS reports the following for that function ...

Name	Type	Date	Time	Class	# of Ver	Ver #	Rel #	User
OPEN	f	96-03-06	19:41		2	2	0*	JS

There are now two versions of the function (I elected not to overwrite the original but create a new version). The current version is 2. (These are always the same unless you delete old/unwanted versions - version numbering is always

absolute.) The previous version is kept unless you elect to overwrite - and is read only for all versions bar the most recent. You can easily scroll between versions to compare any changes etc.

The release number is still 0*. What does the * mean ? Well - the release numbering scheme is a way of protecting versions of code that you have released to your application developers. Once you release a set of utility code from JAD SMS you apply a release number and all the latest versions of code get the * removed and the new release number is applied (unless it already is released status in which case it would already have had its * removed and it would keep its original release number). Once the latest version of an object is released it cannot be changed any more. To change the function definition you must create a new version of the function. Application developers must then retrieve either the latest versions of the functions (if they are developing new code) or the latest releases of the functions (if they are updating production systems).

To show this in action I applied a release number of 0.1 to all my printer functions (I would then communicate to all teams that the latest release is 0.1). This clears all the *s from the latest versions of each object (indicating that they can no longer be edited). To check this I then edited *OPEN* again. When I saved it, JAD SMS automatically created a new version and re-added the * to the release number (to indicate that it is an unreleased object).

Name	Type	Date	Time	Class	# of Ver	Ver #	Rel #	User
CLOSE	f	96-02-29	21:16		1	1	0.1	JS
OPEN	f	96-03-06	20:26		3	3	0.1*	JS

Notice it is still part of the 0.1 release set - that is because JAD SMS always applies the latest release number to any new versions - and because I hadn't yet defined the next release number it used the most recent existing one. I'm not sure I like the way it does this - I think it is confusing. To get the functionality I was expecting you always need to create a development release level that gets applied to all changes following the last release you did.

In JAD SMS applying a release protects the latest versions of all code - but allows you to create new versions of code that still get attached to that release number. Hence application developers might get different sets of code depending on when they pull in the latest release of the code. It would be better if code changes after a release were flagged as 'unreleased' until the code is next released. Thus developers have the choice of pulling in the released set of code or alternatively all the latest versions of the code if they want the development set. In reality it does this with the * - the latest released code is the highest release

number/version combination that does not have an *. (But this is a bit too confusing!)

The JAD SMS version/release mechanism illustrates that you need quite a complex method of tracing utility code effectively.

Associating Objects

The final aspect of JAD SMS I'm going to talk about (and I'm skipping over quite a lot of functionality) is how you associate objects with each other to ease retrieval. Going back to my PRINTER example: the *OPEN* function had quite a complex calling tree and some of the lower functions had calling trees themselves. When a developer retrieves *OPEN* he would ideally like the option to retrieve all the required sub-functions to make that function work - rather than having to call the function 20 times until he has copied all the required functions identified by 20 *VALUE ERROR*s!

JAD SMS provides a method of achieving this by defining associated objects. Each object has an optional list of associated objects - these are objects that will also be retrieved/printed if the base object is selected. The association lists have to be set up manually - so if you add a call to a new function in an object you need to remember to add it to the association list. Some object management systems provide association automatically by lexically analysing the function every time you save it/retrieve it.

There are other aspects of the JAD SMS method of associating that are fairly crude. When you retrieve an object (whatever version) which has associated objects you only ever get the latest versions of the associated objects - this could lead to inconsistencies - in reality it ought to search for the latest version from the same release as the retrieved object. Also associated objects need to reside in the same directory - if you look back at the calling tree for *OPEN* it called functions that resided in all three of the directories that I defined. Fortunately JAD SMS provides a second mechanism for this ...

Object linking allows you to reference objects in multiple directories whilst only really storing them once. Thus, going back to the *OPEN* function again, we need all the functions it calls linked into the PRINTER directory. To do this you have to highlight the functions you want to link to the PRINTER directory in their source directory, and then press <F5> and select 'link to another directory'. (I then tried changing the source of one of the linked functions and the link appears to be to the latest version always rather than to a specific version - again this could cause inconsistencies?)

I was now in a position to associate all the sub functions of *OPEN* - I did this - a simple case of typing in all the names on the association screen that comes with every object/version. (This isn't validated so typing errors could be costly!) My PRINTER directory now looked like (first page only) ...

```
TEST.PRINTER Directory                                          OPEN edited

Esc Cancel F1 Help    F2 Add      F3 OK       F4 Search  F5 Action  F6 Restore
F7 Sort    F8 Print   F9 Attribs F10 Select F11 Cmpre  F12 Tree   Pg ↑/↓

  1 of 18    JS        Keys:  ALT+A..Z, B C D E H L O P R S

 Name                      Type   Date      Time  Class       # of Ver Rel  User
                                                                Ver  #   #
 CHKALPTS                   f    96-02-29  21:16                1   1 0.1    JS
 CLOSE                      f    96-02-29  21:16                1   1 0.1    JS
+OPEN                       f    96-03-07  20:46                4   4 0.3*   JS
 OPTIONS                    f    96-02-29  21:16                1   1 0.1    JS
 APDESC                     v    96-02-29  21:16                1   1 0.1    JS
 APR                        f    96-02-29  21:16                1   1 0.1    JS
 APTYPE                     v    96-02-29  21:16                1   1 0.1    JS
 ASETUP                     f    96-02-29  21:16                1   1 0.1    JS
 ΔΔΔprdest                  v    96-02-29  21:16                1   1 0.1    JS
 ΔΔΔpreset                  v    96-02-29  21:16                1   1 0.1    JS
 CAT                        f 1  96-03-07  22:39                2   2 0*     JS
 IF                         f 1  96-02-29  21:12                1   1 0*     JS
 NEXIST                     f 1  96-02-29  21:12                1   1 0*     JS
 ADOIF                      f 1  96-02-29  21:12                1   1 0*     JS
 AIOTA                      f 1  96-02-29  21:12                1   1 0*     JS
 Classes:  none defined
```

Notice the '+' beside *OPEN* - this indicates that there are associated objects. Notice the 'l' against the functions towards the bottom of the screen - these are the objects I linked in from other directories - they are read only when viewed from this directory - but notice that *CAT* has had its version incremented even though I originally linked version 1 of *CAT*. I subsequently created a new version of *CAT* just to see what happened to the linked version.

To illustrate (a) the power of some of the low-level functions provided with JAD SMS and also the point of all this messing about ....

```
      )CLEAR
CLEAR WS
      )FNS

      )COPY 2 SMS23
SAVED 04/12/1995 15:38:37
      )FNS
jcompare   jdelete    jfile jprint     jselect    jsms       jsurrog
jcopy      jedit      jget  jretrieve  jsetup     jstore
```

```
      '+A' jretrieve 'PRINTER; OPEN'
11 objects retrieved
OPEN  CAT   IF    CHKΔLPTS    NEXIST      WINΔPICK    WINΔUNDO    ΔDOIF
ΔIOTA ΔSETUP      ΔTAB
```

The '+a' in the left argument requested all the associated functions - you can see that this worked fine - but some of these functions had sub-functions as well - to pull back all these I would have to repeat the process I went through for *OPEN*. This would be very time-consuming and difficult to maintain for a large utility set. However, in the list of possible to extensions to the system there is a mention of being able to retrieve called functions automatically - this would remove the need for typing in all the associated function names. (Although I expect you'd still need to do the object linking bit?)

Summary

JAD SMS provides a sound multi-user environment for sharing multiple versions of utility code and allowing multi-user development and retrieval. It also provides a good method of defining the status of code via the release codes and status symbol (*). There are many features I haven't reviewed - printing, code searching, multi-user access and function creation, and the lower-level functions that are available. The lower-level functions could be embedded within your workspaces to automatically retrieve latest versions of utility code. Also developers can use them to automatically add their own changes to utility code. The full screen interface was reasonably good and the online help also pretty good. The only real gripe I had is that I found the manual exceptionally difficult to dip into - you really have to read the whole thing from cover to cover and even then I found it very difficult to look things up quickly.

JAD SMS is currently being ported to Dyalog APL's GUI world. A limited-functionality prototype will be available this summer. The main visual difference is that the first two screens shown in this article will be combined in the style of Windows File Manager/Explorer. Further information can be requested via email to "crossle@io.org" or mail the address for JAD Software in the product guide.

Dyalog-8 Control Structures and Native Files: some Timings

by Adrian Smith

Summary

In a previous Vector (*"Review of APL*PLUS III Control Structures"*, Vol.11 No.1 p.84) I celebrated the arrival of control structures in APL*PLUS as a generally good thing, and it is a delight to report that not only have Dyalog followed suit, but that they have faithfully copied the exact syntax of the Plus III model. The table below reruns in Dyalog-8 some timings I did to investigate the speed of iteration (as against each) in Plus III.

I have also done some rough timings on the new native file support, as I was interested to see whether it would be a worthwhile speed improvement for *NewLeaf* to spool pages "native" rather than buffering them in the workspace or using component files. The second table simulates a report of between 12 and 600 pages, each of 4567 bytes (this needs a 12Mb workspace to complete the times in column-1), and you can see that (a) native files a very fast indeed, and (b) turning off the ASCII translation makes almost no difference. All times were taken on a Dell P90 with a fast IDE hard disk.


```
Control Structures (vvv←5000⍴'fat' 'cat')
=========================================

                          APL*PLUS III      Dyalog/W Vn-8
                          ------------      -------------
      process¨vvv             0.33s             0.38s
      loopy vvv               0.49s             0.68s

Native Files (all times in ms)
==============================
Pages        PG←PG,pg      pg ⎕fappend 1    pg ⎕nappend ¯1
                                              ASCII    raw
-----        ---------     -------------     -------  -----
12              42              521             220    212
40             234              800             330    315
100           1250             1615             673    570
200           6900             2840            1099    960
400          19000             5500            1854   1750
600          43000             7400            2691   2570
```

The file timings all involved creating a new file, appending each page to it, then erasing it. The times are marginally reduced if you tie and clear an existing file.

RECENT MEETINGS

This section of Vector documents all British APL Association meetings, and any other events of interest to the APL world. If you have recently attended any gathering which you feel would be interesting to Vector readers, please let the Editor have a brief note, and we will include it here.

Many thanks to Marc Griffiths of the Toronto APL SIG for sending us a prompt and thorough write-up of Adrian Smith's presentation to the Canadian APL group on 26th February.

Causeway and NewLeaf in Toronto

notes by Marc Griffiths

Background

Adrian Smith took advantage of a business trip to New England to visit Toronto and talk to the Toronto APL SIG on February 26. The meeting took place in a lecture theatre in the Medical Sciences Building at the University of Toronto, the site of the APL'93 Conference. This contributed to a pleasant combination of *déjà vu* and anticipation.

Adrian had organised his presentation into three segments: (i) an introduction to Causeway, (ii) a demonstration of the Organisational Architecture application software written in Dyalog APL/W version 7.1 for SAP AG, and (iii) a look at the NewLeaf report designer, the latest product from Causeway Graphical Systems. Unfortunately, the LCD display panel and Adrian's laptop computer were unwilling or unable to communicate.

Unfazed, Adrian produced some markup pens and proceeded to illustrate the first part of his talk through the older medium of acetate on an overhead projector. Parts (ii) and (iii) were conducted later in scrums arranged around the lectern to look at the screen on the laptop.

Causeway

Adrian sketched the 'Flipper' form to illustrate the principles underlying Causeway. The Flipper consists of a Windows form with a single text input field together with two control buttons, Flip and Close. Clicking on the Flip button reverses the text which the user has typed in the input field (see the notes from Adrian's Helsinki workshop in *Vector* Vol.11 No.3). For the benefit of those in the audience who hadn't seen Flipper or any Causeway application before, Adrian distributed handouts which showed the form and the Event Action table associated with the Flip button.

It was apparent that the audience contained people with a wide range of familiarity not only with Causeway, but also with graphical user interfaces (GUIs) and the concepts behind object-oriented programming. Questions were asked almost from the outset of the presentation, and the evening was highly interactive. For some in the audience, the failure of the LCD display panel was a blessing in disguise.

Without the distracting magic of a live demonstration, we were able to reflect upon the simplicity of Causeway's structure, and appreciate how far Causeway goes towards removing the need for a lot of low-level coding in prototype development and application programming. In fact, one could say that it does for GUI design and implementation what APL does for application programming: it conceals the low-level detail to allow the analyst/programmer to focus on both designing an application that solves a business problem, and enabling the user to interact as effectively as possible with that application.

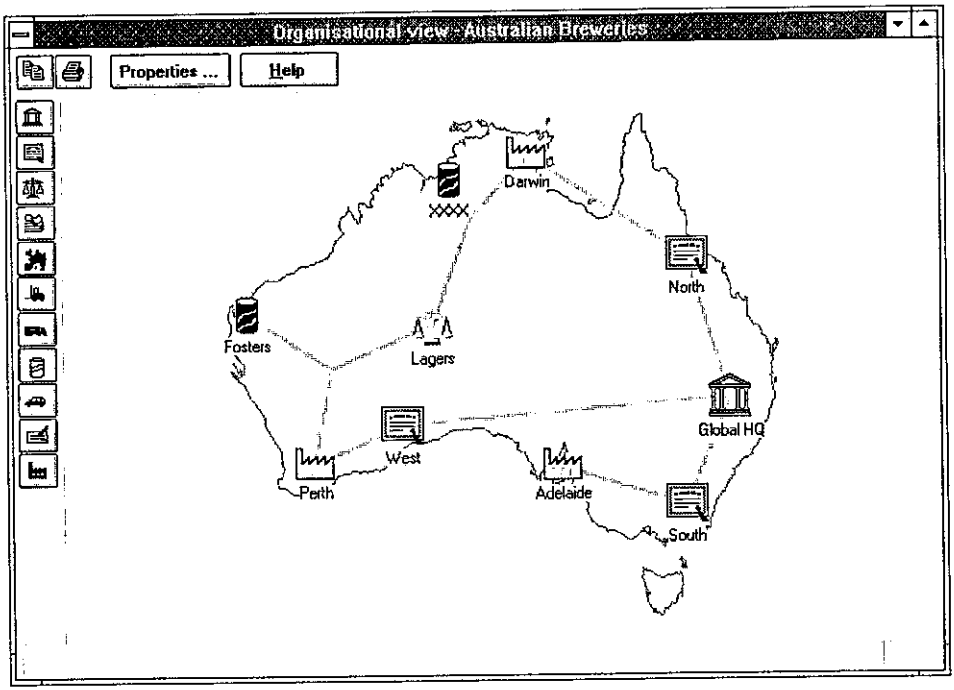
In the course of answering questions, Adrian offered a number of insights and tips with respect to using Causeway:

- Think of forms and applications in terms of the events and actions (conditional and unconditional) which are associated with the constituent objects.
- Don't be afraid to use global variables in the application. Because each form has its own execution stack, there is no central control within the application, but forms must occasionally share data. Localise variables within a form when they must be shared by its children but are not needed outside the form.
- The model used to control the 'watching' and 'hollering' associated with changes to variables is that of a centralised monitor rather than a network of direct object-to-object messages. The monitor maintains lists of the variables being watched, and of the objects which need to be notified when any of the variables is changed. When the monitor is told about a change to a variable, it notifies each of the objects which had previously registered an interest in any changes to that variable. In circumstances where the notifications may be cyclical, using an appropriate condition in the Event/Action tables will terminate the updates.
- Normally, the 'holler' message is sent when the focus leaves a field. To refresh either a large number of inter-related fields, or a field which involves time-consuming calculations or file I/O, remove the variable name from the 'holler' list in every object in the form. Then, provide the user with a Calculate or Refresh button which need do nothing other than holler the variable name, or which executes an APL function which performs the calculations before hollering. This is analagous to switching off automatic repagination of a lengthy Word document and selecting Repaginate Now from the Tools menu.

Causeway is being rewritten for Dyalog APL/W Version 8. It will take advantage of namespaces to avoid workspace name conflicts and to provide some performance improvements.

Organisational Architecture Application

This is a user interface developed by Adrian and Duncan Pearson using the Causeway development environment. It is PC-based and supports the configuration of the R3 client/server integrated business system written by SAP AG in Germany. It is particularly significant because it is not a form-filling, button-clicking, chart-drawing front-end which many people view as a standard GUI. The *Organisational Architect* application is a direct manipulation interface which allows the user to construct a graphical model of the firm's business processes from a pre-defined collection of business entities. The entities can be seen as classes which range from reporting (cost centres) and legal (subsidiaries) to the physical (factories, distribution points).



The user creates an instance by selecting a business entity from a vertical palette of icons located at the left edge of the screen and dragging it into the main window. The user fills in a few fields of a small pop-up information form to set

certain properties of this instance, such as its name. The user then proceeds to link it to other icons already in the window by using the mouse to draw lines between them. Where necessary, another small form may appear to allow the user to describe certain characteristics of the relationship between the connected instances.

When the model is complete, the information about the business entities and their connections is transmitted to the SAP configuration module in a form which it can understand. The performance of the application in redrawing the icons and links was certainly acceptable on the Pentium laptop. The fact that the Organisational Architect application has been distributed worldwide by SAP on CD-ROM is a tribute to its robustness.

NewLeaf

To greatly understate its functionality, *NewLeaf* is a reporting utility. More accurately, it is a page description package which uses collections of page description objects/elements to describe the layout of individual pages which together comprise a report, much as Causeway uses collections of GUI objects to describe forms.

Information about downloading a Dyalog APL/W beta-test version of NewLeaf from the APL385 Website (see Product Guide) had been provided in the meeting announcement, so some of us had been able to give it a test drive beforehand. Still surrounded by quite a crowd, Adrian provided a quick overview of NewLeaf on his laptop by running a few of the demonstration functions in the workspace and displaying the output on the screen.

He explained that the NewLeaf package comprises three namespaces: *class* contains the element descriptions; *leaf*, the principal namespace, contains the functions which are used by the application to construct the report; and *ps* contains functions which translate the report, held in an ASCII-text page description format, into a form which can be viewed on-line, saved to file, or sent to a printer.

The main object is the **frame** which bounds a rectangular area on a page. Text can be placed in a frame, in which case any text which extends to the right or below the edges of the frame is clipped. Alternatively, text can be flowed into a frame. Each flowed text vector is treated as a paragraph. Each paragraph is wrapped automatically between the sides of the frame and, if it extends beyond the bottom of the frame, it will flow into the next frame, which may be on another page (this "autoflow" property can be switched off).

As well as frames, NewLeaf handles objects such as rules, timestamps, page numbers, drop caps, bitmaps, and graphics from *Rain*. The programmer has control over font selection, font size, style (plain, bold, italicised), and leading; paragraph indents left/right/first line, space before/after, and alignment.

In addition to facilities to handle text as standard paragraphs, NewLeaf allows bullet-formatted text with a choice of bullet from a solid circle, numeric or lowercase alphabetic character (the latter two are incremented automatically). The big contribution is a special group of functions specifically designed to handle tables. Provided they can be expressed in a tree structure, the text for column titles can span more than one column. Lengthy tables are broken between rows and the titles appear automatically at the top of the next page.

Summary

Despite the equipment failure, no one left the meeting disappointed. After being shown two products which manage low-level, detailed coding of the user interface and report generation, many in the audience were able to breathe more easily at the prospect of once again concentrating on higher level business and system design problems.

A demonstration of a direct manipulation interface was a bonus in terms of providing a good deal of food for thought, and reminding us what GUIs can and ought to be for certain types of application.

GENERAL ARTICLES

This section of Vector is intended for general readers who have a working knowledge of APL or J. Authors are encouraged to submit articles which illustrate effective APL applications.

In this issue, we have a comprehensive catalogue of likely migration problems, which should be considered by anyone who needs to move from a mainframe environment to Windows on a PC. We also have the final part of John Sullivan's series on arithmetic with large integers, where he explores some of the applications of these techniques to data encryption.

Errata

In Duncan Pearson's 3DTTT article, astute readers will have noticed that at the bottom of page 101, function `DrawPoints` appears to refer to a non-existent sub-function called `ref`. This is actually a catbar ⍪ which Winword interprets as some kind of field code when it appears adjacent to a circlestile ⌽. Beware!

While we are on the subject, two readers have noticed that back in Vector 10.1 we suffered from the notorious ↑ problem. The line of code on page 120 should read:

```
changefont←15↑46 0 0 ptr 0 flags(0 128 128 0)
```

... and it is a good idea to throw in a `GlobalUnlock` before the `GlobalFree` on page 122.

Migration From APL2 to APL/W

by Rex Swain (rhswain@acm.org)

Introduction

I've recently spent a fair amount of time migrating workspaces from IBM's APL2 to Dyadic System's APL/W. This article, a review of various features that will require conversion, is an effort to help anyone else who is either contemplating or actually charged with a similar migration task.

Of course, all us techies know that what *really* should be done is a complete re-design and re-coding of the entire application! Segregate the logic, user interface, and database stuff, create a proper event-driven GUI, and so on. But sometimes project funding realities require compromise, typically spelled p-o-r-t.

APL is remarkably portable in general. But if you are considering an APL2 to APL/W migration, there are many issues that will slow down what you may have thought would be a quick job. Hopefully the advice here will enable you to anticipate the full scope of a migration effort, save you a bit of work, and help you react calmly to errors in code that used to run just fine.

I should point out that Dyalog APL doesn't purport to be identical to APL2, and considering the circumstances, it is surprisingly similar. Dyalog was based on STSC's NARS version of the language. APL2 (even the pre-release IUP version) wasn't released until they had fixed the language specification and coded most of the product. It is Dyadic's intention to develop closer compatibility with APL2, and they have already made good progress.

My specific experience involved moving from APL2 version 2.2 under VM/CMS (sometimes called "APL2/370") to Dyalog APL/W version 7.1 under Windows for Workgroups, but I believe that the bulk of these issues would be relevant with other releases.

Types of Differences

There are several classes of differences that will require changes:

1. Many differences are easy to find with a simple workspace searching tool. For example, $\square TF$ is not supported in APL/W, so scan all functions for any occurrence of $\square TF$, and replace with an emulation function. In this case, you can find *every* instance without ever actually executing the application code.

2. Some differences require a more sophisticated search that may not find 100% of the cases. E.g., optional left arguments must be enclosed in braces in APL/W function headers. So search all dyadic functions for ⎕*NC* of the left argument, and where found, wrap the argument in braces. This will *almost* always suffice, but one can easily imagine tricky cases where the argument is never referenced in any obvious way.

3. Some differences are very difficult to find mechanically. E.g., `1 0 1/⍤4⍴⊂⍳3` works in APL2 but causes a *LENGTH ERROR* in APL/W; you need to enclose the left argument to get this working.

The good news about these first three classes is that they blow up with errors when executed, so even if you can't find them all programmatically, at least if you exercise your application long enough they will announce themselves. The bad news, of course, is that this assumes that your application is straightforward enough that you *can* test all possible execution paths and data cases. (More probably, your end-users will eventually become conversion beta testers.)

Unfortunately, there are two more classes...

4. Some differences are a nightmare! E.g., the spacing of results from monadic format may be slightly different in APL/W, depending on the depth and structure of its argument. In the workspaces I worked on, there seemed to be 1E6 uses of monadic format, and of course the structure of the argument is not always immediately obvious. These are the worst kind of differences, because the code executes just fine and the result is only subtly different, and the difference may or may not be significant in the context of your application.

5. Finally, there is all the operating system dependent stuff. Some of this can be *extremely* difficult (or impossible) to emulate, and may require you to implement a totally different (but more appropriate) solution.

Following is your migration task list...

Getting Started

- **Transfer your workspaces** from APL2 to APL/W. Dyadic supplies a workspace `WDYALOG\WS\APL2IN` that helps to read APL2 transfer files created by *)OUT*. I wrote an enhanced version which Dyadic distributes in workspace `WDYALOG\OUTPRODS\TOOLS\APL2IN2`.

- Set APL/W's **migration level** with ⎕*ML*←3. This will immediately solve several compatibility issues:

Z←∈*R*	Enlist (not Type)
Z←↑*R*	First (not Mix)
Z←⊃*R*	Mix (not First)

$Z\leftarrow\equiv R$	Absolute value of depth
$Z\leftarrow L\subset R$	Partitioned enclose
$Z\leftarrow\square TC$	Order of terminal control characters

- Check for dependence on any **APL2 invocation options**, especially:

`DATEFORM`	Format for timestamps
`DEBUG`	Suppress $\square LX$, etc.
`INPUT`	Queue input strings
`QUIET`	Suppress output
`RUN`	Auto-invoke function via $\square NA$
`TERMCODE(-1)`	Controlled invocation

Character Sets

- Beware of the **atomic vector** (any reference to $\square AV$)! The order of characters is *very* different.

- Remember that the **EBCDIC and ANSI** character sets are quite different. Even characters that seem straightforward may not be — single quotes, vertical bars, and exclamation points should be examined carefully. Watch out for national language characters and currency symbols.

  Also remember that characters read from PC files into APL/W may pass through Dyalog's `APLT=WIN.DOT` translation mechanism. So you really have at least three character sets to worry about: EBCDIC, ANSI and $\square AV$.

- Watch out for four **overstruck APL2 symbols** not present in APL/W's $\square AV$:

  ⌸ ⍁ ⍦ ⍠

  Luckily, they don't actually do anything in APL2, so they probably won't matter much.

Different Syntax

- Wrap braces around **optional left arguments**. For example, if in APL2 you had:

```
      ∇ R←A FOO B
[1]     →(0≠⎕NC 'A')⍴L1   ⍝ Left arg supplied?
[2]     A←⍳0              ⍝ Default left arg
[3]    L1:
```

  In APL/W you need to change to header to:

```
      ∇ R←{A}FOO B
```

I was able to automatically convert 99% of these cases by writing a workspace searching tool that inspects all dyadic functions for ⎕NC 'leftarg' and makes the appropriate change in the header.

- You cannot **assign "read-only"system variables** in APL/W. So if your APL2 functions use them as a sink (e.g., ⎕WA←) you will have to convert. I never liked this technique anyway; I always define a "no-op" function called SINK:

```
     ∇ SINK A
[1]    ⍝ Throw away argument
     ∇
```

Then globally change all '⎕WA←' and '⎕TS←' and etc. to 'SINK '.

Or, you may be able to exploit APL/W's "shy" explicit result feature — if you find ⎕WA←FOO X, you can change FOO's header to ∇{R}←FOO A and then just execute FOO X.

- **Trace and stop** controls are different. Use APL/W's ⎕TRACE 'FOO' and ⎕STOP 'FOO' rather than APL2's TΔFOO← and SΔFOO←. (Note that APL/W also has a much fancier interactive trace facility; see Trace on the Action menu.)

Same Syntax, Limited Capability

- Some complex forms of **selective assignment** are not permitted in APL/W. For example, with a 3 by 4 matrix M, something like:

```
(1 0 1/M[;4])←0
```

works in APL2 but generates a DOMAIN ERROR in APL/W. You will have to re-code this as:

```
Q←M[;4] ⋄ (1 0 1/Q)←0 ⋄ M[;4]←Q
```

Similarly, (A⊃B)[I]←X does not work in APL/W.

- Some APL/W primitive functions generate errors when **reduction is applied to an empty array**. For example, in the case of , / ⍳0 APL2 returns ⊂⍳0 whereas APL/W signals a DOMAIN ERROR.

- Does your code use ⎕DL to **delay**? APL/W does not permit fractional arguments, so something like ⎕DL 1.5 will generate a DOMAIN ERROR.

Same Syntax, But Works Differently

- Watch out for the **rank of the result of some system functions.** In APL2, `⎕NC 'A'` returns a scalar, but APL/W returns a one-element vector. This can escalate into a depth problem if you execute `⎕NC⍤¯1⊢'A'`. Monadic `⎕SVO` also exhibits this behaviour.

- **Compress-each** is interpreted differently. For example, in APL2:

```
      1 0 1/⍤¯1⊢V←4⍴⊂⍳3
1 3  1 3  1 3  1 3
```

  With APL/W, you will need to enclose the compression vector:

```
      1 0 1/⍤¯1⊢V
LENGTH ERROR
      (⊂1 0 1)/⍤¯1⊢V
1 3  1 3  1 3  1 3
```

- **Strand indexing** is interpreted differently. For instance:

```
X Y Z[I]
```

  In APL2, the `I` indexes just `Z`, but in APL/W, the `I` indexes the three-item vector `(X Y Z)`. So for APL/W you will have to convert this to:

```
X Y (Z[I])
```

- **Strand assignment** of an enclosed scalar is treated differently:

```
(X Y)←⊂'ABC'
```

  is treated as:

```
X←Y←'ABC'    ⍝ APL2
X←Y←⊂'ABC'   ⍝ APL/W
```

- APL/W's **name class** does not like system functions and variables. In APL2, `⎕NC '⎕AV'` reports `2` and `⎕NC '⎕CR'` reports `3`, whereas APL/W reports `¯1` in both cases.

  Also, in APL/W it is possible for `⎕NC` to return a `9`. So watch out if you have tools that do something like `Z←'IULVFO'[¯1 0 1 2 3 4⍳⎕NC A]`.

- Beware of **applying a user-defined function to an empty array with each.** In APL2, if you execute `Z←FOO⍤⍳0`, `FOO` is never actually executed, and the result `Z` is based on the prototype of the `FOO`'s argument. In APL/W, `FOO` is

executed once with an argument based on the prototype of *FOO*'s argument, and the result Z is based on the prototype of *FOO*'s result *R*.

```
      ∇ R←FOO A
[1]     ⎕←'Arg is:' A
[2]     R←2 4 6
      ∇

      DISPLAY Z←FOO¨⍳0   ⍝ APL/W
 Arg is:  0
.⊖--------.
| .→----. |
| |0 0 0| |
| '~----' |
'∈--------'

      DISPLAY Z←FOO¨⍳0   ⍝ APL2
.⊖.
|0|
'~'
```

This is a nasty one to detect in advance. Your function may not be prepared to handle a zero or empty argument.

- When **monadic format** is applied to a nested array, the spacing of the result is sometimes different. I first noticed this in mainframe code that composed short messages with format, such as (using ∘ to represent blanks):

```
      ⎕←↑'FOUND' 9 'DOCUMENTS'
∘FOUND∘∘9∘∘DOCUMENTS∘      ⍝ APL/W
∘FOUND∘9∘DOCUMENTS∘        ⍝ APL2
```

There are also subtle differences when some more complex nested arrays are formatted. So if your code relies on a certain number of blanks, beware.

Emulation (or Re-Coding) Required

- **Format-by-example** is not available in APL/W. So every time you see something like `'550.03333%'⍕A` you have some work to do! (And don't forget that in APL2, `⎕FC[1]` might be sneaky and change the decimal point to some other character!)

- The **index function** ⌷ (sometimes called "squish-quad") is not supported in APL/W. Dyalog APL does have plenty of ways to do indexing, so conversion should not be too much of a problem.

- APL/W does not support **n-wise reduction** (as in `2+/A`). The 2-line user-defined operator *NWISE* that Dyadic supplies in workspace `WDYALOG\WS\OPS` can help to emulate this feature. I wrote a more complete version that is also somewhat less prone to *WS FULL*.[1]

- **Scalar functions over an axis** are not supported in APL/W, so if your APL2 code does things like *MATRIX*+[2]*VECTOR* you will have to change them. I use a user-defined operator called *AXIS*.[1]

- Write emulation functions for missing APL2 **system functions**, as required:

⎕*AF*	Atomic Function
⎕*AT*	Attributes[2]
⎕*EA*	Execute Alternate
⎕*EC*	Execute Controlled
⎕*ES*	Event Simulation
⎕*FX*	Fix (dyadic: with execution properties)
⎕*TF*	Transfer Format
⎕*UCS*	Universal Character Set

- Write emulation functions for missing APL2 **system variables**, as required:

⎕*EM*	Event Message
⎕*ET*	Event Type
⎕*FC*	Format Control
⎕*L*	Left argument
⎕*NLT*	National Language Translation
⎕*PR*	Prompt Replacement
⎕*R*	Right argument
⎕*TZ*	Time Zone
⎕*UL*	User Load

- **Name associate** is very different. Write emulation functions for anything involving ⎕*NA*, including:

`10 ⎕NA`	REXX
`11 ⎕NA`	Access external routines
`12 ⎕NA`	Files as arrays

- Write emulation functions for APL2 **external supplied routines**, including:

∆FM, ∆FV, ∆F	File read/write/query
∆EXEC	Execute REXX
ATR, CTN, CAN, DAN	Data conversion routines
EXP, PACKAGE, QNS	Namespaces and name scopes[3]
IN, OUT	Like)IN and)OUT
EDITOR2, EDITORX	Interface to editors

Development Environment

- The APL/W **session manager** is totally different. Inspect your workspaces for use of APL2's AP120.

- The APL/W **function editor** is totally different, so it's time for you to learn yet another editor. (At least it supports cut and paste!)

System Commands

- Note that APL/W does not support some APL2 **system commands:**

)*CHECK*	)*EDITOR*	)*HOST*	)*IN*
)*MCOPY*	)*MORE*	)*NMS*	)*OUT*
)*PBS*	)*PIN*	)*QUOTA*	)*SIC*
)*SIS*	)*SYMBOLS*		

- Also note that some APL2 **system command arguments** are not supported in APL/W:

)*FNS*,)*VARS*, and)*OPS* do not accept a range of names
)*RESET* does not allow an argument

- Are you using APL2 utilities that **execute system commands via the stack** and capture their "results"? If so, you'll need a different technique. APL/W generally makes this easy — it has many more matching system functions that return explicit results (like ⎕*WSID*, which is inexplicably missing in APL2).

External Communication

- Anything involving **shared variables** will probably need conversion.

- Anything involving **auxiliary processors** will probably need conversion.

The Operating System

- Don't forget all **operating system dependent non-APL facilities,** such as:

  The CMS stack
  System command language (REXX)
  System editor (XEDIT)
  Document composition (DCF/Script)
  CMS Pipelines
  System file I/O (flat files, etc.)
  Other data access methods (VSAM, etc.)
  Database (SQL), Graphics (GDDM)
  VM Backup/Archive,VM Schedule

Network Communications

- Review your application's use of your **mainframe communications network.** Will remote users be able to access the new system through the same network? Can you dial in from home? Are you sending jobs to an MVS system?

User Interface

- And, last but not least, there's the **user interface!** The procedural vs. event-driven issue is a huge topic in its own right. Suffice to say that unless you want to do extensive re-coding, you will probably be forced to make some compromises here.

Conclusion

This list is, of course, not necessarily complete — these are just the problems I have run into so far, and I'm sure there are more lurking.

It would be a mistake to come away from reading this with the impression that Dyalog APL/W is missing a lot of APL features — in fact, it's just missing some *APL2* features, and it has many compensating and additional features that APL2 does not have. I mean for all this to be free of value judgments — I'll leave it to you to decide whether one system does things "better" than the other.

Rex Swain
Independent Consultant
8 South Street
Washington, CT 06793
USA

Tel: (+1) 860-868-0131
Fax: (+1) 860-868-9970
Email: rhswain@acm.org
WWW: http://www.pcnet.com/~rhswain

---

1 The user-defined operators *NWISE* and *AXIS* may be downloaded from the "Dyalog APL/W Tools and Utilities" section of my WWW home page.

2 Dyadic is adding ⎕*AT* to APL/W versions 7.2 and 8, so function timestamps should be available by the time you read this.

3 For more details, see my paper "Namespaces: APL/W vs. APL2" in the APL95 conference proceedings.

Multiprecision Arithmetic: Part IV

by John Sullivan (jos@scuk.demon.co.uk)

Applications of Multiprecision Arithmetic

In parts 1 & 2 of this series, I introduced the basic functions for performing multiprecision arithmetic. But why do we need to perform arithmetic to such precision? The day-to-day operations of your bank account, or even of the world's money markets, require no more precision than that supplied by APL for its ordinary arithmetic operations. Scientific measurements no matter how small or large are usually accompanied by some (small or large) error, so accuracy to more than 4 significant figures is usually unnecessary.

In an article in Scientific American[1], Fred Gruenberger points out that in using arithmetic with only a small number of significant digits, repeated squaring of a number only slightly greater than 1 will soon result in inaccurate results. He then gives the result of squaring 1.0000001 seven times (equivalent to raising it to the power of 128), which results in 1.0000128000812803413... (896 decimal places). Try it with the multiprecision functions. Even on my 486/33 the calculation is quick. Set `⎕PP` to 15 and enter `1.0000001*128`, and you will see the problem. The result, 1.00001280008129, is wrong in the last displayed place. And as the power gets larger the results become less accurate. Don't think that this is a silly exercise: some problems involve iterated multiplication like this; can you *really* trust the results your computer gives you?

Next, Gruenberger discusses the continuing search for larger prime numbers. Before the development of computers, the largest known prime was $2^{127} - 1$, which has only 39 digits. This activity requires the multiprecision arithmetic of parts 1 and 2 of this series, and the primality testing and factoring of large numbers from part 3. But, having discovered a few 100-digit primes, and found the factors of a few 100-digit non-primes, you may be wondering what sort of practical application there is for these numbers.

Gruenberger mentions some other problems which require high precision arithmetic, such as the solution of simultaneous equations where the determinant of the associated matrix is close to zero, and a couple of mathematical recreations which lead to very large numbers. However, he makes no mention of what is considered by many to be the only commercial application of number theory in the modern world: data encryption and decryption.

This algorithm for public-key cryptography was described in the late 1970s by Rivest, Shamir and Adleman[2], and it is called the RSA system after them. Although all of the number theory behind it has been known for the last couple of centuries, it is the advent of the digital computer that has made the calculations practical. Two numbers, the encryption key e and the modulus n are published, and anybody can use them to encode a message, which they then send to you. The idea is that n cannot be factored without a lot of effort, so the code-breaker cannot easily calculate the encryption key d to decipher the coded message. There are many books and articles on the RSA algorithm: I obtained my inspiration from chapter 6 of Riesel[3], although I have not used his examples.

Some number theory

Before we can go merrily encrypting and decrypting our secret text, we need some number theory. I am not going to prove any of the following statements: look in any good textbook on number theory, such as those mentioned at the end of part 3. These statements are the key to what follows. All of the numbers concerned are, of course, integers.

If GCD(a,n)=1 then there exists a number b with $1 \le b < n$ such that $ab \equiv 1 \bmod n$. For example $2 \times 4 \equiv 1 \bmod 7$. We call b the multiplicative inverse of a modulo n.

For any integer a with GCD(a,n)=1, $a^{\phi(n)} \equiv 1 \bmod n$, where $\phi(n)$ is the number of positive integers not exceeding n that are relatively prime to n ($\phi(n)$ is known as Euler's totient function). Not only that, but if n is squarefree $a^{k\phi(n)+1} \equiv a \bmod n$ for all integers a and k. If n is prime then $\phi(n) = n - 1$. If GCD(b,c)=1 then $\phi(bc) = \phi(b)\phi(c)$. However, thanks to a theorem due to Carmichael we can do better than this. There is a function, called Carmichael's function, denoted by $\lambda(n)$, such that $\lambda(n)$ divides $\phi(n)$, and $a^{\lambda(n)} \equiv 1 \bmod n$, with GCD($a$,$n$)=1 and $a^{k\lambda(n)+1} \equiv a \bmod n$ for all integers a and k and squarefree n. In general, to calculate Carmichael's function for a composite number one requires all the prime factors of its argument, which is a beyond the scope of this article; all we need here is that if p and q are primes then $\lambda(pq) = \mathrm{LCM}(p - 1, q - 1)$.

The algorithm

The algorithm for what we are about to do is simple. As mentioned above, we need three numbers: the encrypting key, the decrypting key, and the modulus. We represent our plain text as a number by some means, raise it to the power of the encrypting key modulo the modulus, and send the result to the recipient of our secret message. The recipient then raises this number to the power of the decrypting key modulo the modulus, and the result of this is the original

message. To make this work, we choose two primes p and q, and multiply them together to get the modulus n. Choose an encrypting key e such that it is relatively prime to $\lambda(n)$. Then the decrypting key d is just the multiplicative inverse of e modulo $\lambda(n)$. (Some books suggest using $\phi(n)$ here, instead of $\lambda(n)$, but this just means raising to a larger power and doing more work for the same result.) Effectively, since $de \equiv 1 \bmod \lambda(n)$, then $de = k\lambda(n)$, for some k, and we are raising our original number to the power of $k\lambda(n) + 1 \bmod n$, which, as mentioned above, restores the original number.

The multiplicative inverse function

This is put in namespace *mp* along with the other multiprecision building blocks. It is based on the Euclidean Algorithm: if GCD(a,b)=1 then there exist m,n such that $am + bn = 1$. This can be rearranged as $am = 1 - bn$, which immediately gives $am \equiv 1 \bmod n$, thus m is the multiplicative inverse of a modulo n. If the GCD is not 1 then this function returns 0, which is an impossible value for the inverse.

As with the GCD function (see part 3) this function is divided into two paths, depending on whether we are processing scalars or multiprecision numbers. The first thing we do is find the GCD of the input (I use x here to represent a, and n is the modulus). The Euclidean algorithm says that when we start with a and b, we obtain the following system of equations:

$$\begin{aligned} a &= q_1 b + r_1,\ 0 < r_1 < b \\ b &= q_2 r_1 + r_2,\ 0 < r_2 < r_1 \\ r_1 &= q_3 r_2 + r_3,\ 0 < r_3 < r_2 \\ &\cdots \\ r_{n-2} &= q_n r_{n-1} + r_n,\ 0 < r_n < r_{n-1} \\ r_{n-1} &= q_{n+1} r_n + 0 \end{aligned}$$

We start off by generating all of these quotients and remainders, storing the quotients in vector a:

```
      ∇ z←x Inv_mod n;a;b;N;rn;r0;r1
[1]     a←0 ⋄ N←n
[2]     →(1∨.≠(⍴x),⍴n)/Δ10
[3]     z←0 ⋄ x←N|x
[4]    Δ1:a←a,⊃b←(0,x)⊤n
[5]     n←x ⋄ →(0≠x←1⊃b)/Δ1
```

At this point n holds the GCD of the original n and x. If this is not 1 then there is no multiplicative inverse

```
[6]     →(1≠n)/0
```

Now we have to go back recursively. We calculate r_n in terms of $r_0 (= b)$ and r_1. The recurrence relation we calculate is

$$r_i = r_{i-2} - q_i r_{i-1}$$

```
[7]     r0←1 0 ⋄ r1←0 1
[8]    Δ3:rn←r0-r1×⊃a
[9]     r0←r1 ⋄ r1←rn ⋄ a←1↓a
[10]    →(1<⍴a)/Δ3
```

And our result is the coefficient of r_1 modulo n.

```
[11]    z←N|1⊃rn ⋄ →0
```

And now for the multiprecision version. This part of the function is the same as the scalar version, with minor differences to allow for the way the multiprecision functions have been coded.

```
[12]   Δ10:z←0 0 ⋄ x←x Imod N
[13]   Δ11:a←a,(b←n Idiv x)[0]
[14]    n←x ⋄ →(~0 0 Fequal x←1⊃b)/Δ11
[15]    →(~0 1 Fequal n)/0
[16]    r0←(0 1)(0 0) ⋄ r1←(0 0)(0 1)
[17]   Δ13:rn←r0 Fsub¨r1 Fmul¨a[0]
[18]    r0←r1 ⋄ r1←rn ⋄ a←1↓a
[19]    →(1<⍴a)/Δ13
[20]    z←(1⊃rn)Imod N
      ∇
```

Testing the Method

For demonstration purposes it is not necessary to search out abstruse prime numbers. At the back of Riesel[3] there are tables of large prime numbers of various types, and I have chosen some of those. You will notice that when using the factoring function I showed in part 3, the factors of n fall out in no time at all. This is O.K. for demonstrations, but is not supposed to happen in reality!

$p = 9 \times 2^{63} - 1 = 83010348331692982271$
$q = 9 \times 2^{63} + 1 = 83010348331692982273$
$e = 27 \times 2^{70} - 1 = 31875973759370105192447$

Since p and q are primes, $\lambda(pq)$ is just LCM($p - 1$, $q - 1$). From this we can calculate the decrypting key:

$d = 25918802123181064157039718138387250544 63$

I created another namespace to hide these away, called `rsa`. It seemed a good idea to use a setup function to generate these numbers:

```
      ∇ setup;d;e;n;p;q;s
[1]     s←⎕CS'#.mp'
[2]     p←0 ¯1 Fadd 0 9 Fmul 0 2 Fspow 63
[3]     q←p Fadd 0 2
[4]     n←p Fmul q
[5]     e←0 ¯1 Fadd 0 27 Fmul 0 2 Fspow 70
[6]     d←e Inv_mod(0 ¯1 Fadd p)Lcm 0 ¯1 Fadd q
[7]     ⎕CS s
[8]     D←d ⋄ E←e ⋄ N←n ⋄ P←p ⋄ Q←q
      ∇
```

The LCM function (in namespace *mp*) is very simple. As with other functions it has two paths, one for scalars and one for multiprecision numbers.

```
       ∇ z←a Lcm b
[1]     →(1∧.=(⍴a),⍴b)/Δ1
[2]     z←⊃(a Fmul b)Idiv a Gcd b
[3]     →0
[4]    Δ1:z←(a×b)÷a Gcd b
       ∇
```

In order to test the method, I made a function called `test`. It is sufficient to use a single number, smaller than *n*, to generate an encrypted version, and to decrypt it again. If the decrypted version equals the original version then the process works.

```
      ∇ test;M;C
[1]     M←#.mp.Fexec'12345678901234567890123456789012345678901234567890'
[2]     'Original  Message is ',0 #.mp.Ffmt M
[3]     'Encrypted Message is ',0 #.mp.Ffmt C←M #.mp.Impow E N
[4]     'Decrypted Message is ',0 #.mp.Ffmt C #.mp.Impow D N
      ∇
```

And the following output should surprise nobody.

```
       test
Original  Message is 12345678901234567890123456789012345678901234567890
Encrypted Message is 631819554104836268192893066856843989987 6
Decrypted Message is 12345678901234567890123456789012345678901234567890
```

Practical Matters

How you implement this algorithm is very much up to you and your correspondents. Here is one method that I devised for testing purposes. Again, this is different from the examples in Riesel.

First of all we determine what characters we are going to allow in our messages. I added this line to the end of the *setup* function, above.

```
Alf←' ',⎕D,⎕A,'.,:;!?()'
```

Now you need your normal capitalization function, something like this

```
      ∇ x←Caps x;a
[1]     a←⎕AV ⋄ a[⎕AV⍳'abcdefghijklmnopqrstuvwxyz']←⎕A
[2]     x←a[⎕AV⍳x]
      ∇
```

Convert your plaintext to caps, and replace characters not in the "alphabet" with blanks. Generate a set of origin-zero indexes into this alphabet, and treat them as a radix-⍴*Alf* multiprecision number by catenating a zero at the start (this is a beneficial side-effect of the variable-radix design of the multiprecision suite). Since the first character in our alphabet is a blank this has the effect of stripping leading blanks from the message, which is probably a good idea. Convert this number to a multiprecision number in our default radix, then use encode (⍤) on it to ensure that we break it into parts that are not greater than the modulus *n* of our encryption algorithm. Each of the resulting numbers is then encrypted, and the reverse process is carried out on the result, although this time we just use an alphabet of the usual capital letters. Then we split the output into the 5-character blocks, so beloved of espionage systems in the 30's to early 50's.

```
      ∇ z←Encrypt x;a;b
[1]     x←,Caps x
[2]     x[(~x∈Alf)/⍳⍴x]←' '
[3]     z←(#.mp.base,⍴Alf)#.mp.chbase 0,Alf⍳x
[4]     z←N #.mp.enc z
[5]     z←z #.mp.Impow¨⊂E N
[6]     z←N #.mp.dec z
[7]     z←1↓(26,#.mp.base)#.mp.chbase z
[8]     z←⎕A[z]
[9]     z←(5×b←⌈0.2×⍴z)↑z
[10]    z←((6×b)⍴1 1 1 1 1 0)\z
      ∇
```

The decryption process is just the reverse of this. First we eliminate all non-alphabetical characters from the ciphertext, then we compare it with the ciphertext without the blanks. Obviously, if these are not the same we are working on the wrong text, so we signal an error. Then the process continues as for *Encrypt*, except that we do not format in 5-character blocks at the end.

```
      ∇ z←Decrypt x;a;b
[1]     z←x∩⎕A
[2]     ⎕SIGNAL(z≠x~' ')/11
[3]     z←(#.mp.base,26)#.mp.chbase 0,⎕A⍳z
[4]     z←N #.mp.enc z
[5]     z←z #.mp.Impow¨⊂D N
[6]     z←N #.mp.dec z
[7]     z←1↓((⍴Alf),#.mp.base)#.mp.chbase z
[8]     z←Alf[z]
      ∇
```

And finally, here are the functions *dec* and *enc* from namespace *mp*. *dec* is similar to the primitive function Decode (⊥) except that the left argument is a single multiprecision number, and the right argument is a vector of multiprecision numbers (so it must be enclosed if there is only one). *enc* is similar to the primitive function Encode (⊤) except that it only works on one multiprecision number at a time, the left argument is a single multiprecision number which is assumed to repeat as often as required.

```
      ∇ z←b dec x
[1]     z←0 0
[2]    Δ1:z←(⊃x)Fadd b Fmul z
[3]     →(0≠⍴x←1↓x)/Δ1
      ∇

      ∇ z←b enc x
[1]     z←⍬
[2]    Δ1:→(b Fgt x)/Δ2
[3]     x←x Idiv b
[4]     z←x[1],z
[5]     x←⊃x
[6]     →Δ1
[7]    Δ2:z←(⊂x),z
      ∇
```

Exercises

You haven't got off scot-free: now it's your turn to do some work. Using the values of the parameters and the encryption and decryption functions above, decipher the following message.

```
LFWBQ CXLSF EBSPQ JNROT MPAKI VWHOG URVYB DHGFY LOQBH JXCOR
YVFHL ZGWRT XYPRB GCKUT BOPHN ILFHQ MZSOM OHMJO ILEXK XIVSN
QZXLC JGKED ZKWTR SCHXU GNPHV MYGGE VUXCP EAGTB M
```

Finally

I have enjoyed writing this series of articles, and I hope you have enjoyed reading them. I am always interested in hearing about multiprecision arithmetic and its applications, and if you wish to correspond you can get hold of me through the editor, or via email on jos@scuk.demon.co.uk.

The workspace that I used for the examples in this series is available by one of the following methods:

> If you are on the Internet you can get a vendor-independent version in Jim Weigang's APLASCII format by anonymous ftp from archive.uwaterloo.ca, called /languages/apl/workspaces/mult.prec/sullivan/MP
>
> Otherwise send me a blank diskette (you can get hold of me via the Editor). You can either have MP as above, or a Dyalog APL/W 7.1.2 workspace which contains my originals and a lot more.

References

[1] Fred Gruenberger, *How to handle numbers with thousands of digits, and why one might want to.* "Computer Recreations", Scientific American, April 1984.

[2] R L Rivest, A Shamir & L Adleman, *A method of obtaining digital signatures and public-key cryptosystems.* Communications of the ACM, Vol. 21 No. 2, 1978

[3] Hans Riesel, *Prime numbers and computer methods for factorization* (Birkhäuser, 1985).

Not specifically referred to in the text, I have quoted from and made use of some of the items in Peter Merritt's articles in Vector, Vol. 11 No. 1 p.108, and Vol. 11 No. 3 p.119.

In addition to Riesel, there are some other books that contain detailed descriptions of the RSA algorithm, amongst which are the following:

C Bondi (editor), *New Applications of Mathematics*, Penguin Books, 1991

H Beker & F Piper, *Cipher Systems, the Protection of Communications*, Northwood Books, London, 1982.

D E Knuth, *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*, Addison-Wesley, 2nd edition, 1981

THE RANDOM VECTOR

Polynomial Multiplication with Circulant Matrices: Insights Using APL

by Lew Robinson
frgp21a@prodigy.com

A "Hook" to Get your Attention

Here is a puzzle. Show how to use nothing to make something out of nothing. Impossible? Read on and at the end of this article learn how to do it with some plain old APL.

In his book *"APL Programs for the Mathematics Classroom"*, Norman Thomson gives the following APL one-liner to accomplish the multiplication of two polynomials,

Thomson algorithm: `1↓+/⍉(-⍳⍴L)⌽L∘.×R,L≠L`

Two minor problems occur with Thomson's code if either or both arguments are simple scalars or if `⎕io` is zero. These are easily corrected however with judicious use of commas and parentheses, thus, Thomson algorithm, modified,

```
TAMP: 1↓+/⍉((1-⎕io)-⍳⍴,L)⌽(,L)∘.×R,L≠L
```

TAMP stands for (T)homson (A)lgorithm for (M)ultiplying (P)olynomials. (As modified — A further modification might be to replace `⍉` with `⌿`.) Here in APL is a different algorithm to multiply two polynomials,

Circulant algorithm,

```
CAMP: ¯1↓L+.×(⎕io-⍳⍴,L)⌽((⍴,L),⍴R,L)⍴R,L≠L.
```

CAMP stands for (C)irculant (A)lgorithm for (M)ultiplying (P)olynomials. Note that to test these programs one must interpret L and R as the vectors of coefficients of two (different or the same) polynomials in ascending power order.

A two-line dyadic APL function named *TAMP* that incorporates the Thomson approach might have generic form and usage as follows,

```
P←L TAMP R.
```

Example usage: `P←1 0 1 TAMP 1 2 0 3`

A two-line APL function named *CAMP* that utilizes the Circulant algorithm would have the same generic form and usage, except only that the name of the function changes to *CAMP*. Both functions would return the same result in *P*, the list `1 2 1 5 0 3`. The list is exactly the coefficients of the polynomial product in ascending power order, so that the last item in the list multiplies the highest power term of the result, etc.

Thus letting *X* stand for the indeterminate, the highest power term in the product polynomial is `3×X*5`, the next highest is `0×X*4`, the next is `5×X*3`, the next is `1×X*2`, the next is `2××X` and the lowest power term is `1×X*0`, the constant term, `1`.

Circulant Matrices

Circulant matrices have a key property: write down the first row and you know the entire matrix. The second row is a clone of the first, except all members have been shifted one position to the right. The last element is simply "wrapped-around" into the first position. Obtain every succeeding row from its immediate predecessor by repeating the "shift and wrap" operation. The shifted clones all contain the same numbers but in different columns.

Suppose someone gives us a linear list of numbers. Then clearly one can obtain a circulant matrix from the list. The following is a useful shorthand notation for that operation; denote a circulant matrix as: *circ(list) = circ(c1,c2, . . . ,cn)*

Alternatively let the vector *v* = c1, c2, . . . , cn. Then for any vector *v*, circ(*v*;*n*) denotes the corresponding *n* by *n* circulant matrix. A precise definition of a circulant matrix is required at this point. In APL notation, the *I* , *J* element of the *N* square, possibly complex matrix *C* is `C[I;J]`.

Now form: `K ← N | (J-I) + 1` Consider `T ← C[I;J] = C[1;K]`

If boolean *T* is true for all *I* and *J* from `1` to *N*, then *C* is a circulant matrix. Note that the dyadic use of stile is the APL residue function, in this case the right argument modulo the left argument *N*. Note also that `C[1;K]` is just the element in the Kth position in the first row.

Time-out for a Discussion about Jargon

Previously X was called an indeterminate and not a variable. This choice was deliberate and the distinction is useful to algebraists. For those APLers who remember their studies in abstract algebra, the indeterminate X is fraught with non-meaning. The abstract algebraist considers it to be the "universal abstraction" — sort of a surrogate place holder that assumes whatever non-role would be useful to generalize the results of studying a particular algebraic structure. (If this sounds confusing, it is. Fortunately there are approaches to abstract algebra that dispense altogether with such a notion about X. It is no longer an indeterminate; instead it takes on concrete meaning in a formal definition of a polynomial. See for example page 127 of the text *"Applied Abstract Algebra"* by R. Lidl and G. Pilz.)

Undoubtedly some APLers either have never encountered abstract algebra or have been sufficiently traumatized by its study that they wish to forget the experience! Please do not be put off by references to the concepts and jargon of the discipline in what follows. If you wish, just ignore or skip over comments about "indeterminates", "rings", "ideals", "integral domains", "fields", "isomorphisms", etc. However the reader will certainly benefit from remembering several things taught in high school algebra. These include multiplication of matrices and multiplication and long division of polynomials. In any case, the listings of APL functions and their usage should be familiar and, I hope, interesting. Fortunately, given the array-like nature of the language, APLers know how to multiply two matrices with the "+.×" operator, even though they perhaps never studied matrix or linear algebra.

Back to the Discussion of Polynomial Multiplication with Functions *TAMP* and *CAMP*

Function *TAMP* has taken one polynomial with highest power $3\times X*3$ and has multiplied it with another polynomial of highest power $X*2$. Function *CAMP* has done the same thing. Both give the same the result, a product polynomial with highest power term $3\times X*5$. Further, as required, the product list contains the correct coefficients for the remaining powers. The APL expressions

```
⍳⍴P   (⎕io←0)                (⍳⍴P)-1   (⎕io←1),
```

give the ascending power order of the powers for each coefficient. What is going on here? Why does the Thomson program work? Why does the Circulant algorithm work?

In his book *"ALGEBRA, An Algorithmic Treatment"*, Kenneth Iverson explains the "inner workings and hidden mechanisms" of program *TAMP*. He asks the reader to consider two example tables,

```
T1←1 0 1 ∘.× 1 2 0 3
```
(Use the left and right arguments from the earlier example for function *TAMP*.)

```
T2←0 1 2 ∘.+ 0 1 2 4
```
(Note the "plus" sign. Exponents add.).

The results are

```
      1 2 0 3                    0 1 2 3
T1    0 0 0 0        and   T2    1 2 3 4
      1 2 0 3                    2 3 4 5
```

All the counter-diagonals of table *T2* are constant. Read from upper left to lower right, the constants increase from 0 to 5. This is exactly the order required for the ascending powers of the result. Scan and add along a fixed counter-diagonal in table *T1*. Verify that the sum gives the desired coefficient for the "power" found on the corresponding counter-diagonal in table *T2*. The idea behind program *TAMP* now stands revealed. (A caution — the preceding argument is a heuristic one; it is not a proof and may leave room for doubt. For a formal proof that function *TAMP* produces the polynomial product, see Iverson's book and his convincing usage of APL notation to achieve the result.)

Can something similar be going on with the circulant approach? Notice that the primitive dyadic function "rotate" (⌽) is centrally embedded in both *TAMP* and *CAMP*. Check this out as follows. Use as inputs the previous example's arguments (left and right). Stop both *TAMP* and *CAMP* just after the completion of dyadic ⌽ to get these results,

```
        0 1 2 0 3 0 0                  1 2 0 3 0 0 0
TAMP    0 0 0 0 0 0 0          CAMP    0 1 2 0 3 0 0
        0 0 0 1 2 0 3                  0 0 1 2 0 3 0
```

No clear overall pattern is apparent for table *TAMP*, although summing down the its columns gives the polynomial coefficients (except for an extraneous 0).

In contrast, a very useful pattern has begun to emerge for the *CAMP* result; it looks like the first three rows of a 7 by 7 circulant matrix. But where are the polynomial coefficients? They appear when the left argument, vector `L←1 0 1`, multiplies table *CAMP*, considered as a matrix, on the left. Carry out the vector-matrix multiplication specified for the completion of algorithm CAMP and obtain the correct result. (Ignore the extraneous 0 that appends itself to the end.)

Warning: Detour into Abstract Algebra, this Section Skippable

Is the apparent pattern in table *CAMP* a clue to the existence of a more general result? Given the title of this article, the reader will not be surprised to learn that the answer is an unqualified "yes". In the language of abstract algebra, a certain isomorphism exists. Here is the relationship in all the "terminology-as-jargon" words of abstract algebra.

> "The commutative ring of *n* by *n* circulant matrices with entries in the complex field is structurally the same thing as the commutative ring of polynomials with complex coefficients, modulo the ideal generated by the indeterminate expression (*X*n* - 1). This latter is usually written as *K*[X]/(X*n* - 1)"

Stated more briefly, the ring of circulants is isomorphic to the polynomial 'residue class' ring modulo the principal ideal (*X*n* - 1).

Here the notation *X*n* means the indeterminate raised to the *n*th power. Presumably the practitioners of abstract algebra know how to multiply together *n* times the universal abstraction embodied in *X*. While one might consider that such a talent is more theological than algebraic, algebraists can define away the need for a "leap of faith" that such exponentiation is possible. They do so by appealing to linear algebra and invoking the notions of a complex vector space spanned by a prescribed basis set.

The basis is taken to be the infinitely many elements *X*n* where *n* ranges over the non-negative integers. The symbol *n* at this point has lost its property of being an exponent. Rather it merely acts to index the basis elements. As the development proceeds, *n* manages to re-acquire its exponential character. This is achieved by making the product of two vectors meet the specifications of a bilinear mapping and defining the value of an ordered pair *X*m*, *X*n* to be *X*(m+n)*. The details are a diversion, but the result is that the index *n* ends up acting like an exponent again. Some may find this use of definition to be egregious and the whole business rather like a 'self-fulfilling prophecy'.

In any case the mystery can be avoided because all that really matters are the polynomial coefficients and *n*, the number of times *X* multiplies itself. However mysterious those multiplications of *X* might be, no details are needed, other than *n*, the number of factors involved. Why talk about the "ring" of circulants instead of an "integral domain" or "field"? Because in general "divisors of zero" exist for circulant matrices. This abstract algebra "jargon-phrase" means only that the cancellation law of multiplication does not hold. Later, an example is provided where the product of two non-zero circulants is the zero matrix.

Can the phrase "...modulo the ideal generated by the indeterminate expression (X*n - 1)" have some tangible implementation in the world of APL functions? To put some operational meaning to the words, first multiply two polynomials using either *CAMP* or *TAMP*. Then multiply the same two polynomials using *CAMS*. (*CAMS* is a one-line APL function to multiply two circulants. It is described in a later section.) But before using *CAMS*, if one list of coefficients is shorter than the other, make the matrix multiplication conformable. To do this, pad the shorter coefficient list on the right with as many zeros as it takes to get exactly as many total entries as in the longer list. Take *n* to be the length of either list, now both of the same length.

The first result, using *CAMP* or *TAMP*, is the product of two polynomials. The second result, using *CAMS*, is the product of two circulant matrices. Except possibly for zero padding, in both results the coefficient lists used as input are identical.

Next generate a coefficient list corresponding to X*n - 1. If for example, *n* is 10, the list will have 11 entries. All items in the list will be zero except the first and the last. The first will be -1 and the last will be 1. This list is just the coefficients of the polynomial X*10 - 1, with zeros in place of all the intermediate missing monomial terms.

Now divide the *CAMP/TAMP* result for the product of two polynomials by the polynomial X*10 - 1. Do this with an APL function that implements polynomial long division by using polynomial coefficient lists as inputs. This is the "modulo" operation, so focus interest on the remainder list, not the quotient list from the long division. (Note: two APL functions, one named *POLYDIV* and another named *PDIVBY*, both described later, accomplish polynomial long division. They do the same division operation, but *POLYDIV* does it with recursion while *PDIVBY* uses looping.)

Compare the remainder list of coefficients with the list from the *CAMS* product of the two circulants. The two lists should be identical. This result is the tangible quantification of the words and symbols describing algebraically the structural relationship between polynomials and circulants.

Why in the world should this work? In particular what "magic" is involved in identifying the ideal generator (X*n - 1) as the proper choice for the modulo calculation? In my judgement this is just one of many instances where the theorems of abstract algebra shine with considerable glory. The interested reader should consult a text on modern abstract algebra for complete details. (The matter is partially clarified in a later part of this report.)

One book on the subject that I particularly like is *"Rings and Ideals"* by Neal H. McCoy, 1948. It is published by the Mathematical Association of America as one of the Carus series of Mathematical Monographs. Those not familiar with the Carus Monographs might like to know that their stated purpose is "...to contribute to the dissemination of mathematical knowledge by making accessible ... a series of expository presentations of the best thoughts and the keenest researches in pure and applied mathematics."

Chapter VIII, *Rings of Matrices,* is most helpful, as is the discussion on page 162 of that chapter about the fundamental theorem on homomorphisms. A number of other good texts on modern abstract algebra are also available. I suggest, for further investigations, *"A First Course in Abstract Algebra"* by John Fraleigh and *"Applied Abstract Algebra"* by Rudolf Lidl and Gunter Pilz.

Back from the Land of Abstract Algebra: the Polynomial-like Structure of a Circulant Matrix

There is a fundamental polynomial structure to circulant matrices that is surprisingly easy to display. It supports the existence of a useful pattern in program CAMP's table of intermediate results. It also validates the claim of structural equivalence between circ(v;n) and the residue class ring K[x]/(X*n - 1).

Consider the matrix circ(v;n) expanded into the matrix polynomial as follows,

*circ(v;n)= c1ÆA*0 + c2ÆA*1 + c3ÆA*2 +. . . + cnÆA*(n-1).*

Does there exist an n by n circulant matrix A for which this tentative reformulation is true? The answer is yes. The so-called "fundamental permutation matrix" — FPM for short — handles the job. This remarkable matrix is the innocent looking circulant defined as,

FPM = A = circ(v;n) = circ(0,1,0,...0)

Note that vector v in APL is simply `v←0,1,(n-2)ρ0` and that `ρv` is just n.

Here $A*n$ is I, the n by n identity matrix of all ones along the main diagonal and all zeros elsewhere, clearly also a circulant. Further all the powers of A, from 0 to (n-1) span the space of circulants. For want of a better word the FP matrix A may be called "cyclopotent". The choice reflects the influence of the terms "idempotent" and "nilpotent" used to describe other special behaviours of the powers of matrices.

The n by n matrix of all zeros is the additive identity matrix in the ring of circulants. Note that it is also a circulant. The symbol

```
A*0
```

is taken to be the multiplicative identity matrix I in the ring of circulants. APLers beware — any matrix raised to the zeroth power in APL will replace every element, even zero elements, with ones. The result is NOT the identity matrix.

The expansion in powers of A is clearly a polynomial in A whose coefficients are the elements of the vector v (which may be complex). The left hand side of the expansion relation is just as clearly a circulant matrix generated from the coefficients of the polynomial. Its first row is identical to the vector v of coefficients.

For two "circulant polynomials" constructed as above from two different defining vectors v, the operations of addition and multiplication follow the usual rules, with one important modification.

When multiplying two such polynomials, powers of A exceeding the (n-1)st may result. In those cases, rewrite the power of A as $A^*n \times A^*m$ and substitute I for A^*n. This leaves only the "residue" matrix A^*m, multiplied by its coefficient.

Complete the multiplication by collecting all similar powers A^*m together and adding up the common coefficients. Do this as necessary for every m from 0 to (n-1).

Note also that multiplication under these conditions results in powers of A that range only from $m = 0$ to $m = n$-1. Because A is cyclopotent, these n "residue classes" make up the only possibilities. In effect any power p of A is reduced modulo n, by utilizing the result that for A taken as the FPM, A^*n is I, the n by n identity matrix. (Although not explicitly stated so far, one needs to use also the property that the product of two circulant matrices is also a circulant matrix.)

A clue is now manifest regarding the rather mysterious choice of X^*n-1 as the ideal in the residue class ring discussed in the previous 'Abstract Algebra' section. Notice that A^*n - I can be rewritten as A^*n - I = 0.

Here A is the FPM, I is the identity matrix and the right hand side is the zero matrix. Replace A with the indeterminate X, I with the multiplicative identity unity, and the zero matrix with the additive identity 0 to get $X^*n - 1 = 0$. Then check some theorems in the cited texts about mappings and kernels and some definitions of ideals. At this point, the perceptive reader should begin to have doubts about algorithm *CAMP*. Multiply two n by n circulant matrices and you

get as a result another n by n circulant matrix. Assume there are no zero elements in the two original matrices. Then the coefficients in the first row of the result cannot be the polynomial product. Why? — because multiplying two polynomials with n non-zero terms will inevitably result in a polynomial with many more terms than n. Yet as just described, multiplying two circulant polynomials causes a "folding" of coefficients of powers greater than (n-1) into lower powers. No power of A greater than (n-1) occurs in the result.

Before proceeding, the reader needs to have an APL "one-liner" for multiplying two circulant matrices. The sceptic who has an APL interpreter can then check out the several calculations to come. The only tools needed are algorithms *TAMP*, *CAMP*, and *CAMS* — and *CAMS* is given next.

Circulant product algorithm, *CAMS*:

```
L+.×(⍳0-1⍴⍴,L←(⍴,R)↑,L)⌽(2⍴⍴,R)⍴,R
```

CAMS stands for (C)irculant (A)lgorithm for (M)ultiplying Circulant(S). Compare *CAMS* with *CAMP* to see that they are obviously closely related. *CAMS* does not produce the full circulant matrix, just the first row. Now here are some examples to clarify the workings of *CAMS* and *CAMP* and restore any lost faith in using function *CAMP* to produce polynomial products.

Use *CAMP* to multiply the quadratic polynomial whose coefficients are 1 1 2 by the quintic polynomial with coefficients 1 2 0 3 1 4. Get the seventh degree polynomial whose coefficients are 1 3 4 7 4 11 6 8. (Sceptics; check this with an ordinary paper and pencil multiplication of the two polynomials if you like.) Note that all coefficient lists are written in ascending order of powers and that if a power is missing its coefficient must be present as a zero in the list.

To continue, use *CAMS* to multiply the following 6 by 6 circulant matrices,

```
circ(1 1 2 0 0 0) +.× circ(1 2 0 3 1 4)
```

to get the 6 by 6 product

```
circ(7 11 4 7 4 11)
```

Take the first (*CAMP*) result and rewrite the single list of 8 items as two lists, folded after the sixth item and with zeros appended as necessary. This implements the necessary "powers of A modulo 6" calculation and gives table *T3*,

```
T3;   1  3 4 7 4 11
      6  8 0 0 0  0
```

Now add down the columns of table *T3* to get the list `7 11 4 7 4 11`. Bingo! The summation has reproduced the first row in the circulant product, as required when the powers of a product of two circulant polynomials are reduced modulo *n*.

Pretty neat, huh? Yeah except that there is little likelihood that very many APLers are interested in multiplication in the commutative ring of circulant polynomials with a program like *CAMS*. *CAMP* however seems as if it might actually be useful. How does algorithm *CAMP* reproduce ordinary polynomial multiplication? The perceptive reader may have already noticed the trick from close examination of the above example. The list of three coefficients for the quadratic polynomial was extended by three zeros to get a circulant matrix conformable with the circulant matrix corresponding to the list of six coefficients for the quintic polynomial. One has then two 6 by 6 circulant matrices, which being now conformable, can be multiplied together with function *CAMS*.

What happens if both lists, the quadratic and quintic lists, are extended with enough zeros to avoid the reduction of powers of *A* modulo *n*? The answer is that one then gets the desired ordinary polynomial multiplication. Algorithm *CAMP* accomplishes this extension very neatly by starting its calculation with the expression *R,L≠L*.

This in fact is a slight over extension of zeros, one too many, but the modest inefficiency is tolerable for simplicity of the APL expression. Algorithm *CAMP* has taken advantage of the zero padding to simplify the expected circulant matrix times circulant matrix operation. The nature of the zero padding allows the product to be performed instead as a greatly truncated circulant matrix (a vector) times a truncated circulant matrix.

APL Functions for Long Division of Two Polynomials

As promised earlier, here are two APL functions for polynomial long division, or PLD for short. PLD requires only the division of monomial terms with positive integer powers, the division of scalars selected from the real or complex number fields, multiplication of two polynomials, and subtraction of two polynomials.

Recursive APL function *POLYDIV* implements PLD for two polynomials *L* and *R*. The left argument *L* is the divisor and the right argument *R* is the dividend. Enter as a vector only the coefficients for polynomial *L*, in order from lowest to highest degree. If a monomial term is absent, an entry of zero is required in proper order for the missing term. Enter the polynomial *R* the same way. Do NOT pad any zeros after the coefficients of terms of highest degree!

If the coefficient list in *L* exceeds the length of the list in *R*, then the degree of the polynomial divisor exceeds the degree of the polynomial dividend. In this case only the trivial result, quotient equal to 0 and remainder equal to dividend, will be returned.

POLYDIV works just as well when it is fed polynomial coefficients from the field of complex numbers, but only for those APL interpreters with complex numbers as the default variable type. The function also works with real coefficients.

POLYDIV displays results for the quotient and remainder as coefficient lists. These are in the same ascending power order as the input list; embedded zeros in these results represent absent monomials, the same usage as for input lists.

```
    ∇ L POLYDIV R;K;QV;LXQV;Z
[1]   →(2=⎕nc 'Q')/NORESET ⍝ DO NOT RE-INIT GLOBAL VARS ON RECURSIVE CALLS.
[2]   Q←0 ⋄ DD←¯1↑L ⋄ LL←⍴,L ⋄ RMLXQV←,R ⍝ GLOBAL VARIABLES.
[3]  NORESET:
[4]   →(LL>⍴,R)/EOJ ⍝ QUIT IF DIVIDEND OR REMAINDER SHORTER THAN DIVISOR.
[5]   Q←(K←(¯1↑R)÷DD),Q ⍝ SCALAR DIVISION, ALWAYS WITH DD.
[6]   QV←K×⍳1,LL↑(R≠R) ⍝ ZERO PADDING ON RIGHT.
[7]   LXQV←¯1↓L+.×(⎕io-⍳LL)⌽(LL,⍴QV,L)⍴QV,L≠L ⍝ L×QV IS NORMAL POLY PRODUCT.
[8]   RMLXQV←R-LXQV ⍝ ZERO PADS HAVE ASSURED CONFORMABLE SUBTRACTION.
[9]   →(1=∧/⎕ct|RMLXQV)/EOJ ⍝ QUIT IF REMAINDER ALL ZEROS.
[10]  L POLYDIV(RMLXQV←¯1↓RMLXQV) ⍝ RECURSE ON DECREASING REMAINDER.
[11]  →0
[12] EOJ:
[13]  'QUOTIENT'
[14]  (¯1×1≠⍴Q)↓Q ⍝ CAREFULLY DROP SUPERFLUOUS ZERO.
[15]  'REMAINDER' ⋄ RR←RMLXQV
[16] (¯1×0≠⍴RR)↓RR,(0=⍴RR)×⍴RR ⍝ CAREFULLY HANDLE RR IF A ZERO LEN VECTOR.
[17]  Z←⎕ex 4 6 ⍴'Q     ','DD    ','LL    ','RMLXQV'
    ∇
```

I wrote the preceding recursive function for this article. Because it employs global variables, the user must insure that global variable *Q* has been erased from the workspace before execution. Alternatively, the user can execute cover function *POLY* to handle all initialization and execution duties. Left and right arguments are self-documenting.

```
    ∇ DIVISOR POLY DIVIDEND
[1]  ⍝ EXPUNGE GLOBAL VARIABLES BEFORE RECURSION.
[2]   Z←⎕ex 4 6 ⍴'Q     ','DD    ','LL    ','RMLXQV'
[3]   DIVISOR POLYDIV DIVIDEND ⍝ POLYNOMIAL LONG DIVISION BY RECURSION.
    ∇
```

Here is a second function for PLD that does not use recursion. It is function *PDIVBY*, as found on page 260 of U. Grenander's book *"Mathematical Experiments*

on the Computer". Note: Unlike function *POLYDIV*, function *PDIVBY* uses coefficient input lists and returns result lists in *DESCENDING* order with respect to the powers of *X*. Correct answers resulted from limited tests of *PDIVBY* with complex input lists. It probably would produce correct results for those APLs with a complex variable type as the default, but the user should first verify that capability. Aberrant results occurred for certain special cases of no operational interest, for example

```
(0j1) PDIVBY (1 0j¯1) and 1 PDIVBY 1.
```

```
    ∇ Z←P1 PDIVBY P2;LEAD;SHIFT
[1]   ⍝ RIGHT ARGUMENT: THE VECTOR OF COEFFS OF THE
[2]   ⍝ DIVISOR IN DESCENDING ORDER WRT THE POWERS OF X.
[3]   ⍝ LEFT ARGUMENT: VECTOR OF COEFFS OF THE DIVIDEND IN
[4]   ⍝ DESCENDING ORDER WRT THE POWERS OF X.
[5]   ⍝ RESULT: A 2×N ARRAY WHERE N=1+DEGREE OF DIVIDEND.
[6]   ⍝ THE 1ST ROW IS THE VECTOR OF COEFFS OF THE QUOTIENT,
[7]   ⍝ ORDERED THE SAME AS THE INPUTS.
[8]   ⍝ THE 2ND ROW IS THE VECTOR OF COEFFS OF THE REMAINDER,
[9]   ⍝ ORDERED THE SAME AS THE INPUTS.
[10]  →(((⍴P1)-(P1≠0)↑1)≥(⍴P2)-(P2≠0)↑1)/DIVIDE
[11]  Z←(2,⍴P2)⍴((⍴P2)⍴0),P2
[12]  →0
[13] DIVIDE:Z←(⍴P1)⍴0
[14]  LEAD←P2[DEGP2←(0≠P2←(-⍴P1)↑P2)↑1]
[15]  P2←(SHIFT←(DEGP2-(P1≠0)↑1))⌽P2
[16] LOOP:P1←P1-P2×Z[(⍴P1)-SHIFT]←P1[DEGP2-SHIFT]÷LEAD
[17]  P2←¯1⌽P2
[18]  →(0≤SHIFT←SHIFT-1)/LOOP
[19]  Z←(2,⍴P1)⍴Z,P1
    ∇
```

The Isomorphism in Action — It Performs as Advertised

Previously *CAMP* was used to multiply the quadratic polynomial whose coefficients are 1 1 2 by the quintic polynomial with coefficients 1 2 0 3 1 4. The result was the seventh degree polynomial whose eight coefficients are 1 3 4 7 4 11 6 8. Also previously, *CAMS* was used to multiply the following 6 by 6 circulant matrices,

```
circ(1 1 2 0 0 0) +.× circ(1 2 0 3 1 4)
```

to get the 6 by 6 product

```
circ(7 11 4 7 4 11)
```

How now to relate the seventh degree polynomial 1 3 4 7 4 11 6 8 to the circulant product 7 11 4 7 4 11? Answer — consider the sixth degree polynomial `X*6 - 1` and its related coefficient list of seven items,

```
¯1 0 0 0 0 0 1
```

Use the polynomial division program *POLYDIV* to divide the seventh degree polynomial by the sixth degree polynomial `X*6 ¯1`, as follows

```
¯1 0 0 0 0 0 1 POLYDIV 1 3 4 7 4 11 6 8.
```

Examine the remainder, not the quotient, because a "modulo" calculation is desired. Another Bingo! The remainder is 7 11 4 7 4 11, the same as the *CAMS* product. This result is not accidental. It is exactly the result that one should obtain from the structural relationship — the isomorphism — between circulants and polynomials.

A Danger and an Opportunity — Divisors of Zero

The Chinese ideograph for danger is the same as for opportunity. One dangerous aspect of matrix multiplication is that the matrix product can be zero. In such systems, mathematicians talk about "divisors of zero". In my judgement one could equally well phrase it as the "factors of zero". Two "somethings" annihilate each other and produce "nothing". Is there opportunity here as well as danger? Can one turn things around and opportunely use "nothing" to avoid the danger of "somethings" annihilating each other? Developed next is an answer that will also answer the puzzle posed at the beginning of this article. First, as promised earlier, here are two circulant matrices that annihilate each other when multiplied together. With *CAMS*, check that

```
circ(¯1 1 0) +.× circ(1 1 1)
```

produces the null circulant matrix, `circ(0 0 0)`. Hence `circ(¯1 1 0)` and `circ(1 1 1)` factor zero (are "divisors of zero"). "Something" times "something" has produced "nothing". Now pad each coefficient list with "nothing", actually a single zero, and try again. With *CAMS*, check that

```
circ(¯1 1 0 0 ) +.× circ(1 1 1 0)
```

now produces "something", the non-zero circulant `circ(¯1 0 0 1)`. Does this construction "use nothing to make something from nothing"? Well, it certainly uses zeros ("nothing") to make a circulant matrix ("something not zero") out of the null circulant matrix (`circ(0 0 0)`, which is "nothing") produced by the product of the circulant factors of zero.

How to Factor a Circulant Matrix

Polynomials have roots. Then what sense, if any, can be made out of the following procedure?

- Treat the first row of an *n* by *n* circulant as the coefficients of a polynomial of degree *n*-1, the coefficients being in ascending power order with respect to X.
- Assume the coefficients are from the real field. (Could equally well assume they are from the complex field. Actually almost any field will do. Finite fields will lead to some unusual happenings.)
- Factor the polynomial into its linear and possibly quadratic polynomial forms. To do this, solve for all the roots of the polynomial.
- Pad the coefficient list of two items of each linear factor with zeros on the right, until the list contains *n* items.
- Pad the coefficient list of three items of each quadratic factor with zeros on the right, until the list contains *n* items.
- With *CAMS*, multiply all the factors back together.

Does the product reproduce the original coefficient list for the polynomial of degree *n* - 1 and thus the original circulant? The answer is yes. In other words the original circulant matrix has been factored into a product of *n* - 1 simpler circulant matrices.

An example. Consider the 3 by 3 circulant with generating vector `v = 1 0 1`. This list corresponds to the quadratic X*2 + 1 with two imaginary roots `i` and `-i`

The two linear factors are `(X - i)` and `(X + i)`. Padding a single zero to each list, calculate `(-i 1 0) CAMS (i 1 0)`. The result is 1 0 1, the original circulant. The linear lists are circulant matrices whose product is the original circulant matrix. The original matrix has been factored. Note: Caution, APLers who try this using APLIWIN must write the imaginary number `i` as `0j1` and `-i` as `0j¯1`.

How about a tougher problem? Consider the quintic polynomial of an earlier example. Its coefficient list is the first row of the 6 by 6 circulant generated by v=(7 11 4 7 4 11). T. C. Chen's *SCARF* function in his SCARFS workspace is a robust and accurate root finder. It gives generally good results, but the iterative interaction involved requires patience and some experience in making a correct decision. The process appears to be somewhat tedious and not very practical for circulants of any significant size, say 20 by 20.

For the quintic example with coefficients 7 11 4 7 4 11, Chen's *SCARF* function returned these five roots:

```
      one real ∧ ¯0.59948564304709
imaginary pair ∧  0.62914846089673 +/-i×0.83550042957063
imaginary pair ∧ ¯0.51122382119136 +/-i×0.84205466888059
```

The reader may wish to check the roots with dyadic 'base' (⊥). Use `root ⊥ 7 11 4 7 4 11`, where root is one of the five listed. Results are fuzzy zeros, close to, but not quite zero. They are generally in the range of `10×⎕ct`, or less, for the APLIWIN system default for comparison tolerance. How about a different approach, say finding the greatest common divisor of two circulants? Progress might be easier because there exists a known algorithm to find the GCD of two polynomials. Further, with a polynomial division function available, this Euclidean algorithm for polynomials can be programmed with little difficulty. These facts unfortunately are not enough to provide a quick answer. The quest requires, among other things, a good understanding of the meaning of division in a commutative ring.

Pressing On

I began a search for the (possibly non-existent) Euclidean algorithm for circulants on the Internet. The hunt turned into quite an adventure on the 'net'. The full story is rather long and best told as a separate tale. One result is obviously the present article. Another result is the existence of a follow-on to this report. Presently only a draft, the investigation expounds upon the meaning of division in a commutative ring, affirms the existence of a Euclidean algorithm for circulants and pursues a tangible implementation thereof in APL.

The original motivation for the present article was not to find, pose and solve puzzles that seem to violate common sense. The larger purpose was to investigate ways to factor a particular kind of matrix, a circulant. One path led to the subject of Euclidean algorithms and their existence in general and in particular for circulant matrices. Another path led to finding roots of polynomials using Chen's SCARFS workspace.

At every turn, I ran into the need to learn more about modern abstract algebra and had to engage in a crash course of self-study of the subject. The topic seems ready-made for APL. I suggest there exists an opportunity for someone to author a book, perhaps titled *"APL and Modern Abstract Algebra"*. I believe both APLers and algebraists would benefit. (The latter to the extent that they have access to or are even aware of APL.)

Recapitulation

These APL programs have been displayed and discussed: long division of two polynomials, function *POLYDIV* and its associated cover function *POLY* (functions not relisted). Long division of two polynomials, function *PDIVBY*, (not relisted).

Thomson algorithm, modified,

```
TAMP: 1↓+/⍉((1-⎕io)-⍳⍴,L)∘(,L)∘.×R,L≠L.
```

TAMP stands for (T)homson (A)lgorithm for (M)ultiplying (P)olynomials.

Circulant algorithm, *CAMP*:

```
¯1↓L+.×(⎕io-⍳⍴,L)⌽((⍴,L),⍴R,L)⍴R,L≠L.
```

CAMP stands for (C)irculant (A)lgorithm for (M)ultiplying (P)olynomials.

Circulant product algorithm, *CAMS*:

```
L+.×(⎕io-⍳⍴,L←(⍴,R)↑,L)⌽(2⍴⍴,R)⍴,R
```

CAMS stands for (C)irculant (A)lgorithm for(M)ultiplying Circulant(S).

CAMP produces the product of two polynomials as a linear list of coefficients, lowest degree term to highest degree. The internal working of the algorithm is essentially just to multiply two circulants. Because of the effective use of zero padding inside the algorithm, no reduction modulo *n* occurs. The result can also be interpreted as the product of two circulant matrices, but this is not generally very useful because of the extensive zero padding used to avoid reduction modulo *n*. *CAMP* does not expect that in general the input lists of polynomial coefficients will be the same length. It will produce correct results for left and right argument lists of different lengths. Because circulants and polynomials commute, *CAMP* produces the same answer if the left and right arguments are exchanged, even if they are of different lengths.

CAMS produces the first row of the product of two circulant matrices. Such a list is equivalent to polynomial multiplication modulo the ideal generated by (*X*n* - 1). The exponent *n* is the order of the circulant matrices. *CAMS* right argument list can be longer than the left argument, but not vice-versa. Unequal input list lengths are generally not a good thing. Even though the function will zero pad the left argument to match the right's length, the user may inadvertently mix up inputs and so violate conformability requirements for matrix multiplication.

CAMS guarantees commutativity of its left and right arguments only if both lists are of equal length. Both arguments are best input with the same length, say *n*, which is also the exponent in the expression *X*n* - 1. Otherwise *n* is the length of the right argument.

Polynomial Long Division has produced verification in a specific example of the isomorphism between the ring of circulants and the domain of polynomials over a field, modulo the ideal generated by (*X*n* - 1). Judicious use of zero padding provides the key for solving the puzzle posed at the beginning of this article. If two circulant divisors of zero are so padded with the use of "nothing", they will produce "something" from "nothing".

Considered as a polynomial with real or complex coefficients, one can factor the first row of a circulant matrix. Just determine the roots of the polynomial, then judiciously pad the linear or quadratic coefficient lists. Treat these lists as circulant matrices, multiply them all together and they will reproduce the original circulant. One needs a very accurate root finder such as Chen's SCARFS workspace and even then the procedure is not practical for larger circulants. An alternative possibility would be to develop a Euclidean algorithm for two circulant matrices and so find a greatest common divisor circulant.

Note: All APL functions are designed to run under Windows 3.1 using Iverson Software's APLIWIN interpreter. Because almost all expressions are in upper case, the functions can be easily converted to run under IBM's APL2. The only "case changes" required are to convert all quad functions from lower to upper case. Thus `⎕io`, `⎕nc`, `⎕ct` and `⎕ex` need to become *⎕IO*, *⎕NC*, *⎕CT* and *⎕EX*, respectively, in functions *POLYDIV* and *POLY*.

Readers interested in experimenting with the functions can type them in from "scratch" with little difficulty. The author would like to be advised of any interesting results or aberrant behaviour. Email him at: frgp21a@prodigy.com

Acknowledgement

I thank Gerhard Niklasch of the Mathematical Institute, Technische Universität, Munich, Germany for his encouraging internet dialogues, valuable criticisms and cogent comments. As a novice, I often had to struggle with one or another of the various concepts of modern abstract algebra. I particularly appreciated his patient forbearance and helpful guidance in these instances.

Technical Note on Confidence Limits

by Norman Thomson

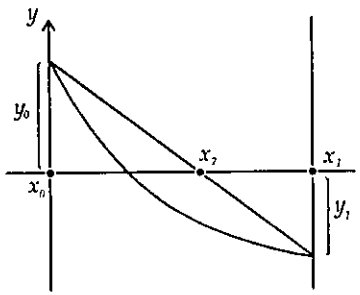
I begin this note by thanking Dietrich Trenkler for his contribution to the Random Vector in Vol.12 No.2, entitled *"Computing Clopper-Pearson Confidence Limits by the Illinois Method"* [1]. However I regretted that Dietrich did not exploit APL operators which greatly enhance the ease of the underlying programming problems, nor does his article make immediately clear the considerable application generality of the techniques he describes. These come under two quite separate headings of numerical analysis and statistics. Accordingly this note is an endeavour to rewrite, expand and clarify Dietrich's material.

Numerical Analysis

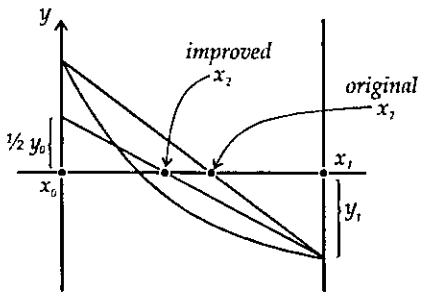
As Dietrich says, the Illinois Method is a refinement of Regula Falsi (sometimes called the Method of False Position), which in turn is a variant of the Secant Method, an operator-based algorithmic technique for which is described at length in *"APL2 in Depth"* [2]. All these methods are non-linear root-finding algorithms which have the objective of solving the equation $f(x)=0$ given two start values x_0 and x_1. All the methods use linear interpolation to obtain a further approximation x_2 based on the formula

$$x_2 = x_0 - \frac{y_0(x_1 - x_0)}{y_1 - y_0}$$

as quoted by Dietrich. The situation is illustrated graphically by



The basis of the Illinois method is that if, as in the case shown above, the graph of $f(x)$ shoots up to the left, it helps to accelerate the process by using $\frac{1}{2}\, y_0$ at the next step, thus:



In programming terms an algorithm is required which processes a large range of functions, and this is one of the things which an APL operator provides. An operator should be thought of as a function generator. In the present instance an operator *ILL* takes a specific function, say *f(x)=cos(x) - x exp(x)* (this was the one chosen by Dietrich for his illustration), and generates an appropriate function *F ILL* to locate one of the roots of *F*. The calculation of x_2 shown above is fundamental, and this, following a small amount of elementary algebra, can be expressed as the cross-product function:

```
Z←X cp Y
Z←((⌽X)-.×Y)÷-/Y  ⍝ cross product (x2.y1-x1.y2)/(y1-y2)
```

(I adopt a convention of using lower-case names for APL functions and upper-case names for operators.)

All the apparatus is now in place to define a single step for the operator *ILL* whose (single) operand is a function *P*:

```
[0]   Z←(P ILL)X;T
[1]   Z←X cp P⍣X                          ⍝ generate a new x-value
[2]   →(1=××/P⍣(1↓X),Z)/L1                ⍝ branch if Yi.Yi-1 >0
[3]   Z←Z,T cp P⍣T←(1↓X),Z ◇ →0           ⍝ obtain second new x-value
[4]   L1:Z←Z,T cp 0.5 1×P⍣T←(↑X),Z        ⍝   ditto    using .5×Yi-2
```

The final requirement is to repeat the process until a pair of successive solutions is within the desired tolerance as defined by the global variable *EPS*:

```
[0]   Z←(F RPT)X
[1]   Z←F ILL X              ⍝ Illinois method F=fn, X=start value
[2]  L1:→(EPS>|F ¯1↓Z)/0     ⍝ Result is intl, 2nd item=best approx
[3]   Z←F ILL Z ⋄ →L1        ⍝ Further step if Z not near enough 0
```

Taking the function *f(x)* above as an example, define

```
[0]   Z←F X
[1]   Z←(2○X)-X××X
```

The final two Illinois values for the root are then found, using the Dietrich's two start values of 0 and 1, by

```
      F RPT 0 1
0.5177573636 0.5177573637
```

The above is a completely general root-finding method, and if this is what you seek, the above eleven lines of APL are all that you require.

Statistics

Suppose that you have carried out an experiment in which you believe that the conditions of a binomial probability model prevail, that is, the probability of achieving, say, a 1 (as opposed to a 0) at each trial is constant, although possibly unknown. (Such a trial is often called a Bernoulli trial.) An observation is thus expressed as *R* out of *N*, for example 3 right guesses out of 10, 3 heads out of 10 tosses and so on. An observation could come about in the presence of any one of a whole range of underlying Bernoulli probabilities *P*, and so the probability of obtaining the observation can be described as a function of *P*. The observed result, *R* out of *N*, thus generates a function, which, as in section 1, suggests an APL operator. With *ILL* above the generator was a function, in this case it is a vector *N,R*. Anticipating the fact that the objective is to generate confidence limits, I combine the required confidence level in the form, say .95, with *N* and *R* to define an operator *GB* analogous to Dietrich's function *G_BINOM*:

```
[0]   Z←(LNR GB)P;L;N;R;T
[1]   (L N R)←LNR ⋄ T←0,⍳R     ⍝ R out of N is an obsvn;L∈[0,1]
[2]   Z←(+/(P*T)×((1-P)*N-T)×T!N)-L
[3]  ⍝ Z is cum binom prob(P,N;R) - L
```

By setting *L* to 0, *GB* can be used to obtain the cumulative binomial probability for any given *P*. For example:

```
      0 6 3 GB .5
0.65625
```

gives the probability of up to and including 3 heads in a toss of 6 coins. The experiment which Dietrich describes is one in which 13 out of 17 are observed, and so the estimate of *P* is 13/17 = 0.7647. The lower confidence limit is that value of *P* which is sufficiently small that the observation 13 out of 17 begins to become implausible. This, according to the Clopper-Pearson methodology, is recognised by the cumulative probability reaching 0.975 (for 95% confidence limits) for *R*=12 (n.b. not 13). So using Dietrich's start values of 0 and 1, use *ILL* to find this probability as

```
      ¯1↑(0.975 17 12 GB)RPT 0 1
0.50101
```

Similarly the upper limit is that value of *P* which is sufficiently large that the cumulative probability up to *R*=13 reaches no more than 0.025.

```
      ¯1↑(0.025 17 13 GB)RPT 0 1
0.93189
```

Generalising the method, define a function *clb* which takes a result and a confidence level and produces a Clopper-Pearson confidence interval:

```
[0]   Z←L clb NR;P;SI
[1]   ⍝ Z is binom conf lims given obsvn (N,R), level is eg. L=.95
[2]    Z←0 1 ◇ P←÷/⌽NR ◇ L←0.5×1+L          ⍝ Adjust L for 2-sided
[3]    SI←⌈CT⌈(⌊/|P-0 1)⌊((P×1-P)÷↑NR)*0.5 ⍝ Set Start Interval
[4]    →(0=↑⌽NR)/L1                          ⍝ Z[1]=0 if R=0
[5]    Z[1]←¯1↑(L,NR-0 1)GB RPT 0⌈P-3 1×SI  ⍝ solve for low lim
[6]   L1:→(=/NR)/0                          ⍝ Z[2]=1 if R=N
[7]    Z[2]←¯1↑((1-L),NR)GB RPT 1⌊P+1 3×SI  ⍝ solve for high lim
```

Dietrich uses 0 and 1 (the extremes of the probability range) as start values throughout. This not only fails to use the knowledge which is already present in the observation in order to speed up the convergence of the Illinois method, but also can cause failure of convergence in the case of *N* out of *N*, since the two start points must not be identical. I have tried various ideas for a universal start value formula, and my current best thinking is reflected in line [3] of the function *clb*, and in the corresponding lines of the matching functions for other distributions which are given below. I have not penetrated the matter of start values deeply,

and it is possible that there are better formulae for them, or indeed that the ones given may not always work. So long as the user has at least some degree of numerical and programming awareness, this need never be a severe problem in practice.

Dietrich gives the results of all confidence intervals for the case N=10, but declines to show the programming he needed to get there. Under my scheme of things this is achieved neatly using the each operator:

```
      T,,[⍳0]0.95clb¨10,¨T←0,⍳10
 0   0 0.3085
 1   0.0025286 0.44502
 2   0.025211 0.5561
 3   0.066739 0.65245
 4   0.12155 0.73762
 5   0.18709 0.81291
 6   0.26238 0.87845
 7   0.34755 0.93326
 8   0.44391 0.97479
 9   0.55498 0.99747
10   0.6915 1
```

Dietrich concluded his article by saying that this approach could be extended to the Poisson and negative binomial distributions. Rather than hand-wave, here are the corresponding operators:

Poisson:

```
[0]   Z←(LR GP)X;L;R;T
[1]   (L R)←LR ⋄ T←0,⍳R        ⍝ R is an integer obsvn; L∈[0,1]
[2]   Z←((*-X)×+/(X*T)÷!T)-L ⍝ Result is cum Poisson prob(X;R)-L
```

```
[0]   Z←L clp R;P;SI
[1]  ⍝ Z is Poisson conf lims for obsvn R, level is e.g. L=.95
[2]   Z←0 1 ⋄ L←0.5×1+L                    ⍝ Adjust L for 2-sided
[3]   →(0=R)/L1                            ⍝ Z[1]=0 if R=0
[4]   Z[1]←¯1↑(L,R)GP RPT 0.5 0.9×R
[5]  L1:Z[2]←¯1↑((1-L),R)GP RPT 2 3×R
```

Negative Binomial:

```
[0]   Z←(LXR GNB)P;L;X;R;T
[1]   (L X R)←LXR ⋄ T←0,⍳R    ⍝ R = no. of 0s before X 1s; L∈[0,1]
[2]   Z←(+/(P*X)×((1-P)*T)×T!X+T-1)-L
[3]  ⍝ Z is cum neg bin p(P,X;R) - L
```

```
[0]   Z←L clnb XR;P;SI
[1]   ⍝ Z=neg binom conf lims for obsvn (X,R), level is eg.L=.95
[2]    Z←0 1 ⋄ P←(↑XR)÷+/XR ⋄ L←0.5×1+L      ⍝ Adjust L for 2-sided
[3]    SI←⌹CT⌈⌊/|P-0 1                       ⍝ Set start interval
[4]    →(0=↑⍵XR)/L1                          ⍝ Z[1]=0 if R=0
[5]    Z[1]←¯1↓((1-L),XR)GNB RPT 0⌈P-0.2 0.5×SI
[6]   L1:Z[2]←¯1↓(L,XR-0 1)GNB RPT(1-⌹CT)⌊P+0.2 0.5×SI
```

Dietrich talks about the "coverage probability *c(p)*" meaning the probability that the true binomial probability lies within the confidence limits. In my view, *c(p)* is either 0 or 1, that is the confidence interval either does or does not include the true probability - the problem is that since you don't know the latter you don't know which of these two values *c(p)* possesses. In short, I do not understand Dietrich's coverage probability, which is a pity, because by not giving his program, Dietrich has missed an opportunity of using APL to describe his intentions in a situation where verbal communication by itself has proved inadequate. Also the diagram on page 93 has presumably suffered greatly in transcription, and so this doesn't help either.

What *is* meaningful is to compute the *relative likelihood* associated with the confidence interval, that is the *average* probability of the observation for *P* values within the limits, relative to the overall average probability. To compute this requires integration of the various probability functions, which in turn requires some more numerical analysis. For those readers who wish to persevere further this is where the next section returns.

More Numerical Analysis

A complete operator-based APL package was described in detail in [3]. A summary version is given here which contains the operator *SIMPSON*, which implements Simpson's Rule, and *ROMBERG* which builds on *SIMPSON*, and extends it satisfactorily for definite integration for a wide range of functions. In each case *P* is the function to be integrated and *R* is the range of integration. The number of Simpson intervals, *L* must be an even integer.

```
[0]   Z←L(P SIMPSON)R;T
[1]   Z←(T×+/(1,((L-1)ρ4 2),1)×P¨(↑R)+(T←-R-.÷L)×0,⍳L)÷3

[0]   R←V refine S
[1]   →(0≠ρ,V)/L1 ⋄ R←S ⋄ →0
[2]  L1:R←,(¯1↓V)refine S
[3]   R←R,(¯1↑R)+(¯1↑R-V)÷¯1+2*4×ρR

[0]   Z←(P ROMBERG)R;T;N
[1]   Z←T←(1+N←0)(P SIMPSON)R
[2]  L1:Z←(T←¯1↓Z)refine(2*N←N+1)(P SIMPSON)R
[3]   →(EPS<|T-1↑Z)/L1 ⋄ Z←¯1↓Z
```

There are situations in which what is called adaptive integration works where *ROMBERG* fails, and so an operator *ADAPT* is also given. *ADAPT* works by repeatedly distributing the tolerance between half-sized intervals, and so it is necessary to provide the initial tolerance, *E*, explicitly at invocation, whereas *ROMBERG* uses the global value *EPS*.

```
[0]   Z←E(P ADAPT)R
[1]   Z←4(P SIMPSON)R
[2]   →(EPS>|Z-2(P SIMPSON)R)/0
[3]   Z←+/(0.5×EPS)(P ADAPT)¨0 1¢¨R,¨0.5×+/R
```

The above fourteen lines of APL provide a complete general-purpose package for definite integration — a good example of a small amount of APL code providing a great deal of function.

More Statistics

Following the above, we are now in a position to compute the relative likelihoods for the binomial and negative binomial distributions by integrating over both the Clopper-Pearson range, and the maximum possible range, which is (0,1). The operators *PYB* and *PYNB* supply binomial and negative binomial probabilities. Expressing the parameter pairs *NR* and *XR* as operands allows their arguments *P* to be set up as suitable variables for integration.

```
[0]   Z←(NR PYB)P
[1]   Z←×/(!/⌽NR),(P,1-P)*(¯1↑NR),-/NR

[0]   Z←L r1b NR
[1]   Z←÷/(NR PYB ROMBERG)¨(L c1b NR)(0 1)

[0]   Z←(XR PYNB)P;X;R
[1]   (X R)←XR ⋄ Z←(P*X)×((1-P)*R)×R!X+R-1

[0]   Z←L r1nb XR
[1]   Z←÷/(XR PYNB ROMBERG)¨(L c1nb XR)(0 1)
```

There is no point in supplying corresponding operations for the Poisson case since mathematics guarantees that the relative likelihood is exactly equal to the confidence level.

It is interesting to evaluate the relative likelihoods for some sample binomial observations.

```
      .95 r1b¨(10 3)(100 30)
 0.98385   0.96238

      .95 r1b 1000 300
0.95354
```

Once N becomes large, say of the order of several hundred, the user needs to be sensibly aware of what is going on in the program. The problems arise because the functions being integrated eventually have values of the same order of magnitude as *EPS*. Some breathing space can be obtained by recognising that since *r1b* produces *relative* likelihoods the results of *PYB* and *PYNB* can be multiplied within the APL function by arbitrary constants. Switching from *ROMBERG* to *ADAPT* is another possibility, which was necessary in the last case above, although for simpler cases *ROMBERG* is faster.

In each case the relative likelihood is greater than the confidence level, which leads to the paradox that for a low N such as 10, the odds are better than 49 to 1 on the confidence limits including true p, even although the confidence level was only 95%. The explanation is that the probability of obtaining *precisely* the observed result is excluded from both summations in *c1b*. As N increases the relative likelihood (which is what you should put your money on!) moves closer to the confidence level. However even with N=1000 there is still quite a bit to go.

Relative likelihood is also observed to exceed confidence limits in the negative binomial case.

```
      .95 rlnb"(2 1)(2 3)(5 20)
0.99595   0.98218   0.95367
```

In a teaching context, this provides striking illustrations of the differences between confidence, probability, and relative likelihood.

I would like to emphasise the practical nature of such calculations. At work I was often asked what should be believed following an experiment in which small numbers of failures were observed, for example 2 out of 1000. The appropriate confidence limits are

```
      .95 clb 1000 2
0.00024 0.007
```

Using a Normal approximation the 95% confidence limits would have been quoted as -0.0008 to .0048, the former of which is ridiculous, while the latter leads to a falsely optimistic view of the true quality.

Norman Thomson
Finnock House
Cliff Terrace Road
Wemyss Bay
Inverclyde
PA18 6AP

References

[1] *Computing Clopper-Pearson Confidence Limits by the Illinois Method*, D Trenkler, The Random Vector, Volume 12, No.2, pp. 87-94, October 1995.

[2] *APL2 in Depth*, ND Thomson and RP Polivka, Springer-Verlag, 1995.

[3] *Integrating with Insight*, Norman Thomson, Vector Vol. 9 no. 3, pp. 113-116, January 1993.

TECHNICAL SECTION

This section of VECTOR is aimed principally at those of our readers who already know APL. It will contain items to interest people with differing degrees of fluency in APL.

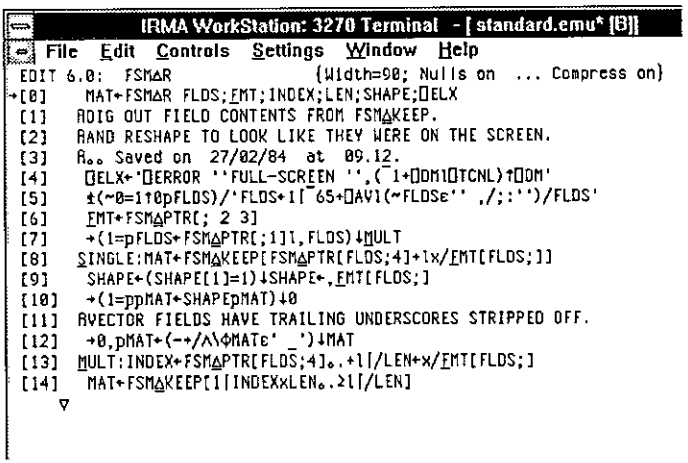
Contents

Hackers' Corner: a New Frock for IRMA

by Adrian Smith (100331.644@compuserve.com)

Motivation

I find it helpful to be able to read my own APL code occasionally, but essential to read other people's code which I am trying to maintain. As I write this, I am a few days into a short contract which involves changing over some quite complex mainframe code to run with a new set of input sources, while maintaining a lot of existing output structures. The first day — having got through the routine pay-and-rations stuff — was spent browsing workspaces and generally trying to understand what the existing system was doing. I found this to be a lot harder than it should have been, at least in part because of the quite awful rendering of the APL font by the IRMAWin software I was using. It seems as if the people who designed (if that is the word) the APL characters had set out to make APL nearly as hard to read as J! Here is the sort of thing I was faced with ...



Notice that almost all the APL symbols occupy far too much of the available width, that there is no differentiation of height so that [] ⌈⌊ do not show clearly, and that ⍳ and ⍴ look like nothing on earth. If I was finding this hard to read, what about all the other poor mainframe users out there, who have had their nice old terminals taken away and replaced by 'modern' Windows-based PCs. If you are suffering as I did — read on.

Investigation

Well, a font is a font is a .FON — even if it has some 32 different sizes in it so IRMA can pretend to be every known type of 3270 device. I already had a little APL*PLUS/PC workspace for maintaining font files, which I used to build the CAUSEWAY.FON that Duncan designed in the style of the old mainframe character set. I settled on 15 by 7 and 15 by 8 as nice sizes to work with, so the challenge was on to find the right place in the DCAAPL.FON file to patch. Now 15 is a good number to look for if you have a hex-dump utility for DOS files, as the file is listed in 32-byte sections so you get a very clear diagonal pattern from a 15-character repeat ...

```
| c6a0:  0  0  0  0 7c 80 80 80 7c  0  0  0  0  0  0  0 | ....|...|....... |
| c6b0:  0  0  0 78 84 84 84 84 78  0  0  0  0  0  0  0 | ...x....x....... |
| c6c0:  0  0 10 20 fe 20 10  0  0  0  0  0  0  0 fc  0 | ... . .......... |
| c6d0:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 | ................ |
| c6e0: 38 44 44 44 44 44  0  0  0  0  0  0  0  0  0 44 | 8DDDDD.........D |
| c6f0: 44 44 44 44 38  0  0  0  0  0  0  0  0  0  0 10 | DDDD8........... |
| c700: 10 10 10 10 7c  0  0  0  0  0  0 38 20 20 20 20 | ....|.....8       |
| c710: 20 20 20 20 38  0  0  0  0  0  0  0 c0 30  c 30 |     8........0.0 |
| c720: c0  0 fc  0  0  0  0  0  0  0  0  0 30 48 48 30 | ............0HH0 |
| c730:  0  0  0  0  0  0  0  0  0  0  0 64 94 88 88 88 | ...........d.... |
| c740: 74  0  0  0  0  0  0  0  0  0 3c 40 78 40 3c  0 | t.........<@x@<. |
| c750:  0  0  0  0  0  0  0  0  0 60 20 20 20 24 18  0 | .........`   $.. |
| c760:  0  0  0  0  0  0  0  0 18 24 44 44 64 58 40 40 | .........$DDdX@@ |
| c770: 40  0  0  0  0  0  0  0 44 92 92 92 6c  0  0  0 | @.......D...l... |
| c780:  0  0  0  0  0  0 44 28 10 28 44  0  0  0  0  0 | .....D(.(D...... |
| c790:  0  0 40 20 20 10 10  8  8  4  4  0  0  0  0  0 | ..@  ........... |
| c7a0:  0  0 30 30  0 fc  0 30 30  0  0  0  0  0  0  0 | ..00...00....... |
| c7b0:  0  0  0 7c 44 28 28 10 10  0  0  0  0  0  0  0 | ...|D((.......... |
| c7c0:  0 10 10 28 28 44 7c  0  0  0  0  0  0  0  0  0 | ...((D|......... |
| c7d0:  0 7c 10 10 10 10 10  0  0  0  0  0  0 38  8  8 | .|..........8.. |
| c7e0:  8  8  8  8  8  8 38  0  0  0  0  0  0  4  8 fc | ......8......... |
| c7f0: 30 fc 40 80  0  0  0  0  0  0 10 10 10 10 10 10 | 0.@............. |
```

If you don't have a suitable hex-dump utility, this one (written in C, by me, in 1988) is included on the DOSPP page on APL-385's web site at www.demon.co.uk/apl385. Now, each character in a .FON file is made up of n bytes, where n is the depth of the character cell. What we would expect to find here is a collection of integer vectors, each 15 long, having the pattern of the character represented as the topmost 7 bits (for the 15 by 7 set). All we need to do is read the file into the workspace (using *rget* from *Vector* 11.1 p126) and dig into it around the c700 spot to see what we find ...

```
      pff←rget'dcaapl.fon'
215040
      16⍴¯1+'0123456789abcdef'⍳'c700'
50944
```

```
      '.□'[1+⍉(8⍴2)⊤15↑50944↓ff]
...□....
...□....
...□....
...□....
.□□□□□..
........
........
........
........
........
........
..□□□...
..□.....
..□.....
..□.....
```

... looks hopeful. We have a straddle across what might be a base and what might be the top of a ceiling! Now to have a look at the font ordering in *charmap*, and sure enough we find that the ⊥ is one character before the [, on the positions normally given to N and O in the font. A little gentle arithmetic yields the base point of the font as *base*15←50232 (incidentally the 15 by 8 set is based at 87096) and we now have the following function to browse the characters:

```
      ∇see15[0]∇
[0]   r←see15 ch;pos
[1]   ⍝ 7 wide by 15-deep chars
[2]   pos←dcav⍳ch
[3]   r←'.□'[1+⍉(8⍴2)⊤15↑(base15+15×pos)↓ff]
      ∇
```

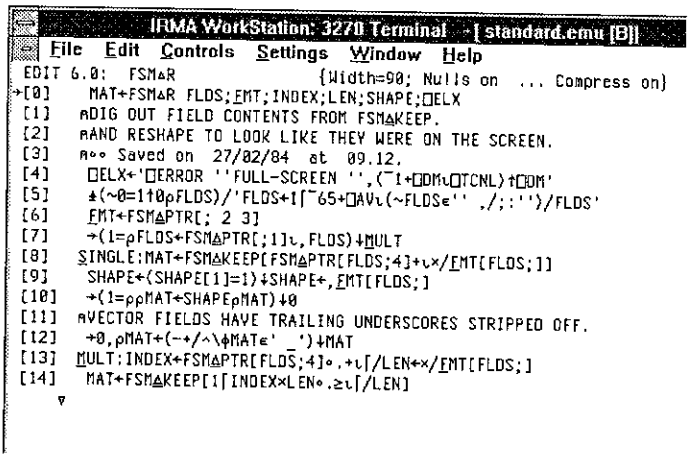
... where *dcav* was simply read off charmap and typed in. Now all we need to do is to read in the causeway font, lift the definition from there, trim unwanted rows or columns, and patch it in to the DCA font ...

```
      ∇put15[□]∇
[0]   ch put15 def;pos
[1]   ⍝ replace 15-deep char
[2]   pos←dcav⍳ch
[3]   (15↑(base15+15×pos)↓ff)←(8⍴2)⊥⍉~def∈' .'
      ∇

      ∇Patch[□]∇
[0]   Patch ch;cs
[1]   ⍝ Substitute baselined causeway char
[2]   cs←1⌽¯1⌽¯1 0↓see ch
[3]   ch put15 cs
      ∇
```

Results

The proof of the pudding is in the eating, and I am glad to report that my mainframe screen now looks like this ...



... which I find very comfortable to work with, and very nearly as readable as the original (excellent) 3179 fonts of 10 years ago. *Moral*: if your job is chopping down trees, there is ***always*** time to stop and sharpen your axe.

Getting Hold of this Stuff

It is probably not sensible for me to put the patched DCAAPL.FON up on the Web, as it has at least 37 copyright statements embedded in it, and I don't want to upset anyone. However the Dyalog workspace is absolutely tiny, so you can grab it from **www.demon.co.uk/apl385/ragbag** and save yourself some typing.

Next issue: *Making J Readable* (no, sorry ... even for the April Vector that was a joke in very poor taste — perhaps a Neural Net application to discriminate between J and line-noise would be more interesting?).

TECHNICAL CORRESPONDENCE

Divided Differences: Reply to Hui & Iverson

From: Norman Thomson March 1996

I should like to thank Roger Hui and Ken Iverson for contributing the elegant solutions for divided differences, Choleski and QR decomposition in the January *Vector* [1]. I would make the point that the MWY solutions reflect my empirical observation that most mathematicians, faced with a programming task, find it more comforting and secure to work at the cell level, rather than to "think array". I don't applaud this, but it does seem to be a fact that mathematicians, like teenagers, do not always do what they are told is good for them! Cell-level working is also the style of most numerical analysis algorithms given in the older classic text-books.

I used the Choleski algorithm because I wanted a simple, but non-trivial illustration of an algorithm which demanded a loop. Clearly I should have looked further for my example! I can only plead that my upbringing on the books of the pre-computer era has shielded me from the Choleski and QR algorithms in J which Ken and Roger give. These are built on results which are so delightful that I consider it worth while "drawing" them below in ways which make the recursive algorithms seem almost inevitable! C(X) means the Choleski matrix of the square symmetrical matrix X, and is the extension of the idea of "square root" to a matrix. Dashes represent transpose. The matrices are divided into halves either by both rows and columns in the case of Choleski, or by columns in the case of QR. Where the number of columns is odd, the extra one is to the left of the dividing vertical bar.

$$C\left(\begin{array}{c|c} X & Y \\ \hline Y & Z \end{array}\right) = \begin{array}{c|c} C(X) & 0 \\ \hline Y'X\ C(X)^{-1} & C(Z-Y'X\ C(X)^{-1}\ Y) \end{array}$$

QR(A0|A1) =

Q(A0)	Q(G)

R(A0)	Q(A0)'A1
0	R(G)

where G = A1- Q(A0).Q(A0)'.A1.

I hope that this further note may encourage others to revisit the J algorithms.

Reference

[1] Hui, Roger KW and Iverson, Kenneth, *A Note on Programming Style*, Vector, Volume 12 no. 3, pp.117-121, January 1996

Linear Recurrences and Matrix Powers

From: Roger Hui 29th February 1996

I read with enjoyment and admiration accounts of Eugene McDonnell's excellent adventure in "Heron's Rule & Integer-Area Triangles" in the January *Vector* (12.3 *pp* 133—142). The article discusses Sloane's sequence 700, `s=: 1 2 7 26 97 362` ... with recurrence relation

```
A(n) = 4A(n-1) - A(n-2)
```

where `(2*s)+/_1 0 1` are the sides of integer-area triangles. Computing the sequence using the double recursion is found to be extremely slow, and is made faster using generating functions, Taylor's series, and partial fractions.

There is another fast solution, based on array operations. The recurrence can be written as the matrix equation:

$$\begin{bmatrix} A(n-1) \\ A(n) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 4 \end{bmatrix} \begin{bmatrix} A(n-2) \\ A(n-1) \end{bmatrix}$$

This form makes clear why it's called a *linear* recurrence. In J,

```
   x=: +/ .*                    Matrix multiplication
   M=: 0 1,:_1 4
   A0=: 1 2

   M x A0
2 7
   M x M x A0                      (M x M) x A0
7 26                            7 26
   M x M x M x A0                  (M x M x M) x A0
26 97                           26 97

   M&x^:(i.5) A0                   (M&x^:(i.5) =i.2) x A0
 1   2                           1   2
 2   7                           2   7
 7  26                           7  26
26  97                          26  97
97 362                          97 362
```

Thus the n-th element of the sequence obtains by n repeated multiplications by `M` or, equivalently, by multiplying by the n-th power of `M`. The n-th power of a matrix can be computed by repeated squaring, with a consequent reduction from O(n) operations to O(log n) operations.

We illustrate the process for `M` to the 30-th power:

```
   [ b=:#:30                    Binary representation of 30
1 1 1 1 0

   b # i.-#b                    Powers corresponding to 1s in b
4 3 2 1

   M x M                        Squaring a matrix
_1  4
_4 15

   x~ M
_1  4
_4 15


   x~^:4 3 2 1 M                Powers with squaring as the function
_109552575  408855776
_408855776 1525870529

     _2911      10864
    _10864      40545

       _15         56
       _56        209
```

```
         _1          4
         _4         15

   (M&x^:16 8 4 2 =i.2) -: x-^:4 3 2 1 M
1

   x/ x-^:4 3 2 1 M          Product of selected powers
_1.11401e16 4.15753e16
_4.15753e16 1.55161e17

   pow=: 4 : 'x/ x-^:(b#i.-#b=.#:y.) x.'

   M pow 30
_1.11401e16 4.15753e16
_4.15753e16 1.55161e17

   M pow 0
1 0
0 1

   M&x^:30 =i.2              Power by repeated multiplication
_1.11401e16 4.15753e16
_4.15753e16 1.55161e17

   x/ 30# ,:M                Insert x between 30 copies of M
_1.11401e16 4.15753e16
_4.15753e16 1.55161e17
```

The three techniques for computing matrix powers are applied for various n, with the following timings (milliseconds; J3.01 on Windows 3.1 on 80486/50):

```
                    30      60      90      12

M&x^:n =i.2       10.4    17.6    26.4    33.5
x/n#,:M            9.3    17.5    25.8    34.0
M pow n            6.6     9.4     9.9     9.3
```

It remains to define a function to compute the n-th element of the sequence:

```
   seq=: [: {. M&pow x A0"_

   seq"0 i.10
1 2 7 26 97 362 1351 5042 18817 70226

   seq 30
7.20106e16
```

The preceding techniques are applicable to any linear recurrence (on any number of terms). For example, the Fibonacci sequence obtains from `M=:0 1,:1 1` and `A0=:0 1.`

A Different DDE Application

by John Sullivan

Regular readers will remember my diatribe on p.7 of Vol.11 No.4 concerning the mangling of Welsh grammar and spelling in an earlier issue. As a postscript I suggested (with some tongue in cheek) that it might be an idea for the *Vector* production team to obtain the new Welsh spell-checker from the Welsh Office, in Cardiff.

The version for Windows, called CySill, was released in August, 1995. Being interested in most things from west of Clawdd Offa (sorry, Offa's Dyke), I ordered my copy and after a short while it arrived. I read the manual, and configured it to work with my word processor. As I read on, I came across a page headed "DDE Interface to CySill". Text is put into the clipboard, the spell-checker is called from another program via DDE, and the corrected text is then rescued from the clipboard and put back into your application. This sounded like an ideal opportunity to show that DDE is not restricted just to APL and the MS Office suite of programs, and that it will work with others as well.

When CySill is installed its directory must be included in the DOS path, so to start it up all you need is the name of its executable file, with no directory path. Only one instance of CySill can run at any one time: trying to start it a second time from the desktop just makes the existing instance active. This makes programming very much simpler because if we suspect that CySill is not already running we just try to start it up and if this fails we can't go any further so we exit gracefully. Of course we could check to see if CySill is already running (see [2] for how) but there seems little point.

For the purposes of this exercise I have written some simple functions with lots of verbosity to enable me to see what's happening. In any real application the functions would be much more terse. The ideas were obtained from Adrian Smith's paper at APL93[1].

The algorithm I used is

```
Can we share with CySill?
If No then
  Assume that CySill is not running so try to start it up
  If the return from our start-up attempt implies that CySill is already running then
    Exit gracefully
```

Can we share with CySill?
If No then
Exit gracefully
Put the text into the clipboard
Request CySill to check the text
Do until CySill has finished
Ask CySill if it has finished
Obtain the corrected text from the clipboard
Minimize CySill and wait for its next call.

The main function looks like this:

```
     ∇ z←Check z;CL;command;reply
[1]   ⍝ Perform Welsh spell checking on the text in <z>
[2]    →(2='DDE:CySill|archwilio'⎕SVO'command ±')/Δ1
[3]    'CySill will not accept DDE. Maybe it is not running. Trying
to start it up.'
[4]    →(32<reply←StartMin'CYSWIN')/Δ3
[5]    →(16=reply)/Δ3       ⍝ If we tried to start it a 2nd time
[6]    'Unable to start CySill, return code is ',▼reply ◇ →0
[7]   Δ3:→(2='DDE:CySill|archwilio'⎕SVO'command ±')/Δ1
[8]    'CySill will still not accept DDE!' ◇ →0
[9]   Δ1:'CL'⎕WC'Clipboard'
[10]   'CL'⎕WS'Text'z
[11]   command←'GwirioClipboard'
[12]  Δ2:→(0=reply←CySillRequest'Cyflwr')/finished
[13]   →(0≠⍴,reply)/Δ2
[14]   'You closed CySill before it finished. Cannot continue.' ◇ →0
[15]  finished:z←'CL'⎕WG'Text'
[16]   ±('ie'≡CySillRequest'Cau')/'''You requested Close on the file
menu.'''
[17]   ±('na'≡CySillRequest'Newid')/'''No change.'''
[18]   command←'Minimize'  ⍝ Keep open for next time
     ∇
```

There are two called functions, one to make the DDE requests and the other to start up the application if it is not already running.

```
     ∇ z←CySillRequest x;y
[1]   ⍝ Send a request to CySill and return its reply
[2]    z←''          ⍝ Default if CySill is incommunicado
[3]    →(2='DDE:CySill|archwilio'⎕SVO'y ',x)⍴0
[4]    z←y
     ∇
```

```
      ∇ z←StartMin x;WinExec
[1]    ⍝ Start a Windows app. and minimize it, don't make it active
[2]    ⎕NA'U kernel.exe.P16|WinExec <0T U'
[3]    z←WinExec x 7
      ∇
```

In this last function I could have used `z←⎕CMD x 'minimized'`, but this fails with a Domain Error if it can't start CySill, rather than giving a return code to help you determine what is wrong.

There is no need to kill the task, because the chances are that having used it once you'll need it again soon, so you might as well keep it alive.

Note that, unlike the applications in MS Office, you do not enclose the DDE commands to CySill in square brackets (in fact, if you do they will not be recognized!).

References

[1] Adrian Smith, *Co-operative programming with Windows DDE*, APL93 Conference Proceedings (APL Quote Quad, Vol.24, No.1) p.244.

[2] Duncan Pearson, *A Windows Task-Killer for APL*, Vector, Vol.10, No.2, p.126.

CySill, The Welsh spell-checker, is available from:

Welsh Language Board
Market Chambers
5-7 St Mary Street
Cardiff CF1 2AT

Telephone 01222-224744.

An Internet Extension to 3D Noughts and Crosses

by Timo Laurmaa (100316.3367@compuserve.com)

Introduction

The competition announced in *Vector* 12.3 depicted a situation, where instead of me playing against one of my children sitting next to me, the PC could be the opponent - probably annoyingly skilled, always concentrating, never failing to take advantage of my mistakes. Although the programming task sounded like a challenge, I wanted to approach the problem in a different way by enabling two people, both using their own PCs, to play against each other via the Internet.

Duncan Pearson's original game is based on three events. The Internet extension needs a fourth event and some additional steps into the original three (all enhancements are marked with *italics*) to be introduced:

- **New Game** (a menu event) initialises the variables, draws the grid and gives the turn to the red *(i.e. the client).*
- **Place** (a mouse event) converts the mouse co-ordinates into the ball index (0 to `-1+n*3`), *sends the ball index to the other player,* checks for a win and switches to the other player's colour.
- **Spin** (a keyboard event) rotates the grid; no change - both players may spin their grids independently and do not know about their opponent's viewing angle.
- *Receive (TCP/IP event) takes place when the other player has placed a ball. The* **Place** *event is then created, with the exception that since the ball index is already known, the conversion from mouse co-ordinates is skipped.*

Communication Protocol

The TCP/IP protocol, which creates the connection between the two players and sends the ball index after every move, is built using AP119/W by Lingo Allegro, Inc. (see review on page 38). The event-driven approach to TCP/IP programming is particularly useful in an application like the 3D Noughts and Crosses, which is inherently event-driven.

The first player selects the menu item *New (as Internet Server)*. AP119/W creates a listening socket and waits for "clients". Nothing else happens until the second player selects the menu item *New (as Internet Client)*, and is prompted for the server's IP address. A socket with a connection to our not-so-well-known port number 3333 is established, and the game starts with the client's turn to place the first ball. The socket remains open to the end of the game.

Each placement of a new ball is communicated to the other player with the TCP/IP *SEND* command, detected by the *READ* event on the other side. When either player has won, the socket is closed and the server is free to play with other clients.

Changes to Duncan's Original APL Code

Only three essential changes to the programs listed in *Vector* 12.3 were required:

1. Two new menu items allow for starting the game as a server or a client:

```
Make[28-29]

'form.menu.game.new2'⎕WC'menuitem'
    'New (as Internet &Server)'
    ('event' 'Select' '⍎colour[]←0 ⋄ Connect 1 ⋄ Draw')

'form.menu.game.new3'⎕WC'menuitem'
    'New (as Internet &Client)'
    ('event' 'Select' '⍎colour[]←0 ⋄ Connect 0 ⋄ Draw')
```

2. If the ball index is received from the other player, the code that handles mouse co-ordinates is skipped:

```
 Place[11]
→Lb1/⍨~hismove←0=≡hit←msg
```

3. The player's own move has to be communicated to the opponent:

```
 Place[24-26]
Lb1:

 →0/⍨~×colour[hit]

 Message hismove↓hit
```

Other Issues

Since there is, apart from yourself, nobody around to say "Wake up, it's your turn", I recorded two *.wav* files *yourturn.wav* and *plsewait.wav* (the names are self-descriptive), one of which is played after a move took place. I modified the standard *PLAY* program to replay the last sound file every 30 seconds to simulate a real-life situation where intensive thinking is interrupted by a reminder "Come on, it's your turn".

I also took advantage of the open socket and allowed the players to send comments to each other. Whenever a character string, which clearly is not a ball index, is transmitted, it is simply displayed in the APL session.

Cheating is possible as it is in real life. There is nothing to prevent you from placing a ball on the other player's turn. However, if you have a sound board and the speakers are on, you will hear that something went wrong.

Conclusion

Of all client/server applications that I have designed, this one demonstrates the most literal participation from the server. A crucial difference with a single-PC, two-player game is that once the grid gets populated, it is increasingly difficult to see where the opponent placed the new ball - it may pop up just as you are looking elsewhere.

Networked 3D Noughts and Crosses can be played at APL96, and the complete workspace can be downloaded from *http://www.demon.co.uk/apl385*. (But you need to have AP119/W rel 2.0 in order to play the game.)

Program Listings

The *TCPxxx* programs are not listed here.

```
 Connect n;a
⍝Start game as a client (n=0) or a server (n=1)
⍝Timo Laurmaa 23.2.1996
 →n/Lb2                                          ⍝ Jump if server
⍝Playing as a client
 ⍎(0=⎕NC'IPADR')/'IPADR←''X.X.X.X'''             ⍝ Initial IP address mask
Lb1:IPADR←IPADR Win_input'IP address of server'  ⍝ Prompt for server's IP address
 →0/⍨0∈⍴IPADR←' '~⍨IPADR                         ⍝ Quit if nothing entered
 →Lb1/⍨'X'∈IPADR                                 ⍝ Jump if clearly wrong input
 a←0 TCPSOCKET IPADR 3333 'ThreeD'               ⍝ Port 3333, protocol ThreeD
 a←PLAY'YourTurn.wav'                            ⍝ Client starts always
 →0
Lb2:a←PLAY'PlseWait.wav'                         ⍝ Wait until a client joins
 →Lb3/⍨Mode=0                                    ⍝ Jump if single PC so far
 →0/⍨(⊂'ThreeD')∈PROTOCOLS                       ⍝ Exit if already waiting/playing
Lb3:a←1 TCPSOCKET'0.0.0.0' 3333 'ThreeD'         ⍝ Wait for clients


Message v;n;s
⍝Tell the other player what your move was
⍝Timo Laurmaa 23.2.1996
 →Lb1/⍨0∈⍴v                              ⍝ Jump if he moved
 →0/⍨Mode=0                              ⍝ Exit if single PC mode
 s←((⍴PROTOCOLS)↑⊂'ThreeD')⍳⍵SOCKETS     ⍝ The last "3D" socket is for sending
 a←s TCPSEND'N'(⍕v)                      ⍝ Send without conversion
 a←PLAY'PlseWait.wav'                    ⍝ Now waiting for his move
 →0
Lb1:a←PLAY'YourTurn.wav'                 ⍝ Your turn.


 ThreeD V;EV;SN;VAR;T
⍝ 3-D Noughts and crosses TCP/IP event handler
 (EV SN)←V                                       ⍝ Event nr, socket
 →1 (EV=1 32)/READ CLOSE                         ⍝ Select the event
READ:A←TCP'RECV'SN 0 'N'                         ⍝ Read without translation
 →MOVE/⍨(1 2∨.=⍴A)∧∧/A∈'1234567890'               ⍝ Other player's move: n or nn
 ⍎('⍎'≠1 A)/'⎕←A ⋄ →0'                          ⍝ Display if not to be executed
 Exe 1↓A ⋄ →0                                    ⍝ Execute with error trapping
MOVE:TCPINFO'The other player's move: ',A        ⍝ Write log info about the move
 Place⍎A ⋄ →0                                    ⍝ Call Noughts and Crosses, exit
CLOSE:colour[]←0 ⋄ Draw                          ⍝ Initialise game if socket lost


 z←Mode;L;SN
⍝Indicated whether the game is played in a single-PC, Server or Client Mode
⍝z ↔ 0 (Single-PC) or 1 (Client) or 2 (Server, playing) or 3 (Server, waiting)
⍝Timo Laurmaa 3.3.1996
 z←0                                     ⍝ Default: Single PC
 →0/⍨(0=⎕NC'SOCKETS')∨2≠⎕SVO'TcpIp'       ⍝ Exit if single PC mode
 →0/⍨1≠z←2⌊+/L←PROTOCOLS≡¨⊂'ThreeD'       ⍝ How many "3D" sockets open
 SN←SOCKETS[L⍳1]
 z←z+2×1 0≡2 ⍷1⊃TCP'SELECT'SN SN SN ¯1    ⍝ Return 3 if only listening socket open
```

At Play with J
Year's Digits for 1996

by Eugene McDonnell

This problem is a variation of an old one that originated as a Fortran puzzle in the MIT alumni magazine, adapted for use with J. Here it is:

Create a character table T, having 101 rows, each row representing a J expression, according to the following rules:

(a) The result of executing row i must be the atom i, and

(b) The characters '1', '9', '9', and '6' must appear in that order in each row, and no other digits may be present. (In the Fortran puzzle, the digits could appear in any order.)

Expressed in J, (a) each row

```
r =. i { T
```

must satisfy the requirement that

```
i -: ". r
```

for i an item of `i. 101` (and thus an atom), and (b)

```
'1996' -: r -. a. -. '0123456789'
```

There are two additional requirements, suggested by Roger Hui:

(c) Character constants are not permitted. If they were then all solutions would need no more than two tokens. For example 7 could be represented by

```
#'1 9 9 6'.
```

(d) J allows 'b' form constants, in which a decimal integer base appears to the left of 'b' and the digits to the right of 'b' may include not only the digits 0 through 9 but also the letters a through z, representing digits 10 through 35. For example, the octal representation of 63 is `8b77` and the hexadecimal representation of 255 is `16bff` and the decimal number 100 can be written as `1buzz`. The b form of constants is allowed, but the digits a through z are excluded, as well as 0 2 3 4 5 7 8. If a through z were not excluded almost all solutions would be one token long.

Here are some examples of invalid rows. The reason each example is unacceptable is given directly after it.

`19+6+9`	The digits are not in the prescribed order.
`1+96+1`	The digits are not 1996.

`3*19[96`	It contains a '3'.
`1{.99 6`	It yields a list result, not an atom.
`#'  1996'`	It uses a character constant.
`1bzp996`	It uses the digits z and p.

As a valid example, row 19 might be

```
+/1 9 9[6
```

and this satisfies the test `19 -: ". '+/1 9 9[6'`.

The objective of the problem is to use the minimum number of tokens in each row, as measured by the J 'Word Formation' primitive (`;:`). The foregoing list for row 19 has 5 tokens, and it is thus superior to:

```
1+9+9[6
```

which uses 7 tokens, but it is inferior to

```
19<.96
```

which uses only 3 tokens.

Entries will be judged in the following way: if `L` is the list of the number of tokens in each row of a given entry, and `M` is the list of the minimum number of tokens in all entries submitted, then the entry which minimizes `+/L-M` is the winning entry.

To ease your minds, I should say that yes, a complete set of solutions is always possible, and this has been demonstrated mathematically by Donald Knuth and Roger Hui, among others. Since `*1996` is 1 then `^.*1996` is a solution for 0; and since `^.o.1` is between 1 and 2, then applying floor or ceiling gives solutions for 1 and 2. Using more instances of `o.` provides solutions for larger numbers, ad infinitum. Clearly, this shows that a solution is always feasible. Most derived using this method are not, however, very short. Coming up with a short solution for each integer is your problem.

To help you get started, let me suggest that you use a strategy like that employed by Roger Hui. He used a J session in the following way to develop his table:

He worked with two windows present on his screen: an executable window, and a script window called "`1996.js`" which contained one solution per line.

Initially, each row is set with the row number, a comma, some spaces, and a 0. For example, row 25 would look like this:

```
25,     0
```

You can write potential solutions in the script window, and have them executed in the execution window to see if they are correct:

```
25,     1+9+9+6
```

Roger provided himself with a suite of utility functions:

```
mat=:   (5&}.);._2 @(1!:1) @((<'1996.js')"_)
len=:   /:~@(({.,#)/.~)@:(#@;:)
check=: *./@(0&= +. (=i.@#))@:".
pfx =: [: ": #@;: ,. i.@#
tab =: [: \:~ pfx ,. ]
```

"mat" reads the script file and constructs a matrix from it. As it stands, it is suitable for use with IBM-compatible PCs. To change it for use on Unix or Macintosh systems, you should replace the text '(5&}.@}:)' with '5&}.'. "check" checks that each row is either zero (unsolved) or has the correct number. "len" makes a two-column table with the first column giving a length and the second column giving the number of solutions with that length (unsolved numbers have a length of 0). "tab" makes a table of the solutions sorted in decreasing length, and thus is handy for attacking the really bad solutions.

I wrote the following, to check that only the digits '1996' appear, in that order, in the solution:

```
d1996=.*./@([:('1996'"_ -: ] -. a."_ -. '0123456789'"_)"1 ])
```

To see what these utilities can do for you, after you've created your 1996.js file and filled in a few entries, experiment with expressions like:

```
$mat 0
check mat 0
len mat 0
+/*/"1 len mat 0  NB. total number of tokens
tab mat 0
```

And after you've filled in all the entries,

```
d1996 mat 0
```

This problem should help familiarize you with some lesser-known parts of J, like b-form constants, the new p: and q: primitives, and the monadic, or base-2 form of the base primitive (#.). For example, the following five-token expression:

```
    #.p:q:|_19b96
91
```

creates the number _19b96, which has the decimal value _165 (in base _19 the values 9 and 6 evaluate to _171 and 6, with sum _165); takes the magnitude of this number, yielding 165; finds its prime factorization with q:, yielding 3 5 11; uses p: to find the third, fifth and eleventh primes in the 0-origin series 2 3 5 7 11 13 17 ..., yielding 7 13 37; and applies the primitive #. to evaluate this list in base-2, yielding 91 (+/4 2 1*7 13 37). Another five-token expression for the same value is:

```
    >:1#.q:996
91
```

There is a solution to 91 which is shorter than this, by the way.

Send your solutions to me either by electronic mail at: eemcd@aol.com or by regular mail to:

Eugene McDonnell
1509 Portola Avenue
Palo Alto, CA 94306
USA

If you think you have a particularly good solution for a given number but don't want to do the whole problem, send it to me anyway; it may merit a special commendation.

Roger and the Amazing Technicolor Ballclock

by Norman Thomson

This note arises from Roger Hui's article on the Ball Clock Problem [1], and is addressed to readers who may have been overwhelmed by the amount of the material covered, the speed with which Roger swept through it, or possible difficulties in obtaining copies of the references.

It would be a pity if such readers were to miss out totally on some of the marvellous things which Roger sped through on his voyage to solve the ACM Programming Challenge, and so in the spirit of slow-lane J, I shall try to expound some of Roger's nuggets in greater detail.

First, here is Roger's basic suite of J verbs which I have not amended, except by re-ordering them in a hierarchical fashion, excluding references to the debug verb `13!:8`, and adding the adverb `each`.

```
pow1=.(^:(]`(i.@#@[))                    NB. perm x. raised to the power y.
pow=.i.@#@[ C.~(#&>@C.@[|])#C.@[         NB. alternative form of pow1
ord=.*./@(#&>"_)@C.                      NB. order of perm = gcd of cycle lengths
log=.{:@(cr/)@(C.@[ mr ])                NB. inverse of pow - logs are modulo ord
cr=.[:|/\*.&{. , ,&{: +/ .* ab           NB. Chinese remainder algorithm
 ab=.!.@(gcd/ * [%+./)@(,&{.)
gcd=.(}.@{.)@(it^:(*@{.@{:)^:_)@g0       NB. gcd in form (a,b) where gcd=ax.+by.
 it=.{: ,: {. -{: * <.@%&{./
 g0=., ,. =@i.@2:
mr=.#&>@[,.(res&><)                      NB. x.=C. a generating perm; y.=a perm
 res=.<:@#@[-{i.{:@[                     NB. each row of mr=#cycle,posns in cycle
each=.&>
```

This note is primarily about permutations. A permutation is an arrangement of objects, and this discussion will be restricted to those permutations which can be obtained by executing the expression `n?n` . A permutation is thus a vector such as the following which Roger used as part of his illustrations:

```
p=.2 3 4 5 6 7 8 1 9 0
```

The verb `C.` converts a permutation into cycle notation thus

```
   C.p
┌─────┬─────┐
│7 1 3 5│9 0 2 4 6 8│
└─────┴─────┘
```

Cycle notation can be thought of as a "permutation *process*" that is `C.p` is a description of the process which generates the permutation `p` starting from `i.10` in the following way. Regard `p` as a sequence of items numbered from 0 to 9, the first cycle should be read as follows:

position 7 is to contain item 1,
position 1 is to contain item 3,
position 3 is to contain item 5,
position 5 is to contain item 7, thereby completing the cycle.

Read the second cycle in the same way. The overall permutation process is the result of doing *both* these cycle operations at every step.

The cycles themselves are disjoint and exhaustive, and each cycle is rotated so that its largest item appears first. Also the boxed cycles are arranged in ascending order of their leading elements. All of this is detailed in the J dictionary, and it guarantees that the cycle representation of the permutation process is unique.

Repeating a permutation process *n* times can be alternatively described as raising the permutation to the power `n`. This is the basis of Roger's two verbs `pow` and `pow1`. They both have the same function — `pow` should be easy to read and understand for anyone reasonably familiar with J; `pow1` is a little less so, but is more efficient. Any permutation raised to the power 0 is `i.n` where `n = #p`.

Thus raising `p` to the power `n` is achieved by `p pow n`, and the results of raising it to the powers 0,1,..5 is

```
   >(<p)pow each i.6
0 1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 1 9 0
4 5 6 7 8 1 9 3 0 2
6 7 8 1 9 3 0 5 2 4
8 1 9 3 0 5 2 7 4 6
9 3 0 5 2 7 4 1 6 8
```

The columns headed 1 3 5 and 7 each contain these digits in repeated 4-cycles, and the remaining columns each contain the remaining digits in repeated 6-cycles. This means that whenever a power is reached which is a multiple of both 4 and 6 the permutation `i.10` will recur. The smallest such power is 12, and

more generally it is the least common denominator of the lengths of the cycles. This quantity is called the order of the permutation and is given by Roger's function ord.

```
   ord p 12
```

log p is the inverse of power adjusted to modulo ord p. It addresses the problem: given one of the rows of the above table, what is the power to which p must be raised to generate it. Again, Roger supplies the verb:

```
   p log 8 1 9 3 0 5 2 7 4 6
4
```

More generally

```
   p log p pow 99
3
```

since 3 is the 12-remainder of 99. In mathematics this is expressed as 3≡99(mod 12); in J or APL as 3=12|99. Since permutations of n are obtained as a primitive verb in J, experimentation is extraordinarily easy, for example:

```
   [u=.8?8
6 1 3 2 5 4 0 7
   C.u
```

1	3 2	5 4	6 0	7

```
    >(<u)pow each i.8          NB. Table of 8 powers of p
0 1 2 3 4 5 6 7
6 1 3 2 5 4 0 7
```

... repeated four times

```
   >(<u)log each(<u)pow each i.8
0 1 0 1 0 1 0 1

   ]u=.8?8
   >(<u)log each(<u)pow each i.8
0 1 2 3 0 1 2 3
```

Roger uses an algorithm associated with the so-called Chinese remainder theorem. What this delivers is solutions to simultaneous congruences such as

x≡2(mod 3) x≡3(mod 5) x≡2(mod 7)

or, in simple English, it finds numbers which have remainders 2, 3 and 5 when divided by 3, 5 and 7 respectively. The verb which Roger supplies is called `cr`. If there are just two congruences, say the first two above, the solution is

```
   3 2 cr 5 3
15 8
```

that is, 8 is the smallest number whose remainders on division by 3 and 5 are 2 and 3 respectively, and the number formed by adding any multiple of 15 to 8 will also possess this property. If there are more than two congruences, use each:

```
   >cr each/3 2;5 3;7 2
105 23
```

so 23 and 128 are the smallest numbers satisfying all three congruences.

Thus equipped, problems such as the following, beloved of puzzle book writers, become trivial:

> When eggs are removed from a basket 2,3,4,5,6 at a time, the numbers left eventually are 1,2,3,4,5 respectively, but when 7 eggs are removed at a time none are left over. How many eggs are there in the basket, given that there are less than 200?
>
> 17 pirates decided to divide the booty of gold coins, but when they did so, 3 coins were left over. In the ensuing brawl, one pirate was killed, so they started again. This time 10 coins were left over. Another pirate perished violently, and the remainder were able to share the coins equitably. What is the smallest number of coins they had to share?

The answers to the above are:

```
   >cr each/2 1;3 2;4 3;5 4;6 5;7 0
420 119

   >cr each/17 3;16 10;15 0
4080 3930
```

A condition for `cr` to deliver the correct answers is that the moduli of all the congruences must be co-prime in pairs. Roger demonstrates a test for this which depends on having the debug facility `13:8` available.

Reference

[1] *The Ball Clock Problem*, Roger KW Hui, Vector Vol.12 No.2, pp. 56-66, 1995.

VECTOR

INDEX TO BACK NUMBERS

Vols 1 – 12 inclusive

Alphabetical by Author

APRIL 1996

Acknowledgements

Index to Vol.1 and Vol.2: David Preedy

Index to Vol.3 and Vol.4: Gavin Schwarzenbach

Index to Vol.5: Adrian Smith

Index to Vol.6 to Vol.12: Adrian & Gill Smith

Index to Advertisers

All queries regarding advertising in VECTOR should be made to Gill Smith, at 01439-788385, Compuserve: 100331,644.

Submitting Material to Vector

The Vector working group meets towards the end of the month in which Vector appears; we review material for issue n+1 and discuss themes for issues n+2 onwards. Please send the text of submitted articles (with diskette as appropriate) to the new Editor:

Duncan Pearson,
Keeper's Cottage
Firby
YORK, YO6 4LH
Tel: 01653-618900
Email: 100265.1564@compuserve.com

Authors wishing to use Windows Write or Word for Windows should contact Vector Production for a copy of the Vector APL TrueType font, a suitable Winword template and the Vector APL typewriter.

Camera-ready artwork (e.g. advertisements) and diskettes of 'standard' material (e.g. sustaining members' news) should be sent to Vector Production, Brook House, Gilling East, YORK YO6 4JJ. Please also copy us with all electronically submitted material so that we have early warning of possible problems.

Tel: 01439-788385
Compuserve: 100331,644.

British APL Association: Membership Form

Membership is open to anyone interested in APL. The membership year normally runs from 1st May to 30th April, but new members may join from 1st August, November or February if preferred. The British APL Association is a special interest group of the British Computer Society, Reg. Charity No. 292,786

Name: _____
Address: _____
_____
Postcode / Country: _____
Telephone Number: _____
Email Address: _____

Category (please tick box) to run from: 1st May ❑ August ❑ Nov ❑ Feb ❑

UK private membership	£12	❑
Overseas private membership	£14	❑
Airmail supplement (not needed for Europe)	£4	❑
UK Corporate membership	£100	❑
Corporate membership overseas	£135	❑
Sustaining membership	£430	❑
Non-voting UK member (student/OAP/unemployed only)	£6	❑

PAYMENT — in Sterling or by Visa/Mastercard/JCB only

Payment should be enclosed with membership applications in the form of a UK Sterling cheque to "The British APL Association", or you may quote your Mastercard, Visa or JCB number.

I authorise you to debit my Visa/Mastercard/JCB account

Number: ⌊_⌊_⌊_⌊_⌋ ⌊_⌊_⌊_⌊_⌋ ⌊_⌊_⌊_⌊_⌋ ⌊_⌊_⌊_⌊_⌋ Expiry date: ⌊_⌊_⌋ / ⌊_⌊_⌋

for the membership category indicated above,

❑ annually, at the prevailing rate, until further notice
❑ one year's subscription only

(please tick the required option above)

Data Protection Act:
The information supplied may be stored on computer and processed in accordance with the registration of the British Computer Society.

Signature: _____ Send the completed form to:

British APL Association, c/o Rowena Small, 8 Cardigan Road, LONDON E3 5HU, UK

The British APL Association

The British APL Association is a Specialist Group of the British Computer Society. It is administered by a Committee of officers who are elected by a postal ballot of Association members prior to the Annual General Meeting. Working groups are also established in areas such as activity planning and journal production. Offers of assistance and involvement with any Association matters are welcomed and should be addressed in the first instance to the Secretary.

1995/96 Committee

Chairman:	Dr Alan Mayer 01792-205678x4274 a.d.mayer@swansea.ac.uk	European Business Management School, Swansea University, Singleton Park, SWANSEA SA2 8PP
Secretary:	Sylvia Camacho 0117-973 0036 100612.1057@compuserve.com	11 Auburn Road, Redland, BRISTOL, BS6 6LS
Treasurer:	Nicholas Small 0181-980 7870 treas.apl@bcs.org.uk	8 Cardigan Road, LONDON E3 5HU
Journal Editor:	Anthony Camacho 0117-973 0036 acamacho@cix.compulink.co.uk	11 Auburn Road, Redland, BRISTOL, BS6 6LS
Activities:	Vacant Post	
Education:	Dr Ian Clark 01388-527190 100021.3073@compuserve.com	9 Hill End, Frosterley Bishop Auckland Co. Durham DL13 2SX
Technical:	Vacant Post	
Publicity:	David Eastwood 0171-922 8866 MicroAPL@microapl.demon.co.uk	MicroAPL Ltd., South Bank Technopark, 90 London Road, LONDON SE1 6LN
Recruitment:	Jon Sandles 01904-612882 100257.1756@compuserve.com	138 Burton Stone Lane, York YO3 6DF
Administration:	Rowena Small 0181-980 7870 treas.apl@bcs.org.uk	8 Cardigan Road, LONDON E3 5HU

Journal Working Group

Editor:	Anthony Camacho	0117-973 0036
Production:	Adrian & Gill Smith	Brook House, Gilling East, YORK (01439-788385)
Advertising:	Gill Smith	Brook House, Gilling East, YORK (01439-788385)
Support Team:	Jonathan Barman (01488-648575), Duncan Pearson (01653-618900), Richard and Adam Weber (01302-539761), Sylvia Camacho, Ray Cannon (01252-874697), John Searle (0181-858 6811), David Ziemann (0181-348 4039), Jon Sandles	

Typeset by APL-385 with MS Word 5.0 and GoScript

Printed in England by Short-Run Press Ltd, Exeter

VECTOR

VECTOR is the quarterly Journal of the British APL Association and is distributed to Association members in the UK and overseas. The British APL Association is a Specialist Group of the British Computer Society. APL stands for "A Programming Language" — an interactive computer language noted for its elegance, conciseness and fast development speed. It is supported on most mainframes, workstations and personal computers.

SUSTAINING MEMBERS

The Committee of the British APL Association wish to acknowledge the generous financial support of the following Association Sustaining Members. In many cases these organisations also provide manpower and administrative assistance to the Association at their own cost.

Causeway Graphical Systems Ltd
5 The Maltings, Castlegate,
MALTON, North Yorks YO17 0DP
Tel: 01653-696760
Fax: 01653-697719
Email: 100265.1564@compuserve.com
Web: www.causeway.co.uk

Compass R&D Ltd
10 Frederick Sanger Road
Surrey Research Park
GUILDFORD, Surrey GU2 5YD
Tel: 01483-302249
Fax: 01483-302279

Dyadic Systems Ltd
Riverside View, Basing Road,
Old Basing, BASINGSTOKE,
Hants, RG24 0AL
Tel: 01256-811125
Fax: 01256-811130
Email: sales@dyadic.com
Web: www.dyadic.com

HMW Trading Systems Ltd
Hamilton House,
1 Temple Avenue,
LONDON EC4Y 0HA
Tel: 0171-353 8900
Fax: 0171-353 3325
Email: 100020.2632@compuserve.com

Insight Systems ApS
Nordre Strandvej 119A
DK-3150 Hellebæk
Denmark
Tel: +45 42 10 70 22
Fax: +45 42 10 75 74
Email: insight@inet.uni-c.dk

Manugistics
2115 East Jefferson St
Rockville
MARYLAND 20852 USA
Tel: +1 (301) 984-5412
Fax: +1 (301) 984-5094
Email: aplsales@manu.com (US)
Email: intl@manu.com (International)

MicroAPL Ltd
South Bank Technopark
90 London Road
LONDON SE1 6LN
Tel: 0171-922 8866
Fax: 0171-928 1006
Email: microapl@microapl.demon.co.uk
Web: www.microapl.co.uk

Soliton Associates Ltd
Groot Blankenberg 53
1082 AC Amsterdam
Netherlands
Tel: +31 20 646 4475
Fax: +31 20 644 1206
Email: sales@soliton.com

Dutch APL Association
Postbus 1341
3430BH Nieuwegein
Netherlands
Tel: 03474-2337