Table of Contents

Ch-ch-changes

Stephen Taylor
editor@vector.org.uk

This is the first issue of *Vector* to appear without the BCS logo. Following an almost unanimous vote by our members we have ended our twenty-year affiliation to the British Computer Society. In his article “Changes at the BAA”, Paul Grosvenor discusses the reasons for this change, and what it now makes possible.

The cover bears our new logo and we have taken this opportunity to refresh the design. There are changes inside the covers as well. With generous support from Dyalog (whose silver-anniversary history appears as a separately-bound supplement to this issue) we have upgraded our production process. Since 1987, *Vector* has been produced using Microsoft Word, thanks in very large measure to the skill of Adrian Smith in handling difficult font-mapping and typographical issues.

The arrival of Unicode, and Adrian’s own work on the APL385 Unicode font, has allowed us in the last few years to present APL code on the Web far more simply. That in turn has allowed us to import HTML documents straight into Word, hugely reducing our typesetting work. This issue of *Vector* takes the next step: the markup has been changed from HTML to DocBook, an XML-based system widely used in technical publishing. This is the first issue to be produced without Word, and brings our production process into the publishing mainstream.

Even without the character mappings of various APL implementations, typesetting *Vector* is demanding. Mathematical typesetting is a specialised skill. Where more verbose programming languages generally have their source code set as separate blocks, terse APL and J expressions appear ‘in-line’ all through our articles, reflecting the role Iverson intended for the ‘tool of thought’, but demanding the greatest care and attention in setting and proofing. (Is that comma part of an expression, or does it punctuate the English sentence?) Even had BAA been able to afford them, it is highly unlikely we could have produced *Vector* with professional typesetters. BAA owes Adrian Smith an unpayable debt of gratitude for keeping *Vector* in production for over two decades until the technology caught up.

Appropriately enough, Adrian describes in this issue his first encounters with Dyalog’s new support for Unicode.

APL publishing has become more active outside our pages too. This issue reviews Jeffry Borror’s new textbook *q for Mortals*. Graeme Robertson has published two APL textbooks, Dyalog has a new textbook in preparation from Bernard Legrand,

and *At Play With J* is now being proofed, the first title to appear under the Vector Books imprint.

Array languages are receiving renewed attention with the expiry of Moore’s Law. Unable to push clock speeds higher, computer manufacturers push more processors into their boxes. But sharing work between them is hard, and this has revved interest in functional- and array-programming languages. Microsoft has established a research centre around the F# group at Cambridge. Web 2.0 networking site LinkedIn.com now boasts a functional-programming group with over 300 members; the Iversonians group there has itself grown to nearly 90.

In this issue David Liebtag reports how APL2’s new parallel-each operators tackle load sharing, and Neville Holmes continues his tutorial series on functional calculation. R.E. Boss discusses how to partition numbers quickly.

Much in this issue on the ‘tool of thought’ theme. Simon Marsden thinks his way through a problem using classes. His moves will surprise OO programmers unfamiliar with interpreters, as much as it will APLers unfamiliar with classes. Gianluigi Quario finds new ways to write and think about polynomials, and Norman Thomson works through handling complex numbers in J. Celebrated educator Ray Polivka makes a welcome debut in *Vector* with this issue’s “In Session” column, returning to the derived and direct ‘ruler’s edge’ functions in *Vector* 23:1&2 and taking them back to traditional forms.

Dan Baronet reports in “SALT II” on extensions to Dyalog’s code-management tools.

Finally, we are pleased to publish an autobiographical essay by Ken Iverson. This has been extracted from the manuscript of *The Story of APL & J*, that he and Donald McIntyre were collaborating on up to his death.

In our next issue we’ll have reports and papers from the Dyalog and APL2000 conferences this autumn: see the conference programmes in “Industry news”.

NEWS

Quick reference diary

Meetings

12-15 Oct	Elsinore, Denmark	Dyalog 2008 Conference
10-11 Nov	Bethesda, MD	APL2000 User Conference

Future issues of *Vector*

	Vol.24	Vol.24	Vol.24
	N°1	N°2	N°3
Copy date	21 Oct	21 Jan	21 Apr
Distribution	December 2008	March 2008	June 2008

Back numbers

Back numbers of *Vector* are available from:

British APL Association
c/o Gill Smith
Brook House, Gilling East
York YO62 4JJ

Price per complete volume (4 issues) including postage:

- £10 in UK
- £12 overseas
- £16 by airmail

NB: Vol.1 N°2 is out of stock.

Changes at the BAA

Paul Grosvenor, British APL Association
chairman@vector.org.uk

A little bit of history

Toward the end of 2007 and the start of 2008, the British Computer Society announced changes to the way in which it manages and runs its Specialist Groups, which included, of course, the BAA. These changes were wide-ranging and had various knock-on effects. Perhaps the most obvious and controversial change was the requirement for all SG members to be members of the BCS.

In the particular case of the BAA, where many of our members are retired from mainstream work or are based abroad, BCS membership would offer few advantages for them and additional expense; even though the first year's membership would be free.

More fundamental however were growing demands from the BCS of the BAA organisational committee, which prevented us from running our organisation in the way we wanted on behalf of the APL community. The restrictions placed upon us were seen to be increasingly obtrusive, right down to how we print *Vector* and to whom we may distribute it.

After much heartfelt discussion, we held a ballot of the members on a resolution to end our affiliation to the BCS. We announced at our EGM on 19 June that this resolution had been passed. The voting was 184 in favour and 1 against. To get so many APLers all agreeing with each other is a rare event indeed and only goes to show the depth of feeling and importance of the issues under debate.

There remains one outstanding issue regarding the BCS: our funds, which remain behind closed doors whilst we negotiate their return to us. There are all sorts of legal issues surrounding the 'ownership' of the funds which I shall not detail here other than to say that we consider the funds to be ours and that we will fight vigorously to recover them. Until this issue is resolved we aim to run the BAA with a starting bank balance of zero. With the continuing, and generous, support of our Sustaining Members we should have working capital very shortly.

Now we are independent and responsible for our own destiny or demise.

So, what happens now?

Well, immediately the decision to de-affiliate was made we had to change our Regulations accordingly and as a result identified a number of additional areas which were now somewhat out of date. Richard Nabavi and Anthony Camacho have volunteered to investigate and report back with suggested updates.

We have decided not to change too much, too quickly. We plan to move forward with care, keep existing initiatives in place where possible and build confidence from our membership in our abilities. Your committee remains largely unchanged from pre-BCS days but with the addition of Chris Hogan who will be auditing our accounts going forward.

At the time of writing we had nearly completed the process of opening a new bank account for the BAA but unfortunately, due to the money-laundering laws in the UK, this takes a long time. Once this account is up and running we can start to collect fees again and have a financial base from which to work. I am sure Nicholas Small will be in touch soon!

We continue to produce *Vector* as before and in fact plan to produce even more varied copy and output over the future months. *Vector* 23:4 is due to go to press by the end of September; just in time for the conference season. Stephen Taylor continues to work on our production process to make the whole thing faster, more streamlined and less reliant on individuals. We hope that you will see a more reliable delivery over the coming years with content from a variety of new (and old) sources.

Over the past months we have been changing our web site so as to deliver a more dynamic content – "as it happens", so to speak. We would encourage all of our members to have a look around the site and let us know what is good, or bad, and maybe provide some useful content for the future. This site is often the first point of contact for APL entrants and so we are very keen to ensure that the content provides for all needs. The 'Community' section provides for all sorts of links to related material, people, associations etc. Please help us add to it and keep it up to date. Thanks to Ian Clark for all of his efforts in getting the *Vector* back issues into the archive.

A proposal has been put forward to try and produce additional booklets from time to time using much of the scattered material already available. We hope therefore to bring to you some new literature over the months, bundled in with your *Vector*, which will be informative and build into a useful reference library over time. Some of you may well be asked to provide us with some 'ingredients' for our Celebrity APL Cookbook!

In the coming months we will be looking at things such as the fee structure and amounts, individual roles and responsibilities plus the recruitment of new blood into our committee structure. I consider this essential to get that dynamism back into the group and to start investigating new avenues of involvement.

Prior to our de-affiliation we had plans to hold a 2-day conference in London early next year. I would still like to see this go ahead, as in my view, a BAA-led conference is long overdue. Exactly what we can do on this front will depend upon our working capital etc. but, at the moment, consider 2-3 March 2009 as the possible date. More information on this will be forthcoming once our funding position with BCS is resolved.

Various other projects remain in-hand; work on the Vector Archive continues; Kai Jaeger expands the APL Wiki ever further and we support him where we can. We remain in contact with other associations particularly SIGAPL who also are renegotiating their relationship with their governing board (ACM SGB).

Finally, we scrapped the blog on our website and now use the comp.lang.apl newsgroup to post messages and questions out to the APL world: this seemed to be one of the most appropriate and vendor-independent mechanisms easily available. Since we have started, this group is now at its most active for 10 years. If you haven't looked for a while, log on and have a read – or even post a comment.

In summary

Far from a gloomy outlook, I see the coming months as very exciting. The APL community seems to have woken up in the past year or so and our challenge is to keep up! We must feed new information and ideas through to the community and ensure that initiatives do not fall by the wayside.

That challenge is great but then so too is the BAA. (Urgh – I've just come over feeling rather ill!)

Extraordinary General Meeting of the BAA

Anthony Camacho, British APL Association
secretary@vector.org.uk

The meeting was held at 5 Southampton St on Thursday 19 June 2008. The chairman opened the meeting at 1:30.

The secretary reported that he had counted the votes cast by midnight on 16 June 2008 and that the resolution proposed by the committee had been approved by an overwhelming majority of 184 to 1. The resolution below was therefore passed.

1. The affiliation of the Association to the British Computer Society as a Specialist Group is terminated.
2. The Regulations of the Association are to be modified as follows:
   i. Paragraphs 2, 11a, 11d, 11e and 13 are deleted.
   ii. Paragraph 11b is to be amended to read: "The Association shall maintain a Current Account with a UK Clearing Bank. The Committee shall be empowered to place funds in investment accounts."
3. The Chairman is to write to the British Computer Society informing them of this change and thanking them for their administrative assistance in the years since the Association first became affiliated to the British Computer Society.
4. The Treasurer is to recover the Association's funds from the British Computer Society and deposit the funds according to Regulation 11b.

The chairman said that the committee would now inform the BCS and make appropriate arrangements.

The chairman closed the meeting at 1:40.

Regulations of the British APL Association

Anthony Camacho, British APL Association
secretary@vector.org.uk

1. Name

The Group shall be called The British APL Association.

2. Objects

a. To promote the use of APL;
b. To develop awareness and competence in APL;
c. To represent the interests of Association members with other bodies;
d. To contribute to the development of the language and definition of international standards.

3. Constitution

The British APL Association shall consist of:

a. President and Vice Presidents;
b. Chairman, Secretary and Treasurer;
c. Other elected officers;
d. Individual fee-paying members;
e. Individual fee-paying student members;
f. Corporate members;
g. Sustaining members;

4. Members in Good Standing

A member in categories 4(d), 4(e), 4(f) or 4(g) is in good standing with the Association by virtue of having paid all membership fees due to the Association for the current year. Any membership which is not renewed within the first four months after becoming due is deemed to have lapsed.

5. Appointed Officers

a. A President may be appointed at the discretion of the Management Committee.
b. A maximum of four Vice Presidents may be appointed in recognition of service to the APL community at the discretion of the Management Committee.

c. Appointed Officers may serve for a period of one year with up to two consecutive renewals. Non-contiguous periods of office are permitted with no limit.
d. Appointed Officers shall have the right to attend free of admission charge all General Meetings of the Association and all events staged by the Association.

6. Elected Officers

a. The Elected Officers shall be Chairman, Secretary, and Treasurer, who must be individual fee-paying members in good standing.
b. Other officers may be elected to fill posts created by the Management Committee.
c. It is the duty of the Elected Officers to attend meetings of the committee.
d. Election: The officers shall be elected by postal ballot – to serve from 1 May in the year of election until the following 30 April. Individual members will be sent one ballot paper; corporate and sustaining members will be sent five to be cast by five users of APL where possible.
e. Casual Vacancy: A vacancy occurring during the term of office may be filled by an appointment by the Management Committee.
f. The occupant of any post who has held it for three consecutive years is not eligible for election to that post in the fourth year.

7. Management

a. The affairs of the Association shall be managed (subject to the control of the AGM) by a Management Committee comprising:
   i. Appointed Officers
   ii. Elected Officers
b. Co-option: The Management Committee may co-opt members as required, normally to hold office until the following 30 April.
c. The General Committee of the Association shall comprise the Management Committee together with members of Sub-Committees and Working Parties which from time to time be set up by the Management Committee.
d. Meetings: The Management Committee shall meet at least four times in its year of office and frequently enough to carry out the business of the Association properly.
e. Notice: At least fourteen days notice of the place, date and time of meetings shall be given to each member of the relevant committee.
f. Quorum: The business of the Association may be transacted by not less than half the members of the Management Committee.

g. In the absence of the Chairman, the Management Committee shall elect one of its members to take the chair for the meeting.
h. Voting: In determining a question by vote of the majority of members present, each having one vote, the chairman of the meeting shall have a second or casting vote.
i. Standing Committees: The Management Committee is empowered to set up Standing Committees to deal with matters that require continuity. A member of the Management Committee shall sit on each Standing Committee.
j. Sub-Committees and Working Parties: The Management Committee may set up at any time sub-committees and working parties responsible to the Management Committee which shall appoint a Chairman and provide appropriate terms of reference.

8. Annual General Meeting

a. Each year the Association shall hold an AGM in May.
b. Notice: The Secretary shall send, at least 28 days before, notice of the date, time and place of the AGM, together with the Agenda, to all members of the Association. For this purpose a notice printed in the official publication of the Association shall be considered sufficient.
c. All members have the right to attend the AGM for which there shall be no attendance charge.
d. Agenda: The following items shall be included:
    i. Minutes of the previous AGM;
    ii. Minutes of any EGM held since the previous AGM;
    iii. Chairman's Report;
    iv. Management Committee Officers' Reports;
    v. Audited Statement of Accounts;
    vi. Proposals for Alterations to the Regulations;
    vii. Proposals for Alterations to the Fees;
    viii. Ratification of Auditors.
e. Nominations: A Nominating Committee will draw up a list of Candidates for each elected post; additional candidates may be nominated by a proposer and seconder. All candidates, proposers and seconders must be members in good standing. Closing date for nominations is 1 February.
f. Voting: Every question at the AGM shall be decided by a majority of the votes cast. Individual members of the Association each have a single vote. The accredited representative of each corporate and sustaining member has five votes which may be split as desired. If a vote is tied, then there shall be a re-

count; if this fails to resolve the matter the candidate with the least votes is removed and the vote retaken, this process continuing until there is a decisive vote – or it cannot be continued, in which case the Chairman has a casting vote.

9. Extraordinary General Meeting

a. An Extraordinary General Meeting (EGM) shall be convened on a resolution of the Management Committee or within five weeks of receipt of the Secretary of a requisition signed by no less than one quarter of the current membership in good standing stating the business to be transacted at the meeting.
b. An EGM shall transact only such business as is specified in the resolutions or requisitions concerning it.

10. Finance

a. The Association shall maintain a Current Account with a UK Clearing Bank. The Committee shall be empowered to place funds in investment accounts.
b. The financial year shall start on 1 May.
c. All cheques drawn on the Association's bank accounts must be signed by any two of Chairman, Secretary, Treasurer and any person selected by the Management Committee for that purpose.
d. The accounts of the Association shall be audited each year by an auditor appointed by the Committee.
e. All income and property of the Association from whatever source derived shall be applied solely to the promotion of the objects of the Association.
f. No member of the Association shall receive payment for his services as a member.
g. The Management Committee shall appoint dates in the year to be membership renewal points.
h. Membership fees are due initially on joining the Association and subsequently at the latest membership renewal point in the following full year and on that date thereafter for the duration of the membership.

11. Dissolution

In the event of the winding up or dissolution of the Association any surplus assets remaining after discharge of liabilities shall automatically vest in the BCS.

Annual General Meeting of the BAA

Anthony Camacho, British APL Association
secretary@vector.org.uk

The meeting was held at 5 Southampton St on Thursday 19 June 2008. The chairman opened the meeting at 2:00.

1. Minutes of the 2007 AGM (as published in *Vector* and distributed to those attending) were approved *nem. con.* – proposed S.J. Taylor sec. A.J. Camacho.

2. The Chairman (Paul Grosvenor) reported that the committee had had to deal with the changes made by the BCS and had proposed a resolution, that we should separate from the BCS, which had been (as just announced) overwhelmingly supported by members. Now we have to carry out the work we resolved to do.

3. The treasurer (Nicholas Small) circulated summary accounts and explained some of the details. He reported that we have 290 individual members, though the BCS has failed to charge some of them for current membership. The accounts and membership report were accepted *nem. con.* – prop. Anthony Camacho, sec. Jane Sullivan.

4. The chairman proposed that the 2007-8 committee should be re-elected for 2008-9 and this was agreed *nem. con.* – prop. Richard Nabavi sec. Chris Hogan. The committee for 2008-9 is therefore:

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

5. Chris Hogan agreed to be proposed and was appointed auditor.

6. Richard Nabavi said that the regulations would need more changes than had been made by the resolution. The chairman asked him to make a suggested draft. Anthony Camacho offered to help.

7. Questions were then invited from the floor and three were asked:

- *What is happening to the BAA funds held by BCS?* Paul responded that the BAA would be making all efforts to recover any funds from BCS over the coming days but for the purposes of budgeting and running the association the committee was assuming a starting point of £0.
- *The regulations state that the EGM should be held in May (item 9a).* Paul agreed that this was the normal position however in the light of the EGM requiring 4 weeks' notice it had been decided to slip proceedings by 1 month to allow due process.
- *Would the BAA be holding a conference as previously planned?* Paul responded that in principle, yes, but this would depend upon the finances of the group over the coming months. It may be that a somewhat simpler format may need to be considered but this would be reviewed nearer the time.

8. As, now we are finally separated from the BCS, the association will again become a partnership with the committee as partners, Anthony Camacho said that there is a company (APL Projects Ltd), that was formed in 1987 and has never traded, that could be available for the association to use when it wished to undertake some activity for which a limited liability is required.

The meeting was closed at 14:19.

BAA Management Committee Meeting

Anthony Camacho, British APL Association
secretary@vector.org.uk

Prior to the EGM and AGM a short and informal meeting was held by the committee to discuss various issues arising from the announcements to be made later.

Specific points raised (in no particular order) were:

- Nicholas had investigated various bank options for the BAA and decided that the best was to open a Barclays Community Account. It was agreed he should proceed.
- Paul Grosvenor, Richard Nabavi and Stephen Taylor would negotiate the return of funds to BAA from BCS.
- The membership would be informed of the agreed changes by internet and an insert placed into *Vector* 23.3 which was due to go out any day now. Paul Grosvenor and Stephen Taylor would write the insert.
- Issue 23.4 of *Vector* is expected to be published in September (suitably modified to exclude BCS logo etc.). We would then issue subscription renewals to our members covering the membership year 1st May 2008 to 30th April 2009. This would give Nicholas plenty of time to arrange our bank account and put in place a mechanism to receive credit card payments.
- Our current fee structure would remain unchanged but may be reviewed later in the year.
- As *Vector* articles now appear promptly on the Web, before they appear in print, we would consider printing two, larger, issues per year rather then the current 4, so reducing costs.
- Richard Nabavi and Anthony Camacho agreed to propose new regulations.
- We discussed the possibility of printing an additional publication this year along the lines of an “APL Hints and Tips”. (It was suggested we prepare a collection of the At Play With J articles first.) Dynamic functions is a possible topic; we might invite a number of APLers to provide some short articles on “hints and tips”.

Industry news

Sustaining Members, British APL Association
editor@vector.org.uk

APL2000

The next User Conference will be 10-11 November 2008 at the Hyatt Regency in Bethesda, Maryland. Details and registration forms from www.apl2000.com or contact Sonia Beekman (sonia@apl2000.com) or on +1 (301) 208-7150. At press time the conference programme is:

- **What's new in APL+Win** *John Walker*. Highlights of the major new features and bug fixes in APL+Win since APL+Win 6. The features include improvements to the session manager such as the session property and `GetSessions` method, the graphical user interface `⎕wi` such as the `CloseDoc` and `OpenDoc` methods for the `Printer` object, the APL Grid print and print preview, the system interface such as the Zip class and other non-GUI related facilities such as the `W_Def` and `W_Reset` arguments to `⎕wcall`.

- **APL+Win interface to .Net libraries** *APL2000 staff*. This presentation describes a feature which can be used with APL+Win to create an interface to Microsoft .Net using a .Net assembly created by this utility. The .Net reflection namespace is used to display the methods, properties and events in a programmer-selected .Net assembly.

- **Grid computing using APL WebServices asynchronously** *Joe Blaze*. This presentation will provide a short overview of APL WebServices as a means to expose APL+Win software to web-connected users and machines. In addition, an example will be provided which illustrates how APL WebServices can be used to coordinate a scalable, asynchronous grid of processors to 'solve' amenable problems of a certain granularity, such as stochastic models, Monte-Carlo simulation, discrete-element analysis, etc

- **C# as the GUI and APL+Win supporting the business rules** *Ajay Askoolum*. This presentation will cover three subtopics:
  - C# using APL+Win as a COM server
  - A C# Windows Service using APL+Win as a COM server
  - Exposing a .Net assembly as COM for use by APL+Win

- **Reverse geocoding with APL** *Jeremy Main*. Using a public database of location names, examples and techniques for reverse location lookup (reverse geocoding) will be shown. Presentation will include several APL searching techniques, distance formulas, comparison presentations and database queries. Possible application areas will be discussed.

- **Interface APL+Win and .NET (C#)** *Eric Lescasse*. This presentation will demonstrate the various ways by which one can interface APL+Win and .Net (C#). This will include:
  - Consuming C# DLLs from APL+Win (using NetAccess)
  - Writing the application User Interface in C# and calling
  - APL+Win in the background
  - Porting your existing APL+Win application to .Net using C# and APL+Win
  - Porting your APL+Win application to Internet as a Client-Server .Net (C#) ClickOnce application using APL+Win on the Server
  - Writing ASP.Net (C#) web sites using APL+Win in the background and Ajax

- **Does APL make your Excel life easier?** *Kevin Weaver*. While Excel is reportedly the most widely used "language" for calculation-oriented work, there are many problems that have complicated solutions. APL can lend a hand in quickly working around these sticky issues. However, can APL *always* make your life easier? Sometimes Excel is the way to go… hard to believe?

- **Improved efficiency of execution of APL primitives** *APL2000 staff*. Executing an expression such as `a+b-c` in right-to-left order incurs costs due to storing and fetching intermediate results. By restructuring execution order we can reduce fetch-and-store overhead and increase execution efficiency. This presentation describes the APL+Win interpreter enhancements that have implemented this approach.

- **VisualAPL: ready for Visual Studio 2008** *Jairo Lopez*. This presentation will demo new development tools included in Visual Studio 2008 and how you can take advantage of these tools with VisualAPL.

- **APL application demonstrations** An opportunity for conference attendees to demo their APL applications for other conference attendees. Learn about how the benefits of APL are being leveraged to create applications across a wide range of industries.

- **Overview of APL2000 product pricing** *Sonia Beekman*. An overview of products and services available through APL2000 including a description of the APL+Win Subscription Program and VisualAPL pricing.

- **Overview of the APLDN Forum and Open Forum with the APL2000 team** This session will include a review of the procedure for communicating with APL technical support. The APL2000 Team will be available to answer questions and listen to comments and suggestions from conference attendees.

Dyalog Ltd

Dyalog Version 12.0.3 has been released:

- 32-bit Classic for Windows and AIX
- 32-bit and 64-bit Unicode for Windows and Linux

The main new features are secure socket communications for Conga, `⎕FCOPY`, and two new workspaces.

`APLIN`	imports mainframe APL2 workspaces exported via the `)OUT` command
`APL2PCIN`	imports PC APL2 or APL+Win workspaces exported via the `)OUT` command

Manuals for Version 12 are now available from Lulu.com, either printed to order, or as freely downloadable PDFs.

2008 conference programme

The Dyalog 2008 conference will be held at LO-skolen in Elsinore in October. Some highlights from the conference programme:

- **Demo of an ASP.NET application with a Dyalog engine** *Chris "Ziggi" Paul, The Childcare Company*. The LASER application is an online training tool for teaching NVQ levels 2, 3 and 4 to nursery staff. The system as an ASP.NET front end with Macromedia Flash plug-ins but is controlled and managed by a Dyalog APL.Net program.

- **Airline Revenue Management** *Maurice Jordan*. Airlines believe good Revenue Management enhances revenue by 5% or more. In an industry that struggles to return profits of even this magnitude, it has spawned a small army of consultants and its own mythology. The presentation will show a simple APL approach to "classical" revenue management. The underlying model for this classical formulation relies on the traditional complexity of airline fares, where very few people can navigate the range of fares on offer. Internet selling blows a hole in this model. The presentation will show how dynamic programming (sorry, nothing to do with d-fns) indicates that a simple modification to the input data allows the original algorithm to be used in this new marketplace. There are parallels in Financial Services and many other industries where there is a fixed resource to be allocated to products offering different levels of return (and risk).

- **Genetic Algorithms** *Romilly Cocking*. Romilly has been interested in biology-based Artificial Intelligence since the 1970s. He's recently started to follow up on his early AI research. After great frustration using Python and Java, he's now using APL again.

- **Heterogeneous development with maximum re-use of APL assets** *Lars Stampe Villadsen & Martin Petri, SimCorp*. The existing SimCorp Dimension APL codebase is huge so it is important that as much as possible can be re-used while using new features provided by .NET/C#. This presentation will make a live demo on how development in APL and C# can be done in parallel by adding features to existing core functionality while maintaining the integrity of the system as a whole.

- **OO for the elderly** *Dick Bowman*. This presentation will explore Dyalog's object-oriented features through a back door, following a path taken by one elderly APL bigot – it may interest others who have a similar background, or new-comers wanting to make fullest use of APL's functionality.

- **Gridifying FinE using the Techila Grid** *Claus Madsen, FinE*. FinE is a comprehensive set of advanced financial functions covering all aspects of risk management, valuation and analysis. FinE is a developer's toolkit; the core is shipped as a DLL. The purpose of this presentation is to show a practical and real-life embedding of the Techila Grid Technology into a financial commercial product.

- **PKZIP your files using APL and .NET** *Gianluigi Quario, APL Italiana*. ICSharp-Code.SharpZipLib.Dll is a .NET compression library that supports Zip files using PKZIP 2.0 style encryption, with GNU long filename extensions. It is written entirely in C# for the .NET platform. It is implemented as an assembly, and can thus easily be incorporated into other projects (in any .NET language). Two Dyalog APL functions are presented that create and use objects that are instances of .NET Classes derived from this library and other .NET base libraries. These functions allow the compression and deflation of Zip files.

- **ADOC** *Kai Jaeger, APLTeam Ltd*. With the introduction of OO in Dyalog Version 11, implementation details of a class remain hidden. The user of a class needs to know about and deal with only the public interface of a class. ADOC is a self-contained class designed to extract and report information about the public interface. ADOC is able to report public methods, fields and properties of any class, by request with detailed syntax information. But ADOC can do more than that: following a small set of simple rules one can put fully-fledged documentation into a class. ADOC is able to extract these pieces of information and create either an HTML report or an XML file from that.

- **Herding cats for fun and profit** *Joakim Hårsman, Profdoc Care*. Profdoc is a leading provider of healthcare information technology. Profdoc Care makes Profdoc HIS – a medical record system, and has grown from two to 13 APL developers in just a couple of years. Joakim was there for the ride and will talk about what had to change as the numbers of programmers grew, and how development at Profdoc Care works today. Joakim will address the following issues: why the idea that it's impossible to recruit APLers is nonsense; what changed as we got bigger; how to handle a ten year old code base weighing in at more than 300 000 lines, and remain nimble; the importance of tools, and the ones used at Profdoc.

- **Performance improvements in Dyalog** *Roger Hui, Jsoftware, Inc.* In the next version of Dyalog, some common boolean functions will be improved by factors ranging from 2 to 1600. We work through one particular example.

- **Serving lunch with Dyalog** *Tommy Johannessen, Jersie Data ApS*. The application is used by 25,000 school children and their parents for ordering and payment of school lunches. The lunches are produced in 25 kitchens located all over Denmark. Each kitchen has its own set-up and menu, and the application caters for the design, creation and running of individual kitchen websites. The presentation will focus on how this success story started and demo the various aspects of the application. The technical aspects will focus on how we created the `.aspx` files, the file structure, backup procedures, communication between the servers and the call structure.

- **The Array Constraint Engine** *Gert L. Møller, Array Technology*. Array Technology provides technology for solving complex constraint problems in real-time on a very small memory footprint. The technology is used in a range of business applications, e.g. for product configuration with many business rules or constraints. The key to the performance of the technology is the use of nested arrays (array-based logic) for handling logical constraints with a multitude of combinations. Dyalog 8.2 was used for prototyping the first version of the technology, but Gert will present how the next generation of the technology will benefit from the power of the latest Dyalog releases.

- **Snooping with APL** *Charles H. Brenner, Ph.D., Consulting in Forensic Mathematics*. Charles is a world-recognised authority in Forensic Mathematics – covering complex areas such as DNA identification, biostatistics, and population genetics. His DNA•VIEW™ APL-based software solution is the acknowledged leader and is the standard worldwide for DNA identification. The software has been used in countless cases amongst others the World Trade Center victim-identification work, Tsunami victim identification in Thailand, mass identification projects including *desaparacito* children from El Salvador, and war victims in Bosnia.

- **COPA-MS – A look under the hood** *Michael Baas, DLS-Planung.de*. Comanufacturing Management System (COPA-MS) is an application based on the Hologram BI-System from Dyalog's Australian distributor Hologram Pty Ltd. Last year, this project was mentioned in Michael Baas' & John Miller's talk on networking as the first fruit of the collaboration between Hologram and Dynamic Logistics Systems GmbH.

- **OOStats – Performing statistical calculations using Dyalog** *Alan Sykes, Acadvent Ltd*. The advent of the object-oriented features in recent Dyalog Interpreters provides a new and exciting framework within which to construct software to analyse data. The talk will demonstrate statistical objects that allow an APL user to perform statistical calculations on realistic data sets that may well contain missing values. All statistical functionality is made available either directly from user commands from the session, or, for more speedy data exploration sessions where a variety of alternative analyses might be envisaged, by using a menu-driven GUI version.

- **Pocket APL** *Ray Cannon*. Notes on developing a system that runs under Dyalog PocketAPL, with two example applications (GPS and Su-Doku), and some utilities making `⎕NA` calls to the underlying operating system.

The conference will be bracketed by training days, plus a couple of workshops on the Tuesday:

- *Sharpening Your APL Knife* Kai Jaeger
- *Source Code Management using SALT and SubVersion* Dan Baronet
- *Migrating to Unicode* Morten Kromberg
- *Fast-track your GUI Design* Adrian Smith
- *Using the Microsoft .Net Framework* John Daintree
- *Introduction to Object-Oriented Programming* Daniel Baronet
- *Web Creole* Stephen Taylor
- *Conga & SSL* Morten Kromberg

- *RainPro* Adrian Smith

Hologram

Dyalog has appointed Hologram Pty Ltd as a new distributor for Australia and New Zealand.

Hologram Pty Ltd is an Australian Business Intelligence company founded in 2004 to develop and deliver future-proofed Business Performance Management solutions. Hologram's founders have an extensive track record as developers of highly successful treasury risk-management and financial-reporting software packages.

Hologram's attention to detail, meticulous programming skills and ability to solve the most complex mathematical problems are invisible to end users, who benefit from ease of use, point-and-click drill down, and instant access to real-time modelling and reporting from enterprise systems.

The company has a strong focus on manufacturing and financial institutions, with customers including leading beverage companies, credit unions, merchant banks and leasing companies.

In connection with the appointment, Managing Director Gitte Christensen says, "I am absolutely delighted that Dyalog is partnering with Hologram. The company's extensive experience in array language development and applications in the finance, banking and treasury industries makes it a natural and easy fit with Dyalog."

Hologram's business focus includes high-performance, real-time transaction-monitoring and data-information systems. The company's principals have an established track record of working with banks, exchanges and credit unions in development and consultancy in areas such as treasury systems, anti money laundering, risk management, compliance reporting, portfolio management and trading systems. They have worked with institutions around the world including the Bank of South Australia and Société Générale, as well as the stock exchanges of Singapore, Indonesia, Istanbul and Oslo.

Kx Systems

Charles Skelton to take over as CTO

Financial industry veteran Charles Skelton will take up the position of Chief Technology Officer on 1 November, replacing Niall Dalton.

Prior to joining Kx, Charles Skelton was the owner of and principal consultant at Skelton Consulting GmbH, a product development and software consulting company specialising in financial-markets technology. Skelton has been working closely with Kx for a number of years; his company's focus was on advanced market data capture and analytics using Kx's kdb+. Skelton has expertise in writing feed handlers, as well as in-depth experience in setting up very large, high-performance, infrastructure market-data systems for tier-one banks. Skelton's responsibilities will include working with Kx's development team on product development, enhancement and support, focusing on new features; he has a proactive style and intends to be in regular and close contact with Kx's client base.

Dalton will relinquish his current position of CTO at the end of October, but will remain with Kx in a consultancy capacity. This new role will allow Dalton greater flexibility to pursue some of his other technology and personal interests.

Says Janet Lustgarten, CEO of Kx: "I am delighted that Charles is joining us. We have been working with him for many years so he has an intimate knowledge of kdb+ and q, and already knows many of our clients. Having been a client of Kx, Charles has been on the 'other side of the fence' and has a great deal of insight into what our clients need from us. It is great to have Charles as part of our team, and I'm very much looking forward to working with him even more closely than before. I would also like to thank Niall for all of his fantastic work at Kx. Niall has always made it clear that he had other interests outside of Kx that he wished to pursue at some stage and I am pleased for him that he is now taking the time to do so. Importantly, his leadership in creating a community environment within Kx has and will continue to be of tremendous benefit to our clients."

Skelton spent ten years in the telecoms industry as a consultant developing real-time software for large multinationals before moving into technology for financial institutions. During his many years in this market Skelton has assisted numerous institutions such as JP Morgan, HypoVereinsbank and Deutsche Bank with their deployment of kdb+. He has provided consultancy to firms based in the US, Germany, the UK and Spain.

"I was fortunate to have worked with Kx on kdb+ applications for a number of years and have been an active member of the Kx community, often presenting at the annual Kx conferences. I am very excited with this opportunity to be more involved with Kx and to use my expertise for the benefit of all Kx clients," Skelton says.

Kx growth continues with new partner in Australia

Kx Systems has signed a partnership agreement with Hologram Business Intelligence, the Australian-based supplier of business performance management solutions.

The relationship between the two companies goes back a long way and means that they already have an in-depth understanding of each other's products and businesses. The deal will see Hologram taking on the sales, support and consultancy of Kx's Kdb+ database and its q language in the Asia-Pacific region. Kx products may also be incorporated into Hologram's offerings in the future.

Hologram's extensive experience in array-language development and applications in the finance, banking and treasury industries makes it a natural and easy fit with Kx. Hologram's business focus includes high-performance, real-time transaction-monitoring and data-information systems. The company's principals have an established track record of working with banks, exchanges and credit unions in development and consultancy in areas such as treasury systems, anti money laundering, risk management, compliance reporting, portfolio management and trading systems. They have worked with institutions around the world including the Bank of South Australia and Société Générale, as well as the stock exchanges of Singapore, Indonesia, Istanbul and Oslo.

Janet Lustgarten, CEO of Kx Systems, comments, "We are very much looking forward to working with Hologram and extending our product reach to include Australia and New Zealand. Their thorough understanding of our language and database means that they will be able to help financial institutions in their region make full use of the benefits provided by Kx products. I'm excited about the opportunities offered by the partnership and of extending the Kx product family to the region."

Says Guy Pitman, founder and director of Hologram, "We see great potential for Kdb+ and q, particularly in the areas of real-time data analysis and high-end database engines for data-intensive applications in finance, fraud, banking and other high-volume transaction systems. With two offices in Australia and eyes firmly set on New Zealand and Asia we are exceptionally well positioned to build momentum and growth around Kx's products." Pitman adds, "I believe that high-end transaction processing is currently less sophisticated than it could be in Australia and New Zealand. Working with Kx will allow us to offer an exceptional service to institutions operating in this field."

Enhanced multi-core Kdb+ includes DTrace and more speed

Kx Systems has announced a new version of Kdb+. With the progress in hardware development and rising data volumes it is essential that software is able to not merely keep up with progress, but make the absolute best use of hardware improvements. This is why every new version of Kdb+ is described as being even faster than the previous version – because it is.

From the outset Kdb+ was specifically designed and optimised for multi-core capability, and can handle all machines available on the market. The soon-to-be-released v.2.5 benefits from even faster multi-threading as well as a number of other significant enhancements including DTrace support, optimised intelligent memory allocation, and a number of new interfaces. Now available to customers for final testing and evaluation, v.2.5 will be noticeably faster for queries, especially on large databases.

Support for DTrace, a powerful infrastructure for analysing the allocation and usage of resources on large servers running Solaris, is now available in Kdb+. Some of Kx's largest clients, including many tier-one banks, now see support for DTrace as non-negotiable in their core infrastructure products and critical services such as Kdb+.

Kx recognises that institutions are facing growing pressures on their resources. A number of enhancements in v.2.5 have been made to help financial institutions address some of those issues. DTrace allows very detailed (low-level) monitoring which does not modify code and helps to track down bottlenecks. More efficient threading means that large partitioned databases will be considerably faster. In addition existing interfaces to Java, C#, C, C++ and Excel have been enhanced, while interfaces to R and F# have been added.

“Intel is focused on the massive compute workloads across the trading lifecycle – from market data, through analytics and risk to trade routing,” said Nigel Woodward, Intel Global Financial Services Director. “Throughput and low-latency at every stage are critical, which is why Intel works closely with Kx so that Kx's software structure and Intel's parallel processor infrastructure combine to provide a competitive advantage to our mutual clients.”

Says Simon Garland, chief strategist at Kx, “Our customers expect Kx to be the fastest. To achieve this we constantly work on optimisation and ensure that Kdb+ is able to make use of the latest developments in technology. We are also very conscious of the need to be good citizens when using resources: where several years ago a vendor might have expected to have exclusive use of a machine that is no longer the case. The new version of Kdb+ has been further optimised to allow our clients to get the most out of their new hardware.”

DISCOVER

Fire from heaven

q for Mortals, Jeffry Borror, Continuux, New York, 2008, 478pp

Adrian Smith
adrian@apl385.com

To learn a new language, you need an environment to fool around in, and a good textbook. The Kx people provide the former at kx.com [1] where you can grab a time-limited copy of kdb+ for personal use, and Jeffry Borror has provided the second. In the good old days, we had a mainframe to fool with and Gilman and Rose to read – maybe those days are here again? This review will document my attempt to find out.

You can judge as we go if prior knowledge of APL and a smattering of SQL (lightly dusted with Paul Mansour's flipdb engine) is a help or a hindrance. My feeling is that it will help with the array-thinking part and hinder with the database query language. I will probably trip over the q keywords in a big way – I already tried to use `ss` as a scratch table name and there are bound to be others in the pipeline. Fortunately, Arthur has not yet used `qq` to mean anything important!

Getting started

Just as `helloworld.c` is the hardest C program you will ever write, `2+2` is generally the hardest expression in any of the APL family. Once this works, you have cracked the installation, got your laptop working again, and generally calmed down. In this respect, the free kdb+ is much better than most, but it does have a couple of minor annoyances:

- Please will someone decide whether this thing is called q or kdb+. For the newbie it is not particularly clear that *personal kdb+* is what you need to run the q examples.

- The install suggests unzipping under `c:\q` without the option – I tried `d:\tools\q` and of course it died on me. In fact you can set `QHOME` to be anywhere you like, but the readme ought to tell you!

Either way, you will probably want a tiddly batch file like:

```
@echo off
Rem Kick off q from anywhere with optional script
set QHOME=d:\Tools\q

if ""=="%1" goto clear
d:\tools\q\w32\q.exe %1.q
goto exit

:clear
d:\tools\q\w32\q.exe

:exit
```

So you can hang about wherever you put your toy scripts and just type `q fluff` to kick off the q engine with your script already loaded. Here we go…

```
D:\>q
KDB+ 2.4 2008.03.31 Copyright (C) 1993-2008 Kx Systems
w32/ 1()core 502MB Adrian blue 192.168.2.103 TIMEOUT 2009.04.01

q)2+2
4
q)\\

D:\>
```

Gas, are we cooking with… let's try something a little more challenging:

```
D:\data\q>q sp
KDB+ 2.4 2008.03.31 Copyright (C) 1993-2008 Kx Systems
w32/ 1()core 502MB Adrian blue 192.168.2.103 TIMEOUT 2009.04.01

q)select from sp where qty>200
s  p  qty
---------
s1 p1 300
s1 p3 400
s2 p1 300
s2 p2 400
s4 p4 300
s1 p5 400
```

I think that concludes the first phase – now to get moving on the review proper.

Overview and language basics

One of the hardest choices an APL author has to make is the ordering of the material. Jeffry is aiming this at programmers (which probably includes anyone who has read any kind of science subject at university these days) so he starts with a lot of things his readers will already know.

Functions and atoms

This sets the tone for the rest of the book – Jeffry is kind but firm and makes no bones about the need to understand *functions* in the mathematical sense. There are a couple of typos here – the missing space in the code example (bottom of p.8) is a bad one. Yes it is trivial, but it undermines the reader's faith in the code having been pasted from a running system. I am sure it will be fixed in the next printing.

I like the constant distinction between the *q gods* who write perfect code every time (and have no need of whitespace or comments) and *mere mortals* who should use meaningful names and split complex operations over multiple lines. I have even taken to writing `//` for comments, partly to save finger reprogramming, and partly to make TextPad's syntax colouring work properly! I also like the little sample program Jeffry shows us right at the beginning – as he says "We promise that by the time you reach the end of this tutorial, this program will be easy, and you'll feel right as rain."

Atoms are the basics of any language, so it makes sense to introduce them early. Jeffry does his best to put 'verbose language' programmers at ease by starting with a clear table of equivalences from what they already know. I *think* I would label the 4th column '.Net' rather than 'C#' though – C# coders would always say `bool x;` rather than `System.Boolean x;` but of course `int` in C# can mean Int32 or Int64 depending on the platform, so he is right to quote the more pedantic style of name. All the types are listed with simple examples, and this feels like a few pages which will get well-thumbed on the odd occasion you actually need `byte` data and can't recall the suffix. I am still a little unclear why a *symbol* isn't a legitimate implementation of a *string*, as my C# brain is happy with the idea that strings are immutable. I suppose they do have *Length* and the `String` class does support an indexer, but that's about as far as it goes. The table of infinities and the section on *null values* is informative and helpful. Database gurus go hairless arguing about nulls, so it is good to see q taking a rather pragmatic approach which will work fine in all sensible situations.

Lists

APLers pay attention – this takes Trenchard More's array theories and hits them well onto the next court but one. These be *Lists* and Jeffry is very consistent in calling them *Lists* just to keep us awake. That said, the only surprise is that atoms have a count of 1 and that we can skip the brackets in indexing expressions (indexing is just a function, after all). The more verbose notation for lists is used consistently so we have `l1:(1;2;3)` where `l1:1 2 3` will generally work in the same way. This makes sense as soon as you hit a nested list like `m1:(1 7;2 9;8 9)` where our APL habits would be `(1 7)(2 9)(8 9)` and would lead us astray. Indexing is very nicely brought in here, with the classic 'matrix' syntax:

```
q)m1[1;1 0 1 0 1]
9 2 9 2 9
```

being approached via the idea of indexing at depth with `m1[1][1 0 1 0 1]` which gives exactly the same result. By the end of Chapter 3 I think most readers should have grasped the basics, and will probably have a well-used q console with lots of 'I wonder what this does' expressions in it. They should take a well-earned break to let it sink in.

Primitives and functions

The primitives are no great surprise to an APL guy, but it is worth paying careful attention to the sections on *nulls* and *infinities* as well as the extensions to dates and times. Jeffry takes the 'no precedence' rule on the chin nice and early, and gives plenty of good examples of the sort of expressions that can puzzle anyone brought up on the C execution order. He also flags up a key difference from APL-derived languages (including k) in that there are no overloads on *valence* (so no monadic `-`) but there are overloads on *type* in functions such as `?`, for example:

```
q)3?5
3 2 2
q)1 2 3?3 2
2 1
q)?12
'?
```

User-defined functions follow on nicely, particularly as Jeffry already showed us that we can type `+[2;3]` or even `(2+)3` in the previous chapter. He again starts with the most verbose form, and gradually eliminates the redundant parts as the examples progress.

```
q)sqr:{[x] x*x}
q)sqr 3
9
q)sqr:{x*x}
q)sqr 4
16
q){x*x} 12
144
```

I think I would like him to make it clearer that I can't have a user-defined dyad, and the rules for what is returned are explained very oddly on p.100 – I think that a lone `:` reads as *return* and that otherwise the result of the function is the result of the last expression executed. I would also like to see a big fat warning about this one:

```
q)sqr:{r:x*x;}
q)sqr 3
q)
```

If you have spent any time in the C family (which includes scripting tools like PHP) you get a bit of a semi-colon reflex in your fingers, which often results in an empty statement as the last one in your function. I am not sure about the claim that 'functions are nouns' in 4.1.6 – for example:

```
q)plus:+
q)plus[3;4]
7
q)incr:plus[1;]
q)incr 12
13
q)0 plus/8 9
17
```

Surely the primitive `+` and the user-defined function `plus` have the same syntactic status in the language? If it walks like a verb and quacks like a verb, I say it's a verb. Shame you can't use it with *amend* though. Maybe it's only a verb when it feels like it!

The section on *function projection* has no such quibbles, and I found the section on *adverbs* simple and clear. The final section on *index, apply, dot* feels a bit hard – if it was in the TeX manual it would have 3 'dangerous bend' signs. Probably essential reading for the serious q systems developer but I'm afraid your reviewer started turning over pages in search of something less strenous.

Casts and enumerations

Something in me wonders if this shouldn't come a little earlier, probably after the chapter on lists. Casting is something that programmers are very used to, and there is nothing very strange about the way q handles it. Enumerations are just factored lists, and although the process will take some getting used to, there is nothing very startling here either.

The serious stuff – dictionaries and tables

This is where the power of q cranks up a notch. Anyone with an APL or J background has probably been saying "so what?" up to this point, although we can hope that C++ and VB kids may have got excited by now. The next two sections are the reason q has been taking the world of timeseries database by storm, as they extend the raw language into the domain of database programming, but without the limitations imposed by years of SQL thinking.

Dictionaries – a statement of the bleedin' obvious?

Open any Javascript or PHP book at the section on *arrays* and you will find something like this:

> You can assign an index when using the `array()` function as follows: `$list = array (1=>"apples",2=>"bananas",3=>"oranges");`. The index value you specify does not have to be a number, you could use words as well. This technique of indexing is very practical for making more meaningful lists.

That was from *PHP for the World Wide Web, Visual Quickstart Guide, 2001* but anything will do. The point is that APL (and J) purport to be array languages, and neither of them support something that most modern scripting tools take as a given. In q we find that we can type:

```
q)fruit: (1 2 3!`apples`bananas`oranges)
q)fruit 2
`bananas
q)fruit 2 1 1 3
`bananas`apples`apples`oranges
```

We can even add new elements (again like PHP) by indexing with a non-matching key:

```
q)fruit[23]:`pears
q)fruit
1 | apples
2 | bananas
3 | oranges
23| pears
```

Jeffry takes *dictionaries* at a steady pace, and I am fairly sure I came out of this section with a clear feel for what they are and how I could use them. The section builds nicely via *column dictionaries* where the content has more than one value (these examples are my maunderings in the q session, by the way. The fact that I got all of them right at the first attempt is a tribute to both q and Jeffry):

```
q)emp:(`id`name!(1001 1002;`Adrian`Richard))
q)emp
id  | 1001   1002
name| Adrian Richard
q)emp[`name]
`Adrian`Richard
q)emp[`name;0]
`Adrian
```

… and as if by magic, we arrive at *tables* and a whole new world lies before us:

```
q)et:flip emp
q)select id from et
id
----
1001
1002
q)select from et where id=1001
id   name
-----------
1001 Adrian
```

I think I would like a little more advice from a systems-design angle on when to use a *dictionary* and when to use a *table* in constructing a real life application. There are so many helpful tools in q to make working with tables simple and painless that maybe Jeffry has told us rather more about dictionaries than we really need to know? I guess we will have to await his next book to find out!

Tables like you never seen 'em before

At this point you can swap in your SQL brain and take on a few more surprises. What is most appealing is the way that anything that acts on a table returns a table, so expressions can build on each other:

```
q)select name from select id,name from et
name
-------
Adrian
Richard
```

This would make Chris Date [2] *very happy indeed*. Having been provoked by Paul Mansour into reading most of the C.J. Date books on advanced database design, I suspect q complies with virtually all his criteria for a *true* relational database, which no traditional SQL-based system does. The other big bonus is that you can do arithmetic directly on the columns. I know standard SQL has some pretty cool `Alter table` stuff but you can only do the things the SQL designers thought of at the time. In q you can do anything Ken Iverson thought of, which adds the full gamut of array power to table syntax:

```
q)et.xx: (+\)et.id
q)et
id   name    xx
-----------------
1001 Adrian  1001
1002 Richard 2003

q)add2:{x+2}
q)et.xx: add2 et.id
q)et
id   name    xx
-----------------
1001 Adrian  1003
1002 Richard 1004
```

So simple to create a scratch column using a bit of k to accumulate the values, or to apply our own (incredibly complex) data-processing expression encapsulated in a function. I begin to see why Richard came back from Cantor Fitzgerald with a big grin on his face – algorithmic work against databases does begin to look very simple when you can write code in this way.

Primary keys and keyed tables

Time to concentrate a little – we arrive at section 7.4 and a cup of strong coffee is called for. Up to this point Jeffry has led us by the hand through green pastures, from now on in the path winds uphill, and you might find yourself coming back here several times as you begin to build serious applications. The syntax for creating and indexing a keyed table is clear enough:

```
q)et:([id:1001 1002] name:`Adrian`Richard;pay:1234 12345)
q)et
id  | name    pay
----| -------------
1001| Adrian  1234
1002| Richard 12345
q)et 1001
name| `Adrian
pay | 1234
```

which brings us (via multiple keys and other stuff) to section 7.5 where we hit *foreign keys* and *virtual columns*. I found this fairly comfortable reading, but I have been immersed in database design for longer than I care to remember [3] and was responsible for lots of APL utilities for handling just these concepts in my Rowntree years. I think that this section could really use some diagrams – even for the simplest toy database, I find myself reaching for the back of the nearest envelope and drawing boxes on it! Essentially the foreign keys define the lines on the diagram, and enforce 'referential integrity' meaning that you can't have an employee working in a department that doesn't exist. In q we find that we are revisiting the *enumeration* which is the construct that implements the 'refers-to' or 'is composed of' database semantics:

```
q)dp:([id:23 34] descr:("Op Research";"General Dogsbody"))
q)dp
id| descr
--| ------------------
23| "Op Research"
34| "General Dogsbody"
q)et:([id:1001 1002] name:`Adrian`Richard; dept:`dp$34 23;
                                            … pay:1234 12345)
q)et
id  | name    dept pay
----| ------------------
1001| Adrian  34   1234
1002| Richard 23   12345
```

So far we have something very like the toy database in my experiments with `System.DataSet` [4] but in q you can take things one step further by using the *dot* notation to look down the chain of relationships without having to write lots of obscure *join* syntax in the SQL:

```
q)et
id  | name    dept pay
----| ------------------
1001| Adrian  34   1234
1002| Richard 23   12345

q)select name,dept.descr,pay from et
name    descr              pay
--------------------------------
Adrian  "General Dogsbody" 1234
Richard "Op Research"      12345
```

One of the 'tough challenges' you are set in the Oracle training programme is "Find the employees who earn more than their manager". With the addition of an appropriate column to our department table, this sort of inter-table cross-referencing becomes quite trivial:

```
q)dp:dp,'([] mgr:`et$1001 1001)
q)select id,name from et where pay>=dept.mgr.pay
id   name
------------
1002 Richard
```

Of course there is a lot of advanced stuff on tables that I can skip over here – refer to it when you need it – but the basics are simply explained, and my experience is that by tabbing over to a q session and following along (with examples of your own) you will 'get the drift' very quickly. Next up is 80 pages telling us a lot more about q-sql which I think I am going to enjoy. Time to make a little script out of those *emp-dept* examples so I can keep fooling with it after my free q has timed out.

Working with queries in q-sql

This is very plain sailing, up to the point where you hit *Grouping and Aggregation* which deserves close attention. In SQL these are tightly bound up, whereas q-sql gives you the option to preserve the content of the groups in the resulting table. Paul Mansour copied this in flipdb and used his Minnowbrook session to show us lots of nice examples of 'hard' problems that just fall out if you have some set functions to hand. A trivial example could be:

```
q)select pay by dept from emp
dept| pay
----| -----------------
23  | ,23451
34  | 12345 32141 51324
```

Note how q gives us a heavy hint that the singleton is a 1-element list, *not* a scalar here. There may be a way Dyalog could discriminate between these with the session syntax colouring (hint, hint) these days, as the visual similarity often leads newbies astray. Of course you can also throw in your own code here:

```
q)select {(sum x) % count x}pay by dept from emp
dept| pay
----| --------
23  | 23451
34  | 31936.67
```

This one just reproduces the built-in `avg` keyword, but there are plenty of other things you could do here, like quartiles, which q-sql doesn't support directly. Entire queries can be 'canned' with the usual function syntax. (Jeff calls these *parameterized queries* but they just look like functions to me.) For example:

```
q)q1:{select name,pay from emp where id in x}
q)q1 1003 1004
name pay
----------
Gill 32141
Tim  51324
```

Of course you would normally name the argument(s) here (as Jeff does in the examples) to make things clearer. *Views* are implemented using the underlying *alias* syntax (which I'm sure I recall seeing somewhere in earlier chapters):

```
q)v1::select Name:name,Department:dept from emp
q)v1
Name    Department
------------------
Adrian  23
Richard 34
Gill    34
Tim     34
```

Finally, we hit the *functional forms* of both *select* and *exec* (which returns the underlying data rather than a table, incidentally) which are what q-sql parses your statements down to before it runs them. Being able to call these forms directly can be essential if the user can build the query dynamically in some fancy front-end. It saves a huge amount of hassle creating the query string (with appropriate string escapes) that we APLers have had to face for years when talking to DB2 or Oracle. It all looks pretty hairy in the examples, but I'm sure that with a little practice you can write these expressions as comfortably as you can write the *select* templates. Let's tab over to the q session and have a try…

```
q)?[emp;();0b;`Id`Name!(`id;`name)]
Id   Name
------------
1001 Adrian
1002 Richard
1003 Gill
1004 Tim
q)?[emp;(enlist (in;`id;1001 1002));();`name]
`Adrian`Richard
```

There we are, that wasn't *so* hard, was it now? I am not clear how it knew that the first expression was a select and the second one an exec, but I'm sure some q minor deity will explain it to me if I ever really need to know. Time to skip over 13 pages of 'things to do with a trading system' and move on!

Loops, files, namespaces and other matters

Yes, you can write boring procedural stuff in q but at least you don't get told how until right at the back of the book. Error-trapping and debugging support (there isn't any) rate a couple of pages, as do scripts and startup parameters. Finally (page 301) we get to read and write files, parse .csv input, and chatter with other q processes over the network. This is clearly how trading systems are written in the real world (lots of little tasks watching feeds and nattering to each other) and I don't think I am competent to say how well this section of the book works as an introduction. I had less trouble with the section on *contexts*, although it appears from some of the warnings that these are not quite all they appear, and you may want to keep your code only one level deep!

Finally we get the usual summary of system commands and variables, and a couple of appendices list all the functions and the rather minimal set of error messages. The index looks pretty thorough, but I have yet to give it a decent test.

Summary

This book is hard to fault. It has taken me on a very well-planned exploration of the strange land of q and I feel that I could already find my way around quite a lot of it unaided. I also know where to look to remind myself of the more obscure parts of the language that will never stick in the brain until I need to use them for real. Anyone with an interest in the APL language family should probably get a copy, if only to keep them alert to possible future extensions in the APL of their choice! I shall be agitating for *dictionaries* at future Dyalog conferences, and I might revisit my experiments with the .Net DataSet class to see how much of the q-sql syntax it is possible to fake. Maybe a few exercises would be a good addition, although it is easy enough to make up your own challenges as you go along. If there is another book on the way, I will be first in the (as it were) queue to get my hands on it.

References

[1] http://kx.com/developers/license2.php

[2] *Database in Depth*, C.J. Date, 2005, O'Reilly

[3] "Structuring Data with APL", Adrian Smith, in *Proceedings of APL Business Technology 83*, p175

[4] "Using the .Net DataSet with Dyalog 12", Adrian Smith, *Vector*, 23, N°3, 2008 p89

First experiences with Unicode in Dyalog 12

Adrian Smith
adrian@apl385.com

Abstract
This is a first summary of the experience I have had in moving the bulk of my daily schedule across to Dyalog 12 (Unicode edition). The little traps this process set you in no way invalidate the decision to move, but they can really spoil your day if you don't take good care of them. I will take things mostly in order of importance, starting with possible application crashes and moving on to minor presentational annoyances which can be fixed over a longer time.

Things to test in the interpreter

In spite of a long beta period, it was impossible to catch all the 'funnies' that the interpreter came up with. There may well be some gremlins left, for example I very recently discovered (when updating the source for one of my help files) that when I selected some text and hit Ctrl+B it was correctly wrapped with <b>tags</b>, but when I did the same with Ctrl+I nothing appeared to happen. Hmmm – then when I just tried to type the tag manually into the edit field, the first character I typed magically produced the 'i' for me.

Along the same lines, I was teaching a new Dyalog user the basics of CPro, and we got to the point where I needed to type a typical indexing expression such as `inx⊃myvec` into the dialogue designer. That was when I found that AltGr+C and AltGr+V were firmly wired to Copy/Paste and completely ignored my keyboard definition!

This sort of thing is *almost* impossible to test for, and I'm sure there will be other similar little annoyances. The moral of this story is to believe what you see (even if you only see it once) and let someone know about it. Dan (Baronet) is the man who can repro pretty well anything if you give him a good few clues where to look.

Things that may break your application

Things they tell you about

There are two places code has to change: any `⎕NA` calls will need some revision, and any calls to `⎕DR` to check for character data will definitely not work any more.

Most `⎕NA` calls can be fixed automatically with the new star syntax, for example:

```
     ∇ ret←GetCurrentDirectory;func;buffer_size
      ⍝ Cover for the windows GetCurrentDirectory function
      ⍝ Returns the current directory name
       buffer_size←10000
       'func'⎕NA'U4 kernel32.dll.C32|GetCurrentDirectory* U4 >0T'
       ret←func buffer_size buffer_size
       :If 0≠⊃ret ⍝ Function OK
         ret←2⊃ret
       :Else
         ret←''
       :EndIf
     ∇
```

This will call the correct DLL function (`GetCurrentDirectoryA` for Dyalog 12, or `GetCurrentDirectoryW` for 12-unicode) and so you can share these functions between interpreter versions. The only place I got snagged was with:

```
     ∇ {RC}←HANDLE PutString NV;SUBKEY;VALUE;RegSetValueEx
      ⍝ Stores a text value in a Registry SUBKEY
      ⍝ HANDLE is the handle for an existing Registry Key
      ⍝ Function by JS based on  SE.WSDoc.GetRegKeyValue
       (SUBKEY VALUE)←NV
       ⎕NA'I ADVAPI32.dll.C32|RegSetValueEx* U <0T I I <0T I4'
       RC←RegSetValueEx HANDLE SUBKEY 0 1 VALUE(1+2×⍴,VALUE)
     ∇
```

This is the Unicode version – spot the `2` in the last line! I spent quite a while wondering what was eating my registry settings (e.g. the last saved file in the Rain viewer) as every time I read the settings and re-saved them, the length halved!

Checking for numeric data-type used to be done like this:

```
      1=0=1↑0⍴2 3 4
1
```

… with something very similar for character. Maybe old ways are best:

```
      ⎕dr 'Fat cat'
80
      ⎕dr 'Let’s try ∑x also'
160
```

I had various bits of code looking for `82` here, and of course things tended to crash some way down the track, as the test ‘works’ but returns the wrong answer. I suggest:

```
      ' '=1↑0ρ∊'Let’s try ∑x also'
1
```

This should keep working for ever! It also gives you a good way to test if you are in a Unicode interpreter:

```
      unc←80=⎕dr ' '
```

Things you find out the hard way

The only real nasty I have hit so far is with native files – the problem here is that you may have a long-running logfile (saved out as plain text) to which you occasionally append messages.

```
      'c:\temp\unc1.txt'⎕NCREATE ¯1
      ('Hello',⎕av[4 3]) ⎕nappend ¯1
      ('World',⎕av[4 3]) ⎕nappend ¯1
      ⎕nuntie ¯1
```

This may have started life several years ago in a previous version of APL, and to start with all seems fine when you move your application:

```
      'c:\temp\unc1.txt'⎕ntie ¯1
      ('Wow, here we are again',⎕ucs 13 10) ⎕nappend ¯1
      ('How about x²',⎕ucs 13 10) ⎕nappend ¯1
      ('How about ∑x² then?',⎕ucs 13 10) ⎕nappend ¯1
DOMAIN ERROR
      ('How about ∑x² then?',⎕UCS 13 10)⎕NAPPEND ¯1
     ^
```

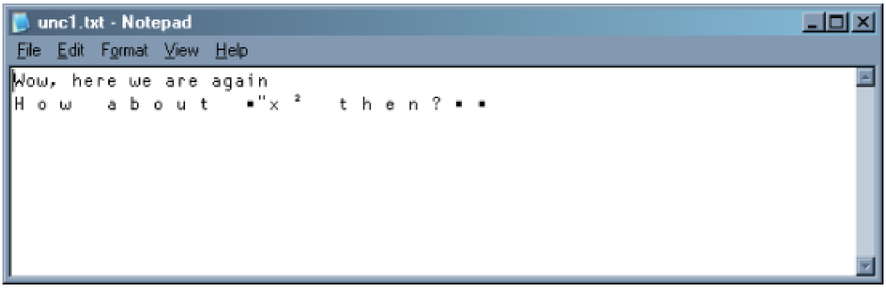
This can strike at any time, if you allow the user to edit the log message, and the user has the ability to type some symbol that falls outside the base 256 characters. Even if you clear the file down, you still have the problem unless you always specify the translation:

```
 0 ⎕nresize ¯1
('Wow, here we are again',⎕ucs 13 10) ⎕nappend ¯1 160
('How about ∑x² then?',⎕UCS 13 10)⎕NAPPEND ¯1 160
```

… and you have to be really careful not to mix translations in the same file:

```
0 ⎕nresize ¯1
('Wow, here we are again',⎕ucs 13 10) ⎕nappend ¯1
('How about ∑x² then?',⎕UCS 13 10)⎕NAPPEND ¯1 160
⎕nuntie ¯1
```

This appears to have 'worked' but if you open the file in Notepad, it looks like this:



```
Wow, here we are again
H o w   a b o u t   ▪"x ²   t h e n ? ▪ ▪
```

Analysing such a file at some point in the future will be a nightmare. So what to do? I have a couple of 'old faithful' utilities for writing and reading text files, so these were the obvious place to start:

```
∇ {r}←fi Putcrv txt;fh;cr;lf;msk;pos;⎕TRAP
 ⍝ Put a ⎕TCNL vector to file <fi>
 ⍝ Returns 1 if OK; errors if failed.
  ⎕IO←1 ⋄ r←0 ⋄ (lf cr)←⎕AV[3 4]

 ⍝ Try for existing file (includes 'prn' etc.) ...
  ⎕TRAP←(22 'E' '→New')(19 'E' '⎕SIGNAL ⎕EN') ⋄ fh←fi ⎕NTIE 0

 ⍝ We cannot tell if this is a printer,
   … so if the resize fails just carry on!
 Resize:⎕TRAP←22 'E' '→Append' ⋄ 0 ⎕NRESIZE fh ⋄ →Append

 New:⎕TRAP←22 'E' '⎕SIGNAL ⎕EN' ⋄ fh←fi ⎕NCREATE 0

 Append:⎕TRAP←22 'E' '⎕SIGNAL ⎕EN'
 ⍝ Pair CR/LF from ⎕TCNL
  msk←txt=cr ⋄ pos←msk/⍳⍴msk
  txt←(1+msk)/txt
  txt[1+pos+0,+\¯1↓(⍴pos)⍴1]←lf

  :If 160≤⎕DR txt ⍝ Make it UTF8 - and mark it with ef bb bf
    txt←⎕UCS 239 187 191,'UTF-8'⎕UCS txt
  :End

  txt ⎕NAPPEND fh ⋄ ⎕NUNTIE fh ⋄ r←1
∇
```

This writes UTF-8 files rather than full 2-byte Unicode files which has the advantage that most text-editors can handle them, and the file size is generally much smaller. To read either this or any other common format, we have to examine the header bytes and decode appropriately:

```
     ∇ txt←Getcrv fi;fh;ptn;⎕IO;⎕TRAP
      ⍝ Read entire text file as vector
       txt←'' ⋄ ⎕IO←1
       ⎕TRAP←22 'E' '⎕SIGNAL ⎕EN'
       fh←fi ⎕NTIE 0
       txt←unpick ⎕NREAD fh 83(⎕NSIZE fh)
       ⎕NUNTIE fh
     ∇

     ∇ txt←unpick bytes
      ⍝ See what we read from file
        … and decode depending on the leading byte(s)
      ⍝ Make sure it is unsigned,
        … and drop the UTF-8 marker bytes if present
       bytes+←256×bytes<0
       :If 239 187 191≡3↑bytes    ⍝ UTF-8 marker
         bytes↓⍨←3
         txt←'UTF-8'⎕UCS bytes
       :ElseIf 255 254≡2↑bytes    ⍝ Unicode (normal byte order)
         bytes↓⍨←2
         txt←⎕UCS 256⊥⍉⌽((0.5×⍴bytes),2)⍴bytes
       :ElseIf 254 255≡2↑bytes    ⍝ Unicode (big end up)
         bytes↓⍨←2
         txt←⎕UCS 256⊥⍉((0.5×⍴bytes),2)⍴bytes
       :Else   ⍝ Don't try UTF-8 as we may have chars >127 in here
         txt←⎕UCS bytes
       :End
     ∇
```

Obviously, if you want a logfile to persist across a Dyalog-12 upgrade, you should re-create it using something similar, or just check for 160 in the message-type and 'do something' to make sure you don't simply trash the file. I think a good solution would be to check for the 160 case, convert the text to UTF-8 as above, check if the first 3 bytes of the file had the UTF-8 marker and regenerate the entire file if not. Otherwise the only safe option is to substitute a 'standard' marker character ('?' is normal) for anything above `⎕UCS 256` which will at least keep things running while you think about it.

Mappings that stop working

In the 'non-catastrophic' section, we may have been exploiting the DIN mapping from `⎕AV` to the font position for several of the APL characters. When Chris Lee first developed this Windows font layout, he tried to choose 'good' associations, for example the APL comment symbol was mapped to the © character. Dyalog copied the mapping fairly faithfully, and we have all got to know it over the years. For example, I regularly use:

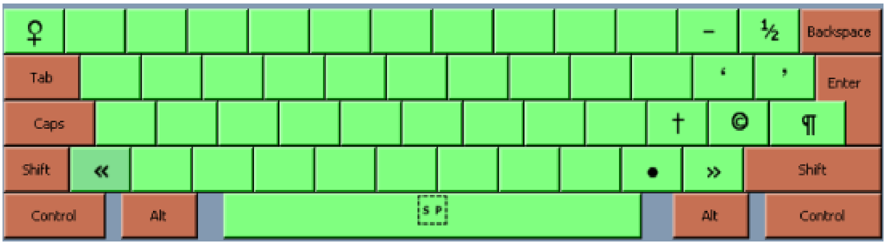
- *delta* and *del* for the ‘typographic’ single quotes. *grade up* and *grade down* are the “speechmarks” also.
- *execute* is – *endash* and *format* gives you a • *bullet*.
- *reverse* and *transpose* (rather surprisingly) give superscripts^{23} which is nice for equations.
- *jot* is good for the *degree* symbol, as in °C in chart captions

I am sure there are others – it took me best part of a day to find and fix all of these in the examples in RainPro. You also need to be aware that *some* of these fixes are a one-way door to Unicode. If you attempt to `)LOAD` a workspace with any text outside the normal range, it will simply say TRANSLATION ERROR and refuse at the first fence. Identifying the cause of the problem can be a bit tricky, so here is another little helper I wrote to get me moving:

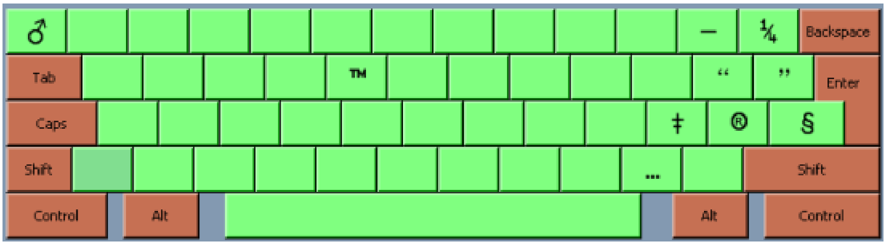
```
     ∇ list←Scan4u ns;nl;fn;cr;ok;dodgy;nslist;vn;var;badchars
       ⍝ Report any dodgy functions - chars not in ⎕AV are dangerous
         … as these won't copy into Classic 12
       ⍝ Rain12 has a test called Unicode which should report here
       ⍝ Takes a namespace as argument - scans current NS if empty
       ⍝ Returns 2-column matrix with namespace.function name
         …  and distinct uncopyable chars
       ⍝ By ACDS Dec 2007
       ⍝

        :If 0∊⍴ns
          ns←''⎕CS''
        :End

        nl←(ns⍎'⎕NL')¯2
        list←0 2⍴''
        nslist←(⊂ns,'.'),¨(ns⍎'⎕NL')¯9.1
        badchars←'‘’“”–—'    ⍝ Load in classic, but map incorrectly!

        :For vn :In nl
          var←⍕∊ns⍎vn
          ok←∧/(var∊⎕AV)∧~var∊badchars
          :If ~ok
            dodgy←∪(var~⎕AV),var∩badchars
            list⍪←(ns,'.',vn)dodgy
          :End
        :End

        nl←(ns⍎'⎕NL')¯3
        :For fn :In nl
          cr←,(ns⍎'⎕CR')fn
          ok←∧/(cr∊⎕AV)∧~cr∊badchars
          :If ~ok
            dodgy←∪(cr~⎕AV),cr∩badchars
            list⍪←(ns,'.',fn)dodgy
          :End
        :End

        :For ns :In nslist ⍝ Go walkies
          list⍪←Scan4u ns
        :End
     ∇
```

One other thing that caught me here – I had a global variable in the workspace to check for 'ascii' characters which included the ^ (ASCII caret) character. When I moved the workspace across, this became an APL ∧ logical *and* which stopped this particular function working in a rather hard-to-detect way.

A side-effect of losing the mappings, was that I suddenly found that it was *really hard* to type these very useful characters. For chart captions and the like, you often want "proper" quotes and dashes, as well as expressions like $3x^2$+2x to label regression lines. The pragmatic solution was to grab a few more keystrokes and set myself up with a little typographic pad on Ctrl+[] and nearby keys:



… with a similar collection on the shifted keys. Most of these were a straight copy from Aldus Pagemaker, and they cover nearly everything you need in day-to-day typing. As you can see from the first character in this paragraph, they also come in quite handy when typing HTML text into Notepad! No *degree* symbol yet though … poor old *ellipsis* might be for the chop, although that ® looks a bit vulnerable too!



Either way, being able to build your own keyboard map and use it everywhere in Windows is really great.

Removing workarounds

Now we hit the downhill part of the ski-run – getting rid of all the nasty stuff we had to do to get around the 256-char limit. A typical example I had was writing SVG (a very picky XML format) files to represent my charts. Any characters above ASCII-128 must be encoded as `&#nnnn;`, and several common characters in the 145 range (like the quotes) required the pukka Unicode values. I won't bore you with the messy stuff I had before – all I need now is to use the native character translator which automatically gives me all the right numbers:

```
 ⍝ Hex any remaining hi-bit characters (reliable translation now)
  vec←,vec ⋄ asc←128>⎕UCS vec ⋄ hi←(~asc)/⍳⍴vec
  :If 0<⍴hi
    aix←⎕UCS vec[hi]
    hex←,(((⍴aix),2)⍴'&#'),('ZI4'⎕FMT aix),';'
    vec←(1+6×~asc)/vec
    hi←hi++\¯1↓0,(⍴hi)⍴6
    vec[,hi∘.+0 1 2 3 4 5 6]←hex
  :End
```

…and of course I had to beef up my C# translator to handle this:

```
       cc.chk 'x←⎕ucs 65 67 3348'
        x     ←     ⎕ucs  65 67 3348
  MARK  NOUN  ASGN  VERB  NOUN
  x = AE_ucs(new int[] {65,67,3348});   // Using ...

 static string AE_ucs(int[] iv)  // Int array translate
 {
   char[] charvec = new char[iv.Length];
   for(int i=0;i<iv.Length;i++)
     charvec[i]=(char)iv[i];
   return new String(charvec);
 }
```

This spins off a suitable helper function the first time it sees a particular call (I just supported the 4 obvious variants with scalars and vectors either way round) and then calls it in-line.

Wrap-up

There is clearly some pain involved, but I think the gain beats it hands down. I do like throwing away code! Fewer lines always means fewer bug-opportunities, and translate tables are a great place for bugs to hang out.

Parallel Each

David Liebtag, IBM Corporation
liebtag@us.ibm.com

The APL Working Group of GUIDE SHARE EUROPE in Germany and APL-Germany e.V. held meetings in Hanover, Germany on 12 and 13 June 2008. David Liebtag made several presentations at those meetings about IBM Workstation APL2 Service Level 12. This and an article in the next issue summarise his presentations about APL2's new support for parallel processing and structured storage. *Ed.*

There has been a lot of discussion recently about how to make it easy for programmers to exploit multiple processors. This topic is of great interest to computer scientists, language developers, application developers and end-users. The problem is that most languages are designed to provide instructions to computers on how to perform a sequence of calculations in series. They do not provide a convenient way to specify how an algorithm should be partitioned and distributed onto separate processors.

For a few years people have been suggesting that because APL already works with whole arrays rather than individual data items, perhaps APL has an opportunity to lead the way in this field. Perhaps APL's semantics already reveal the parallel nature of many algorithms. If that is so, perhaps APL language processors could detect the parallel nature of application components and automatically distribute them to multiple processors.

This article includes an overview of the hurdles to be overcome in the exploitation of multiple processors. It also demonstrates two new operators in Workstation APL2 which allow developers easily to distribute sections of applications onto multiple processors, to network connected machines and achieve significant performance improvements.

What is performance?

When discussing performance, it is a good idea to understand what we mean. However, performance can be hard to define.

For example, does performance simply mean how many calculations are performed? If so, then what is a calculation? Do calculations always use input data and produce results? What about an application's internal calculations, such as incrementing array indexes and determining where and how to access data?

Simply copying data to and from memory can significantly impact performance. In addition, some algorithms require more storage than others and can be constrained by the amount of available storage. Providing more storage using virtual memory and reorganizing algorithms to work within constraints can also significantly impact performance.

Furthermore, performance is affected by *how* data is accessed. Data is read and written from disk drives, networks, and real-time measurement devices. The speed of all these devices can impact performance.

You may be thinking, *Wait a minute. It's obvious. Performance means how fast an application runs.* But what does it even mean for an application to run? Consider that sometimes you may require results with a high degree of accuracy and precision. Other times you may require results that provide only a rough estimate of the answer. These two ways to run an application may require very different amounts of time to run, or amounts of performance. So even beyond the general specification of the results, the specification of the required precision and accuracy of the results can affect performance.

Finally, all these contributions may be insignificant to overall application performance, when using sample data. However, when using scaled-up production data, they may have a profound effect on performance.

How does hardware affect performance?

For years, computer manufacturers' specifications have included clock speed and memory size. More sophisticated users may also understand specifications for chip architectures and instruction sets, and machine cache sizes and bus widths. Clearly, these hardware features affect performance and we have all used some or all of them to inform our decisions when buying computers. If we wanted our applications to run faster, we bought faster machines with more memory.

More recently, machines have become available with multiple cores. You can even buy multiple machines and connect them in a grid or a cloud. How will this affect performance?

It depends on the application. Although some specialised applications are beginning to appear that exploit multiple processors, most applications process a single set of sequential instructions. In general, no matter how many processors you have available, these applications will use only one processor and will not run any faster. Modifying these applications to use multiple processors or multiple machines, can be a very, very difficult job.

What affects distributed computing performance?

In order to distribute an application onto multiple processors, you, or a language processor, must identify which parts of an application can be distributed, slice the data into sections, copy them to the processors, run the code on each of the processors, and gather the results back together in the main application's storage.

Each of these decisions must be done efficiently. The time it takes to determine whether an operation can be distributed, and to copy the data and results between main and multiple processors, can easily overwhelm the performance gains achieved by using multiple processors. In addition, simply managing multiple processors takes time. It is important to minimize the number of times that processors are started and data is sliced, distributed, and gathered.

What affects interpreter performance?

Several features of APL interpreters and applications affect performance.

First, consider scalar operations such as this:

```
      →BIN/LABEL
```

This type of code is very common. It is widely used to control flow of operations within applications. In fact, it is so common that most applications spend the vast majority of their time executing scalar operations. Indeed, since most applications spend most of their time running scalar operations, even significant improvements in array operations can have little affect in overall application performance.

Next, consider array operations such as this:

```
      Z←A×B
```

You may think that these expressions are terrific candidates for automatic distribution to multiple processors. The APL interpreter could slice up the arrays and distribute the primitive to multiple processors. However, consider this example, that uses multiple array primitives:

```
      Z←A×B×C
```

If the interpreter distributed each primitive array operation to multiple processors, it would have to slice up the array, distribute it, and gather the results together for each primitive. This could require a lot of overhead. These remarks seem to imply that the pundits were wrong: APL code does not appear to lend itself to distribution on multiple processors. How can we prove them right? Clearly, we need a way to avoid distributing operations over and over again. We need a way to identify when to distribute operations at times when it will give us the most bang for the buck.

Parallel Each

The APL2 primitive operator *each* ¨ applies a function to multiple arrays. The application developer uses *each* to indicate when operations can be performed independently. IBM has extended this facility with two external operators:

`PEACHP`	parallel *each* using processors
`PEACHT`	parallel *each* using threads

Like the *each* operator, the parallel-each operators apply the function to each element of the argument arrays. Unlike *each*, the elements are processed asynchronously.

`PEACHP` processes each element in a separate process, which can be on the same or other machines. `PEACHT` processes each element in a separate thread running on the same machine.

Both operators use multiple processes or threads to process the data. The number of processors or threads used is controlled by the number of elements in the arrays and the number of machines and processors that are available.

Syntax

```
  result ← [larg] (function PEACHP (options [processors])) rarg
  result ← [larg] (function PEACHT options) rarg
```

where

`options`	Vector of character vectors containing APL2 invocation options
`processors`	Integer vector containing identifiers of processors running AP 200, the *Calls to APL2* processor.
`function`	Character vector containing a function name
`larg` and `rarg`	*each* arguments

The parallel-each operators start multiple slave APL2 interpreters, each with a separate workspace. The invocation options are used when starting these interpreters. The list of processors is used to locate the machines to use.

The function named in the operand is copied from the caller's namescope into the slave interpreters. For this reason, the named function is usually an association with a function in a namespace.

The elements of the arrays are copied into the separate slave interpreters' workspaces. Because the function is applied on the array elements in separate workspaces, side-effects are not supported. The function can not access the caller's workspace.

Sample

The following example compares using the ¨ primitive operator and `PEACHT` to call the `FFT` function from the `MATHFNS` namespace to calculate Fast Fourier Transforms on a machine with two processors.

```
      DATA←⊂[2]?20 65536⍴1000      ⍝ Generate some data

      START←⎕AI                    ⍝ Record the time
      'MATHFNS' 11 ⎕NA 'FFT'       ⍝ Associate FFT
1
      ⍴FFT¨DATA                    ⍝ Apply FFT on each element
20
      ⎕AI-START                    ⍝ How much time did it take?
0 6209 6443 218

      START←⎕AI                    ⍝ Record the time
      4 11 ⎕NA 'PEACHT'            ⍝ Associate PEACHT
1
      ⍴('FFT' PEACHT '')DATA       ⍝ Apply FFT with PEACHT
20
      ⎕AI-START                    ⍝ How much time did it take?
0 47 3541 203
```

The second and third elements of `⎕AI` are the compute and connect times. Notice that by simply changing the code to use `PEACHT` rather than the ¨ primitive operator, the connect time was reduced by 45 percent. The compute time was also reduced to nearly zero, but this is only because CPU time used by the asynchronous interpreters is not accumulated in the calling interpreter's account information.

Usage guidelines

The parallel-each operators currently copy the argument data from the application's workspace to the slave interpreters' workspaces. The results are created in the slave interpreters' workspaces and then copied to the application's workspace. The time required to copy the arguments and results can overwhelm the benefits of using multiple processors.

The parallel-each operators are appropriate for a subset of possible uses of the primitive *each* operator. They can give significant improvements for applications which do not require extremely large arguments, perform computationally intensive calculations, and do not return extremely large results.

Summary

People have been suggesting for a number of years that APL may be able to automatically exploit new machines with multiple processors. With Workstation APL2 service level 12, the IBM APL Products and Services group has proved they were correct. With trivial changes to their programs, APL2 developers can now run their applications on multiple processors and gain performance improvements that are directly proportional to the number of cores available to the applications.

Further information about APL2 and Service Level 12 is available at http://www.ibm.com/software/awdtools/apl. Detailed information about APL2, the parallel each operators, and Service Level 12's other new facilities can be found in the *APL2 User's Guide* through the Library link.

Classes as a tool of thought

Or: acquiring a new father after you're born

Simon Marsden, MicroAPL Ltd
microapl@microapl.co.uk

This article was originally presented as a talk at the BAA AGM in June 2008. *Ed.*

APL has always been a great language for trying things out, changing and experimenting until you're happy with an application. That's why it is often referred to as a 'tool of thought'.

My purpose in this talk is to demonstrate how the design of APLX encourages the user to work this way even when writing object-oriented code. In other words, to show how you can design and modify the class hierarchy as you go along, even when you have instances of the classes you are changing.

To do this, let's consider a concrete example, one small enough to explore here but large enough to demonstrate some interesting features of interactive OOP development. Imagine that we work for a company called BigCorp and have been given the job of organising a 5-a-side football tournament for its employees.

Creating our first class

To start off with, we need to get a list of employees from the company's personnel database. We can do this very easily in APLX using `⎕SQL`.

```
      1 ⎕SQL 'connect' 'aplxodbc' 'DSN=BigCorpDB;PWD=xx;UID=root;'
 0 0 0 0
      1 ⎕SQL 'do' 'use bigcorp'
 0 0 0 0
      fields←'FIRSTNAME,LASTNAME,SEX,EMAIL,EMPLOYEENUMBER,DEPARTMENT'
      (rc errmsg employeeData)←1 ⎕SQL 'do' 'select', …
                                   fields,' from employeedata'
```

We decide on an object-oriented solution to our 5-a-side problem, and that our first class should represent an `Employee`. This will have a number of properties like `firstName`, `lastName`, etc., and a constructor to initialise them when an object is created.

To save space, let's omit the step of using the APLX class editor to create the class. Here is the listing of the finished class. The first part of the `⎕CR` listing shows the properties, followed by the constructor. (In APLX, the constructor always has the same name as the class.)

```
      ⎕CR 'Employee'
Employee {
firstName
lastName
sex
email
employeeNumber
department
&nbsp;
∇Employee arg
(firstName lastName sex email employeeNumber department)←arg
∇
}
```

We now have a new workspace entry called `Employee`, which is a class reference which we can use to create new instances of the class. Here we create a vector with an object representing each employee:

```
      )CLASSES
Employee
      Employee
{Employee}

      employees←⎕NEW ¨⊂[2]Employee,employeeData
      ⍴employees
44
      3↑employees
[Employee] [Employee] [Employee]
```

Notice that the default display of the first three list elements is not very helpful – it just tells us that we have three objects of type `Employee`. We can examine an individual object (or many) using `⎕STATE`:

```
       employees[1].⎕STATE 0
firstName                        Bert
lastName                        Brown
sex                                 M
email           Bert.Brown@bigcorp.com
employeeNumber                     24
department                          C
```

However, it would be nice to have a quick way of telling `Employee` objects apart, and we can do this by changing the default Display Format of an object using the system method `⎕DF`:

```
      employees.⎕DF '[',¨employees.lastName,¨']'
      3↑employees
[Brown] [Ptolemy] [Oakum]
```

Let’s play football

The next thing to do is to tell everyone about the 5-a-side competition and invite them to play. We can do this fairly easily by sending everyone in the company an e-mail:

```
      SM←'⎕' ⎕NEW 'SendMail'
      SM.host←'smtp.bigcorp.com'
      SM.user←'bigcorp'
      SM.password←'sesame'
      SM.Open
0
      SM.from←'simon@bigcorp.com'
      SM.to←¯1↓∊employees.email,¨','
      SM.subject←'Five-a-side Tournament'
      SM.body←'Would you like to play five-a-side football?'
      SM.SendMessage
0
      SM.Close
0
```

Let’s imagine that we have had replies to our e-mail and that 25 people are interested in playing. We’ll choose 25 random players for the sake of the example:

```
     wantingToPlay←employees[25↑(⍴employees)?⍴employees]
```

Now we need to allocate them into teams of five players. We’ll use a very simple new class `Team` which has no constructor and no methods yet, and just has properties called `name` and `players`.

```
      ⎕CR 'Team'
Team {
name
players
}
```

We can then create an object to represent each team:

```
      numTeams←⌊(⍴wantingToPlay)÷5
      teams←⎕NEW¨numTeams⍴Team
      teams.players←,⊂[2](numTeams,5)⍴wantingToPlay
      teams[1].players
 [Brown] [Ptolemy] [Oakum] [Fry] [Wittering]
```

Saving objects

Since the 5-a-side tournament is going to last for several weeks, we need to save the details of the teams somehow.

Imagine for a moment that you were writing this application in another object-oriented language like C# or Java, or even another interpreted object-oriented language like Ruby. In this case, you would need to write the team details to some kind of external file, either a plain text file or something more elaborate. You would also need to write code to read back the details from the file and recreate the team objects.

In APLX, things are much simpler. APL objects continue to exist if you `)SAVE` the workspace and reload it at a later date. Although this seems natural to an APL programmer, it's only possible because APL uses the concept of a workspace. It's interesting both because you have to write less code, and because it allows you to experiment freely with how classes should be structured…

Re-factoring classes

Let's go back a step and imagine that we only had 24 people who want to play football. Unfortunately, we don't now have enough players to make five complete teams, so some people will be left out:

```
      wantingToPlay←24↑wantingToPlay
      numTeams←⌊(⍴wantingToPlay)÷5
      teams←⎕NEW ¨numTeams⍴Team
      teams.players←,⊂[2](numTeams,5)⍴wantingToPlay
      wantingToPlay~∊teams.players
 [Smith] [Jones] [Khan] [Pasty]
```

However, they 'have this mate who plays a little football.' He's not a company employee, but can he play anyway?

The new player doesn't have an employee number or a department, so it looks like we made a mistake with our initial design of the `Employee` class. We need to move most of its properties into a new class `Person` and then make `Employee` inherit from it.

This is where the interactive nature of APLX starts to get interesting, because we can accomplish this without needing to re-do everything from scratch. We can modify the class structure but carry on using all our existing objects! This is very unusual in the world of object-oriented programming languages.

To create the new `Person` class, we could use the APLX class editor to create the class and then add the constructor and all the properties we need, one at a time. For a small class this would not take too long but there is a quicker way to do it, because the system functions `⎕CR` and `⎕FX` have been extended to work with classes:

Get a text representation of `Employee`:

```
      text←⎕CR 'Employee'
```

Edit this text to rename the class as `Person` and delete the two unwanted properties `employeeNumber` and `department`:

```
      text
Person {
firstName
lastName
sex
email
&nbsp;
∇Person arg
(firstName lastName sex email)←arg
∇
}
```

Fix the new class

```
      ⎕FX text
Person
```

Having created the new `Person` class, we want to make `Employee` into a child class which inherits from it by using the system function `⎕REPARENT`:

```
      )CLASSES
 Employee        Person

      Employee.⎕PARENT
 [NULL OBJECT]
      Employee ⎕REPARENT Person
      Employee.⎕PARENT
 {Person}
```

Finally, we change the class definition of `Employee` to delete the properties which we've moved into `Person`, and change the constructor as follows:

```
      ∇Employee arg
      ⍝
      ⍝ Call the constructor of our parent class
      Person 4↑arg
      ⍝
      ⍝ Initialise extra fields
      (employeeNumber department)←4↓arg
      ∇
```

(Notice how a constructor can be called just like an ordinary method. Here we call the base class constructor, `Person`.)

All the objects we've already created continue to exist:

```
      3↑wantingToPlay
 [Brown] [Ptolemy] [Oakum]
```

Now we've changed our class hierarchy we are ready to bring in the extra player from outside the company to make up the fifth team:

```
      newplayer←⎕NEW Person 'Ronaldo' 'Moreira' 'M' ''
      newplayer.⎕DF '[Ronaldinho]'
      newteam←⎕NEW Team
      newteam.players←newplayer,wantingToPlay~∊teams.players
      teams←teams,newteam
      ⍴teams
 5
```

Notice that the new team is of mixed composition:

```
      newteam.players.⎕CLASSREF
{Person} {Employee} {Employee} {Employee} {Employee}
      +/Employee=newteam.players.⎕CLASSREF
4
```

Any questions?

You may have some questions at this point. First of all, what happens to an existing object if you delete its class? The answer is that the object still exists but you can no longer do much with it:

```
      )SAVE football
2008-06-02 14.10.32
      teams[1]
[Team]
      teams[1].players
[Brown] [Ptolemy] [Oakum] [Fry] [Wittering]
      )ERASE Team
      teams[1]
[DELETED CLASS]
      teams[1].players
VALUE ERROR
      teams[1].players
      ^
```

How about if, instead of deleting the class `Team` we just delete the `players` property? Because the property no longer exists, APLX immediately deletes it from any object instances and reclaims the memory, whilst leaving the objects otherwise untouched.

```
      )LOAD football
SAVED 2008-06-02 14.10.32
      )ERASE Team.players
      teams[1].players
VALUE ERROR
      teams[1].players
      ^
```

However, before doing so, APLX will check to see whether the parent class contains a property of the same name. If so, the object's property value is still accessible and is not deleted. (This is why it was important to re-parent our `Employee` class before deleting its unwanted properties, so that we didn't change any existing `Employee` objects.)

How about trying to create circularities in the class inheritance hierarchy?

```
      Employee ⎕REPARENT Person
      Person ⎕REPARENT Employee
DOMAIN ERROR
      Person ⎕REPARENT Employee
      ^
```

Creating a football league

It’s now time to assign names to the teams. Again we’ll use the `⎕DF` trick to make it easier to identify different `Team` objects:

```
      teams.name←'Sheffield Tuesday' 'Water Cooler Wanderers'
                 'The P45s' '5-0-0 Formation' 'Brazil Nuts'
      teams.⎕DF '[',¨teams.name,¨']'
      1↑teams
[Sheffield Tuesday]
```

The last class we need to create is a `Match` class. This will have three properties – the two teams who will play, the `score` (initially 0,0), and a Boolean `played` indicating whether the match has taken place yet. The class also has a constructor, and a method `Winner` which returns the winning team. Here is the listing of the completed class:

```
      ⎕cr 'Match'
 Match {
 played
 score
 teams
 &nbsp;
 ∇Match arg
 ⍝
 ⍝ Store object references for the two teams who will play the match
 teams←arg
 ⍝
 ⍝ Match not played yet
 played←0
 score←0 0
 ∇
 &nbsp;
 ∇R←Winner
 ⍝
 ⍝ Returns winning team as 1-element vector,
 ⍝ or empty vector if match is a draw or not yet played
 ⍝
 R←played/(×-/score)↑teams
 ∇
 }
```

Having created the `Match` class, we can create a vector of all the matches, assuming that each team plays every other team once:

```
       numTeams←⍴teams
       x←x≠∨\x←(2⍴numTeams)⍴(numTeams+1)↑1
       x
 0 1 1 1 1
 0 0 1 1 1
 0 0 0 1 1
 0 0 0 0 1
 0 0 0 0 0
       matches←⎕NEW¨Match,¨(,x)/,teams∘.,teams
       ⍴matches
 10
       matches[1].teams
 [Sheffield Tuesday] [Water Cooler Wanderers]
```

Changing the class of an object

What would happen if one of the employees leaves the company but wants to continue playing? We can use the `⎕RECLASS` system function to change the class of an object instance:

Pick someone to leave and check a few things:

```
       leaver←wantingToPlay[?⍴wantingToPlay]
       leaver
 [Jones]
       leaver.⎕CLASSREF
 {Employee}
       leaver.employeeNumber
 5
```

Change the class, and note how some properties are no longer accessible:

```
       Person ⎕RECLASS leaver
       leaver
 [Jones]
       leaver.⎕CLASSREF
 {Person}
       leaver.employeeNumber
 VALUE ERROR
       leaver.employeeNumber
       ^
```

The normal use of `⎕RECLASS` is to change the class of an object to another related class, as in the example above. However, there is nothing stopping you changing to a completely unrelated class if you wish.

Send in the clones

Now suppose that someone is injured and has to drop out of the tournament completely, so that their team needs to bring in a new player.

It's easy enough to modify the appropriate `Team` object, but if we do so we'll no longer have any record that the injured player took part in the earlier matches. It would be quite nice to keep a record of who actually played in each match.

To do this, we can use the system method `⎕CLONE` to make a copy of each team object at the time that a match is played. After adding a new `Match` property called `whoPlayed`, we can do:

```
       matches[1].played←1
       matches[1].score←3 2
       matches[1].whoPlayed←matches[1].teams.⎕CLONE 1
```

The `⎕CLONE` method has produced one duplicate copy of each of the two teams. The clone objects are independent from their original parents; changing the teams will not affect the copies.

Producing a league table

Next, we want to be able to produce a League Table which shows the teams' positions as the 5-a-side tournament progresses. To do this, we choose to modify the definition of our `Team` class to add the following methods:

`MatchesPlayed`	Return a vector with one element for each match the team has played.
`MatchesWon, MatchesDrawn, MatchesLost`	Return a vector with one element for each match the team has won, drawn or lost.
`GoalsFor, GoalsAgainst`	Return the number of goals scored and conceded.

Here, for example, is the first new method. Note how the niladic system function `⎕THIS` is used within the class method to find out whether our team matches any of the teams playing:

```
∇R←MatchesPlayed
⍝ Returns a list of all the matches the team has played in
⍝
⍝ Get a list of all the matches that have been played so far
→(0=⍴R←(matches.played)/matches)/0
⍝
⍝ Did we play as either team in each match?
R←(∨/¨⎕THIS=R.teams)/R
∇
```

And here is the global function which will produce the league table:

```
      ∇R←League;points;goalDifference
 [1]  ⍝ Return league table of current positions
 [2]  ⍝
 [3]  ⍝ First calculate points and Goal Difference
 [4]  ⍝ (3 points for a win, 1 for a draw)
 [5]   points←(3×∊⍴¨teams.MatchesWon)+(1×∊⍴¨teams.MatchesDrawn)
 [6]   goalDifference←teams.GoalsFor-teams.GoalsAgainst
 [7]  ⍝
 [8]  ⍝ Now create the league table
 [9]   R←teams.name
 [10]  R←R,(∊⍴¨teams.MatchesPlayed,teams.MatchesWon,
       teams.MatchesDrawn,teams.MatchesLost)
 [11]  R←R,teams.GoalsFor,teams.GoalsAgainst,goalDifference,points
 [12]  R←⍉(9,⍴teams)⍴R
 [13] ⍝
 [14] ⍝ Sort it. For teams with the same number of points,
 [15] ⍝ sort on least matches played, then on Goal Difference, then
      Goals Scored
 [16]  R←R[⍒⍉⊃[2](points)(-∊⍴¨teams.MatchesPlayed)(goalDifference)
       (teams.GoalsFor);]
 [17] ⍝
 [18] ⍝ Add the column headings
 [19]  R←((' ')('Pld')('  W')('  D')('  L')(' GF')(' GA')('GD')
       ('Pts')),[1]R
      ∇
```

Let's imagine that the results are in, and the final positions are these:

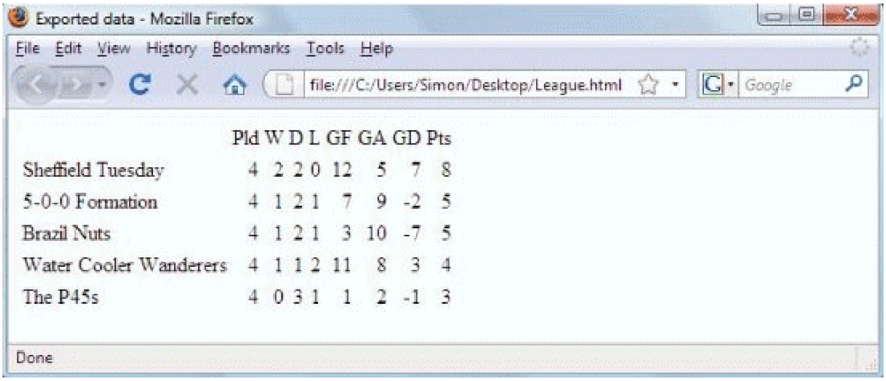
```
      matches.played←1
      matches.score←(3 2)(0 0)(7 1)(2 2)(0 0)(1 5)(8 0)(1 1)(0 1)(0 0)

      League
                          Pld   W   D   L  GF  GA  GD  Pts
 Sheffield Tuesday          4   2   2   0  12   5   7    8
 5-0-0 Formation            4   1   2   1   7   9  ¯2    5
 Brazil Nuts                4   1   2   1   3  10  ¯7    5
 Water Cooler Wanderers     4   1   1   2  11   8   3    4
 The P45s                   4   0   3   1   1   2  ¯1    3
```

One more thing…

Finally, we need to publish the results so that the teams can see how they did. We can easily export the League Table to HTML format for inclusion in a web page:

```
League ⎕EXPORT 'C:\Users\Simon\Documents\LeagueTable.html' 'html'
```

Here is the HTML file opened in a web browser. Although you cannot tell from the screenshot, APLX has produced a properly formatted HTML table, not just a simple text representation.


Exported data - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Users/Simon/Desktop/League.html
Google
Done

	Pld	W	D	L	GF	GA	GD	Pts
Sheffield Tuesday	4	2	2	0	12	5	7	8
5-0-0 Formation	4	1	2	1	7	9	-2	5
Brazil Nuts	4	1	2	1	3	10	-7	5
Water Cooler Wanderers	4	1	1	2	11	8	3	4
The P45s	4	0	3	1	1	2	-1	3

In conclusion

A recent article about objects in the C# programming language included the sentence: "Unfortunately, though, objects are like snowmen; they live happily for a brief period of time before disappearing into the spring sunshine." In APLX, this is by no means the case. APL objects are persistent; they can be saved in a workspace and even survive and adapt as you modify and extend your classes.

An autobiographical essay

Kenneth E. Iverson

Ken and Donald McIntyre worked on what was to be *The Story of APL & J* in 2004 through a series of e-mail messages, with the last e-mail from Ken coming in the morning of Saturday, 2004-10-16. Now Donald McIntyre's health prevents him continuing work on *The Story of APL & J*. The following autobiographical sketch has been extracted from the manuscript, which *Vector* hopes eventually to publish in its entirety. Donald McIntyre emphasises that the text of the sketch is in Ken's own words. *Vector* is grateful for Roger Hui's help with preparing it and for adding the endnotes. *Ed.*

Preamble

My friend Dr Donald McIntyre has a penchant for well-documented historical treatments of topics that interest him. His more important works concern his chosen discipline of geology; notably his discovery and discussion of the lost drawings of James Hutton, and his commemorative works on the occasion of Hutton's bicentennial.

Because of his fruitful use of my APL programming language, and its derivative language J, Donald has asked me many questions concerning their development. I finally suggested (or perhaps agreed to) the writing of a few thoughts on these and related matters.

Because of my other interests in developing and applying J, I have deferred work on these essays, but now realise that the further application of J has already fallen into younger and better hands, such as those of Professor Clifford Reiter in his *Fractals, Visualization, and J* [1].

Likewise, the further development and implementation of J are now in better hands, such as those of Roger Hui, my son Eric and nephew Kirk, and, last but not least, Chris Burke.

Because of my negligence in the keeping of records, I am relieved to realise that I can now in good conscience ignore the provision of carefully-documented references, leaving such matters to McIntyre's expertise. I will also exercise the freedom to explore ideas as they arise, and not restrict myself to fulfilling Donald's expressed desires.

I will begin with a series of topics chosen more or less at random, and will defer the matter of an overall organisation.

Schooling

I started school on 1 April 1926, eight months before my sixth birthday, was promoted to Grade 2 at the end of the school year in June, and was promoted to Grade 4 at the end of the next year.

As I understand it, my father chose the inauspicious day and date as a joke on the elderly and superstitious schoolmaster (Mr McLeod), who reciprocated with his own joke by promoting me to Grade two after just three months. I was, of course, far from ready, but since he was about to retire, I was left as a problem for the new and much younger Mr Norman Bowles.

To my surprise, Mr Bowles put me in neither Grade 1 nor 2, but kept me separate. I recall my eager anticipation of being placed in Grade 2 when (after about a month) I reached their reading lesson. I was disappointed to find that nothing changed, but was rewarded by promotion to Grade 4 at the end of the year, and was ready to leave the school to enter Grade 9 at age 12.

All this was possible in a one-room school (with one teacher and some 30 students). Younger students could overhear the work of their elders, who might also be asked to help them. I also had much help from my older sister Aleda, who herself went on to become a teacher.

The timing was fortunate, because progress soon overtook the rural province, the one-room schools disappeared, and local children were bussed some fifteen miles in snowy Alberta winters.

For Grade 9 I had to go to the nearest village about 14 miles away, and stayed with family friends on a farm only 5 miles from the school. As a 12-year-old I was a social misfit in high school, but completed the year with a good start on Grade-10 subjects.

Although I enjoyed school, I chose to quit at the end of the year to work at home on the farm. In those years of the Great Depression my choice was perhaps a relief to my family. But my reasons were simple: as far as I knew, the only purpose of further schooling was to become a schoolteacher, and that I decidedly did not want. I knew absolutely nothing about universities and the preparation they provided for careers quite unknown to me.

I finally learned about universities from my Air Force mates, many of whom planned to return to university, thanks to government support for servicemen. In fact, one of my buddies threatened to return and beat my brains out if I did not grasp the opportunity.

When I quit school, my younger brother Byron was sent to school in the city of Edmonton, staying with an aunt and uncle. I asked him to borrow science books for me from the city library, and he chose some on radio – sparking my interest.

In 1938 at the age of 17 I enrolled in a correspondence course with De Forest's Training of Chicago, at a cost of $205. Fortunately, my elder brother Elmer chose to do likewise – it was very helpful to have someone to work with.

We desperately wanted radio parts to work with, and persuaded Dad to buy a radio that would run on a re-chargeable automobile battery rather than on expensive 135-volt dry cell batteries – and, more importantly, to let us have the old radio.

We then began to dissect it, carefully making a circuit diagram as we went. I find it interesting to recall our difficulties in identifying parts (such as "condensers" or "capacitors"), having studied them in the abstract, but never having seen one on the hoof.

The correspondence course culminated in two weeks practical training in Chicago, a thrilling time for us, made more so by our uncle Ingmar with whom we stayed – he loved Chicago, and made sure that we saw all we could of it (including giant presses at The Chicago Tribune) in the time available.

Looking for further reading after completing that course, I soon realised that the really interesting books on radio and electricity used calculus. On a visit to Edmonton, I looked for a book on the subject, and came away with a used copy of *Calculus* by Herman W. March and Henry C. Wolff [2].

The long cold Alberta winters sometimes made outdoor farm work impossible, and granted me the leisure to study M&W. My most vivid memory is of the joy in discovering how the beautiful circular functions were finally united in a single family under the exponential.

I think it worth contrasting this joy with the reaction of my youngest brother (Clayton Ray – no less competent than I) to his encounter with the same material in a university course. I was disturbed that he didn't share my joy in it, but dismissed it as "just more formulas".

I was drafted into the army in 1942 and joined the air force in 1943. The Canadian Legion offered correspondence courses to men in military service, and I took eight of them, nearly enough to complete my high school. Three things I remember – the amazing patience of my tutors, the fact that I never met another serviceman who took courses, and the useful discipline of studying a text with no immediate recourse to a teacher.

Years later I was pleased to find that my own work in APL was exploited by a Mr Clementi who, some 40 years ago, chose to use it in a correspondence course in Australia because of its brevity. This brevity made it feasible to enter a student's written submission, run it (perhaps with modifications to make it work, or otherwise improve it), and mail the typed result from the computer terminal.

After my discharge in 1946, I enrolled at Queen's University in Kingston (Math and Physics in the Arts Faculty). My experience as an undergraduate at Queen's University in Kingston was unusual in that about half the class consisted of returning veterans: the faculty was overjoyed to find students so serious about their work, and the non-veterans were resentful of the “unfair competition”.

I rather resented the requirement to take two courses quite outside of my main interest; one in English and one in Philosophy. I found both most rewarding, and am left with a firm belief in forcing students to look at some areas that they are confident are of no interest to them.

Physics labs required a lot of time, and taught me two things. The first was that they were designed only to confirm things we “already knew” from theory, and were not “experiments” in the sense of discovering anything. This was a lesson I made use of in later work (at IBM) in designing “computer experiments”.

The second was the importance of recording results in pencil, so that they could be fudged to show reasonable results in the required lab reports. A close friend among the younger students learned this the hard way (in a lab designed to measure the difference in the heat capacity of air at constant volume and at constant pressure).

Because of “bad” results he had to repeat the experiment. His new bad results infuriated the professor in charge, who came in on a Saturday to watch the whole process. The upshot was that he re-analysed the experiment, concluding that secondary effects (such as draughts from an open window) would mask the intended effect – in spite of which, students had been reporting “good” results for a full five years.

I also learned something of the difference between detailed meticulous teaching, and good teaching. In physics, Professor Cave was meticulous in copying from his notebook only the main steps in a proof, making brief comments on the algebra or calculus needed to connect them, but leaving students to fill in details for themselves. This we did, to great advantage.

In math, Professor Jeffrey (head of the department) achieved much the same, though inadvertently. Beginning a lecture with an obviously well-planned approach, Professor Jeffrey (head of the Mathematics Department) might soon pace the stage and say “I’ll bet we could do it this way”. The lecture often ended in some confusion, and our subsequent work to unsnarl it taught us a good deal. I remember the class as the first time I actually saw a mathematician at work.

Jeffrey urged me to go on to graduate school – an idea I accepted because the veteran’s benefit would pay, and because I had no idea what else to do. But, when I proposed to stay on with the newly formed graduate program at Queen’s, he objected, saying that in my four years I had absorbed their point of view, and should move on.

Acknowledging his point, I declined the advice to move because I was already 30 years old, and had the responsibility of a wife and two children. Jeffrey restored my perspective by remarking that 30 was his age at beginning graduate school, that he and his wife had no children and still did not, but did have two cats.

In the event, on my graduation in 1950 I went to the math department at Harvard, a move I have never regretted. Although I got respectable grades, I found the math department a very cold place, perhaps in recognition of my limited potential for math. Anyway, I got my Masters degree in one year, and then switched to the Department of Engineering and Applied Physics, after being attracted by a course that I took with Professor Howard Aiken – called “Switching Theory” I think.

Although press reports of the new “giant brains” were beginning to appear, I had no idea of the computer work going on at Harvard under Aiken. Although he worked in a different division, I chose to attend his course, and was quickly captivated. It was the first time as a student that I was given the impression that there might be new work to be done.

Although we sometimes had as many as six students to a desk, Aiken squeezed his graduate students into his Laboratory. To us it made an enormous difference to finally be “on the inside”, with significant access to faculty, and with good opportunities to serve as Teaching Fellows in courses.

Aiken quickly arranged a thesis topic for me by introducing me to economics Professor Leontief, whose graduating student had used Aiken’s computer in his work in Input-Output Analysis. I was expected to extend the work to handle “Capital Goods”; mathematically an extension to systems of differential equations.

I believe that Aiken did a great deal to encourage clear writing by his students, and with relatively little effort on his part. He (deliberately, I expect) developed a reputation for fierce reading of drafts of theses, with the result that we all asked our fellow students to criticise our work before approaching Aiken.

When he read my first draft, I found him both helpful and gentle, often asking if I was sure that this was just the right word, when he knew full well that it was not.

Teaching I

When I received my degree in 1954, Aiken got me appointed to the faculty as an Instructor to help man his new Masters program in Automatic Data Processing. Except for a 6-month “mini-sabbatical” to work for McKinsey, I continued until 1960.

At the time that I graduated in 1954, Aiken was advising that universities should get out of the business of designing and building computers, and should turn their attention to the applications of computers. He argued that companies such as IBM and Remington Rand already recognised a business opportunity, and were in a better position than universities to address such problems as quality control.

Against strong opposition from the Harvard administration, Aiken managed to introduce a Masters program in Automatic Data Processing in 1955; in effect, the first computer science program. I was one of four of his graduating students that Aiken was able to get appointed to the position of Instructor to man his program. A thorough account of these years is included in *Howard Aiken: Portrait of a Computer Pioneer* [3], by the Harvard historian I. Bernard Cohen.

Although Aiken had mapped out a broad program that included economics, business applications, switching theory, operations research, numerical analysis, and computer programming, it was largely left to us green graduate students to flesh out the courses.

I was appalled to find that the mathematical notation on which I had been raised failed to fill the needs of the courses I was assigned, and I began work on extensions to notation that might serve. In particular, I adopted the matrix algebra used in my thesis work, the systematic use of matrices and higher-dimensional arrays (almost) learned in a course in Tensor Analysis rashly taken in my third year at Queen’s, and (eventually) the notion of Operators in the sense introduced by Heaviside in his treatment of Maxwell’s equations.

Harvard had a benign rule of five years to tenure or out, benign because it prevented fierce Bostonophiles from clinging forever in the hope of eventual tenure. It was particularly benign in those postwar days of the establishment of commercial research centres, where anyone from the science faculty of a respectable university could expect to double his salary – as I did when I joined the Research Division of IBM.

Schools of education prepare teachers for elementary schools with a great emphasis on the techniques of teaching. At college level, on the contrary, no preparation for teaching is (or at least was) provided, and graduates, full of their subjects, were unleashed on students. I was fortunate in having as a Teaching Fellow a graduate student who had been born at a podium – Fred Brooks. Moreover, Fred and I were sufficiently close friends that he could, and did, tell me bluntly of my bad teaching techniques in post-mortems held after each lecture. For example, he once said "That was a very interesting point you made about such-and-such – too bad you made it to the blackboard, and could not be heard past the first row."

First test of notation

The first real encouragement of my work in notation came when I took a six-month mini-sabbatical after two years of teaching in Aiken's program, to work for the still-thriving consulting firm McKinsey & Co. This was in the fall of my third year on the faculty.

This occurred because M&Co. had undertaken work that required the use of a computer. Having no computer knowledge themselves, M&Co. turned to the source of all knowledge (Harvard, and eventually Aiken), and Aiken recommended me.

When I arrived in San Francisco, Ted Strong (of M&Co.) explained that his client, Hawaiian Sugar (in a dispute over shipping rates with Matson Lines), had embarked on a thorough analysis of their options – including, for example, the choice of locating refineries in Hawaii or in the US. This required a computer, and Ted had secured the services of Bob Oakford and Dan Fisher of Stanford University, to program the (then rather new) IBM 650.

Ted held weekly meetings with Oakford to communicate his remarkably-detailed knowledge of sugar and shipping. These meetings seemed to go well, but he was unable to get anything understandable about the progress of the programming. I asked Ted to do the same for me, but recorded the steps and decisions in the notation I had been developing for teaching, notation that was eventually implemented as APL.

The encouraging thing was that we were able to converse in accurate detail, Ted knowing nothing of computers, and I knowing nothing of sugar and shipping. Moreover, Oakford and Fisher were able to understand these same programs, just as I was able to understand their 650 programs. In a few months the 650 programs were nearly complete, and we four eagerly anticipated their use – whereupon the project was abruptly terminated.

This was a great disappointment, even though the reason was victory – all this computer stuff had intimidated Matson to agree to a re-negotiation of rates.

IBM Research Center

I left Harvard with offers from IBM Research and Bell Laboratories. Fred Brooks (who had joined IBM somewhat earlier) advised me to choose IBM because computers were their primary concern. I did not join IBM until 1960, at which time I was just finishing up my *A Programming Language* [4], which included a chapter (called *Microprogramming*) on the formal description of the IBM 7090 machine. After I joined the IBM Research Division, Fred advised that I stick to whatever I really wanted to do, because management was so starved for ideas that anything not clearly crazy would find support. In particular, I was allowed to finish and publish *A Programming Language* [4], as well as *Automatic Data Processing* [5] with Fred Brooks as co-author.

I soon met Adin Falkoff, and we worked jointly on the notation for some twenty years. It was Adin who suggested [6] the name *APL* (from the title of my 1962 book). In the 1962 book, I had used APL to describe an IBM computer, and Adin and I (with Ed Sussenguth) used it to produce a formal description [7] of the IBM System/360 computers then under design.

An important consequence of this work (published by John Lawrence in the IBM Systems Journal) was that it attracted important contributors to APL, notably Larry Breed (a Stanford student, who joined us at IBM and chose to undertake the crucial work of implementing it as a programming language), and Donald McIntyre, head of Geology at Pomona College (who had acquired a 360 Model 40 for Pomona, and used the formal description to become more expert than the IBM Systems Engineer assigned to Pomona).

Teaching II

I believed that APL could be used in teaching, and Adin said that to test the point we must take a text used in the State school system, and try to teach the material in it. He further proposed that we invite active high school teachers.

We hired six for the summer, with the plan that two (nuns from a local school, who could provide a classroom in which we supplied a computer [typewriter] terminal) would do the teaching, while the other four (with a two-week head start) would write material.

To our surprise, the two teachers worked at the blackboard in their accustomed manner, except that they used a mixture of APL and conventional notation. Only when they and the class had worked out a program for some matter in the text would they call on some (eager) volunteer to use the terminal. The printed result was then examined; if it did not give the expected result, they returned to the blackboard to refine it.

There were also surprises in the writing. Although the great utility of matrices was recognised (as in a 3-by-2 to represent a triangle), there was a great reluctance to use them because the concept was considered to be too difficult.

Linda Alvord said to introduce the matrix as an outer product – an idea that the rest of us thought outrageous, until Linda pointed out that the kids already knew the idea from familiar addition and multiplication tables.

Finally, it was this interest in teaching that led us to recruit Paul Berry, after seeing his “Pretending to Have (or to Be) a Computer as a Strategy in Teaching” [14] when it appeared in *Harvard Educational Review*.

Rigidity of viewpoint

It was a revelation to see that the outer product (adopted from Tensor Analysis) could appear so simple from a different point of view. However, this was a lesson that had to be learned again and again. A few examples:

- We became so wedded to the power of operators that we made complex and awkward attempts [8] to use them to achieve the effects of `f+g` and `f×g` as used in calculus texts. One such proposal [9] was made by me and Arthur Whitney at an APL conference – probably in Heidelberg.
- On the return flight from the 1988 APL conference in Australia, Gene McDonnell and I worked out what came to be called *fork* and *hook* [10] – realising that the forms `f+g` and `f×g` were not used in APL, and could be introduced without conflict.
- But again it took some time to adopt the more general, and more accessible, notion of a train, and to suppress discussion of forks. It is more accessible because this sense of *train* occurs in English dictionaries: “An orderly succession of related events or thoughts; a sequence”.
- Similar rigidity appeared in our use of the terms *vector*, *matrix*, and *higher-dimensional array* instead of the accurate and familiar *list*, *table*, and *report* [11]. The last provides a completely general term, which may be qualified by *rank-1*, *rank-2*, etc., and even by *rank-0* for a scalar.

- The notion of the dual of a function [12] (or system) is clearly important, because it is invoked in many areas of mathematics. It was not until I realised that the dual of a function `f` was more properly expressed as “the dual of `f` with respect to a second function `g`” that I saw the possibility of using a conjunction `&.` in the present formulation: `f&.g` is `(g inverse)` of `f` of `g`. This has found wide application, and the conjunction is called *under* – from analogies such as “surgery under anaesthetic”. The general idea is that many (if not most) procedures (not only in mathematics) must be performed after some preparation `g` and finally the effects of `g` must be undone.

IBM Scientific Center

As manager of IBM’s Scientific Centers, Joe Mount invited us to open a centre devoted to APL. We chose Philadelphia, and moved the entire group, remaining for about seven years.

These were very fruitful years, beginning with the organisation of the first APL conference in Binghamton, NY in mid-1969 (dubbed “The March on Armonk" by Garth Foster of Syracuse University, because of the perceived neglect of APL by IBM headquarters). This conference attracted a number of new people, including Donald McIntyre – who had been “sent” by IBM because of Pomona’s Model 40 and his expert knowledge of it through studying the APL description of the 360 computers. In spite of a busy schedule, Donald spent a short time with the APL group the same summer.

Teaching III

My daughter Janet attended Swarthmore High School, and recommended Rudy Amann (head of the math department) as an excellent teacher. I therefore approached him with a proposal that we put an APL terminal in his school as a tool for teaching mathematics, suggesting that he first spend the summer with the APL group to assess the matter.

Rudy responded that he could spend only two weeks, which he did. I gave him an office with a terminal (and the calculus text in APL [13] that I had written after our earlier experiment with high school teachers), and invited him to come to me or anyone in the group with questions. Since he never stirred from his office, I despaired, but at the end of the two weeks he announced that he wished to go ahead with the project.

Rudy was pleased with the results, and told me of canvassing those of his students who went on to college, finding that they were pleased with the preparation he had given them. One thing he had done was to use some of the final two “review” weeks to show them the translation from things like `+/` to the sigma notation they would encounter in college.

Rudy continued through a second year, but the inevitable occurred – he was promoted to principal, and had been unable to interest any other teacher in the math department in continuing.

About this time we hired Paul Berry. Paul had degrees in psychology (including a PhD), adding to the rather wide diversity in the early APL crew:

- Chemistry: Adin Falkoff
- Math and Physics: me
- English: Gene McDonnell, Graham Driscoll
- Computer Science: Larry Breed, Roger Moore, Phil Abrams
- Mechanical Engineering: Richard Lathwell
- Secretary: Colleen Conroy (learned APL and promoted to Programmer)
- Other: Ziad Ghandour who, on a trip to his native Lebanon, contributed the Arabic in one of the documents, saying that APL should have originated in Mecca.

I.P. Sharp Associates

I.P. Sharp Associates of Toronto was a pioneer user of APL, establishing it as the base of a time-sharing service widely used in Canada, U.S., and Europe. This came about because Larry Breed had proposed that we hire Roger Moore as a consultant to aid in his implementation of APL. Roger was his classmate at Stanford, and a senior member of I.P. Sharp – he was impressed enough by APL to propose that his firm adopt it.

Having effectively lost control of APL to the Palo Alto Scientific Center (under Horace Flatt), I left IBM and joined Sharp. It was exhilarating to work for a company seriously devoted to APL. Moreover, the seven years to my next retirement (at 67) saw many enhancements to the APL system, largely under the direction of my son Eric.

His group was largely made up of young men who had come to Canada from U.S. to escape service in Vietnam. Their beards and otherwise scruffy appearance led a visiting businessman to say “This place is a zoo.” Rather than resent the implied insult, the group proudly adopted the name, and were known thereafter as “The Zoo”.

My work that proved most important to the later development of J was the publication of my "A Dictionary of APL" [15]. In particular, it proposed a parsing scheme that depended on the "first four elements only" of an execution stack.

Parsing was controlled by a brief four-column table of cases – the first row that agreed with the first four elements of the execution stack determined what action was to be taken.

Roger Hui later told me that during a six-month period off work he had had the leisure to study this paper carefully, and it was this study [16] that led to our collaboration on J after my retirement from Sharp.

Second retirement

In my "A Personal View of APL" [17] I said:

> … the initial motive for developing APL was to provide a tool for writing and teaching. Although APL has been exploited mostly in commercial programming, I continue to believe that its most important use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects.

When I retired from paid employment, I turned my attention back to this matter, and soon concluded that the essential tool required was a dialect of APL that:

- is available as "shareware", and is inexpensive enough to be acquired by students as well as by schools
- Can be printed on standard printers
- Runs on a wide variety of computers
- Provides the simplicity and generality of the latest thinking in APL

The result has been J, first reported in the *APL90 Proceedings* [18].

In *Incunabulum* [19], an appendix to his *An Implementation of J* [20], Roger Hui describes the inception of work on J as follows:

> One summer weekend in 1989, Arthur Whitney visited Ken Iverson at Kiln farm and produced – on one page and in one afternoon – an interpreter fragment on the AT&T 3B1 computer. I studied this interpreter for about a week for its organisation and programming style; and on Sunday, August 27, 1989, at about four o'clock in the afternoon, wrote the first line of code that became the implementation described in this book.

Roger's acknowledgement in this book was the most extravagant I have ever received: *Ex ungue leonem*.

Details of the development of J can be seen in every release by clicking on the Help menu, and then on Release Highlights. For example, from the J 4.01 Release:

- Use `=:` for global definitions in scripts. Run → Window and Run → File use `load` and definitions made with `=.` are local to `load` and disappear when it finishes.
- The J file suffix has changed from `.j?` to `.ij?` (`.js` to `.ijs`) to avoid javascript conflicts.
- Use `system\extras\migrate\ext.ijs` if you have lots of files.
- system directory contains the other distributed directories (`main\stdlib.js` is now `system\main\stdlib.ijs`).

Chris Burke and my son Eric soon became interested in J, and, among many other things, designed and implemented a user interface (GUI) to the Windows operating system. The aspect of their work that I most appreciate is the Labs and Lab Authoring system – which is, unfortunately, not yet widely used.

Teaching IV

Since J is now free, and uses only ASCII characters, the requirements set forth in my "A Personal View of APL" have been fully met. Moreover, a few have used it seriously at college level (notably, Clifford A. Reiter of Lafayette College [23]), but interest in such work spreads slowly.

But my attempts at interesting schools at lower levels have had no result – if it is not in the approved curriculum, it doesn't exist. However, I have gained important teaching experience by using J in a workshop called "Exploring Math", presented to classes of retired folk drawn from a wide variety of non-mathematical backgrounds, including teachers at all levels, lawyers, medical doctors, and psychiatrists.

This workshop suffered from the fact that few had easy access to a computer, a situation that has now changed radically for the better. More importantly, it suffered from the fact that I drastically over-estimated the speed at which they could overcome their fear of mathematics and assimilate ideas foreign to their experience. As a result, the class soon dropped to three, but it reached the point where the work was largely directed by their interests and resulting questions – which we explored at length using J.

For this workshop, I wrote *Math for the Layman* [21], basing it on the idea of mathematics as a language, as expressed in Lancelot Hogben's still-popular *Mathematics for the Millions* [22]:

The view which we shall explore is that mathematics is the language of size, shape and order and that is an essential part of the equipment of an intelligent citizen to understand this language. If the rules of Mathematics are the rules of grammar, there is no stupidity involved when we fail to see that a mathematical truth is obvious. The rules of ordinary grammar are not obvious. They have to be learned. They are not eternal truths. They are conveniences without whose aid truths about the sorts of things in the world cannot be communicated from one person to another.

References

[1] *Fractals, Visualization, and J*, Cliff Reiter, 3rd edn, 2007, ISBN 978-1-4303-1980-1, Lulu.com http://www.lulu.com/content/635966

[2] *Calculus*, H.W. March & H.C. Wolff, McGraw-Hill, 1917

[3] *Howard Aiken: Portrait of a Computer Pioneer*, I. Bernard Cohen, MIT Press, 2000, ISBN 0262531798

[4] *A Programming Language*, Kenneth E. Iverson, Wiley, 1962

[5] *Automatic Data Processing*, Fred Brooks & Kenneth E. Iverson, Wiley, 1963

[6] *A Source Book in APL*, http://www.jsoft-ware.com/jwiki/Doc/A_Source_Book_in_APL#origins_of_APL

[7] "A Formal Description of SYSTEM/360", A.D. Falkoff, K.E. Iverson & E.H. Sussenguth, *IBM Systems Journal*, 3, N°3, 1964 http://www.research.ibm.com/journ-al/sj/032/falkoff.pdf

[8] *Remembering Ken Iverson*, Roger Hui, 2004, http://keiapl.org/rhui/remember.htm#fork0

[9] "Practical uses of a model of APL", Kenneth E. Iverson & Arthur T. Whitney, *ACM SIGAPL Quote-Quad*, 13, N°1, September 1982, ISSN 0163-6006, ht-tp://portal.acm.org/citation.cfm?xml:id=390006.802236

[10] "Phrasal forms", Eugene E. McDonnell & Kenneth E. Iverson, in *Conference Proceedings on APL as a Tool of Thought*, 1989, ISBN 0-89791-327-2, http://portal.acm.org/cita-tion.cfm?xml:id=75172

[11] *Nouns*, in *The J Dictionary*, http://www.jsoftware.com/help/dictionary/dicta.htm

[12] *Ken Iverson Quotations and Anecdotes*, Roger Hui, http://keiapl.org/anec/#under0

[13] *Elementary Functions: An algorithmic treatment*, Kenneth E. Iverson, 1966, Science Research Associates, http://www.jsoftware.com/jwiki/Doc/Elementary_Func-tions_An_Algorithmic_Treatment#Preface

[14] "Pretending to Have (or to Be) a Computer as a Strategy in Teaching", Paul Berry, in *Harvard Educational Review*, 34, 1964, pp.383-401

[15] "A Dictionary of APL", Kenneth E. Iverson, in *ACM SIGAPL Quote-Quad*, 18, N°1, September 1987, ISSN 0163-6006, http://portal.acm.org/citation.cfm?xml:id=36983.36984

[16] Roger Hui, *op. cit.*, keiapl.org http://keiapl.org/rhui/remember.htm#parser

[17] "A personal view of APL", Kenneth E. Iverson, *IBM Systems Journal*, 30, N°4, 1991 http://www.research.ibm.com/journal/sj/304/ibmsj3004O.pdf

[18] "APL\?", Roger K.W. Hui, Kenneth E. Iverson, Eugene E. McDonnell & Arthur T. Whitney, in *Conference Proceedings on APL 90: for the future*, 1990, pp.192-200, ISBN 0-89791-371-X, http://portal.acm.org/citation.cfm?doxml:id=97808.97845

[19] *Incunabulum*, Roger Hui, 1992, http://www.jsoftware.com/jwiki/Essays/Incunabulum

[20] *An Implementation of J*, Roger Hui, 1992, http://www.jsoftware.com/jwiki/Doc/An_Implementation_of_J

[21] *Math for the Layman*, Kenneth E. Iverson, http://www.jsoftware.com/books/pdf/mftl.zip

[22] *Mathematics for the Million*, Lancelot Hogben, W.W. Norton, 1993

[23] Reiter, *op. cit.*

LEARN

SALT II

Dan Baronet, Dyalog Ltd
danb@dyalog.com

Introduction

SALT stands for Simple APL Library Toolkit, a code-management tool for APL. It first appeared in V11 as a prototype. Since then it has undergone many changes and is now fully supported by Dyalog. It allows you to store code in flat Unicode text files, often called scripts, outside the workspace. There are many ways to keep code out of the workspace but text files have advantages over traditional APL ways. they let you:

- Exchange, send or archive the scripts without using the interpreter
- Compare scripts easily
- Divide and work on different sections of code in parallel
- Use an external code-management systems

In environments where the ability to manage code externally is important this is a considerable advantage over storing code in workspaces.

Basics

Besides storing and retrieving, SALT allows you to list and view folders of scripts. It can save multiple versions and manage them locally. It can compare them. And it comes with its own set of utilities.

If you share code it is important to be able to manage it, one way or another. You want to be able to:

- Store multiple versions and retrieve *any* of them
- Compare them
- Do housekeeping

Dyalog offers all this with SALT and more. You don't have to use SALT's ability to store multiple versions but it might well be all you need. If a third-party version-control system is used, SALT's versioning should not be used as it would probably interfere rather than help.

SALT can store functions and sourced namespaces [1] onto file. Non-sourced namespaces can be stored but they need to be converted into sourced form first

[2]. At present, there are restrictions and root variables, for example, cannot be stored.

All the SALT code resides in `⎕SE` and is enabled by default in V12.

Simple examples

1. Function `Foo` is defined and we want to keep a scripted copy of it. To do so we enter

   ```
   ⎕SE.SALT.Save 'Foo  \projectZ\fns\Foo'
   ```

   and file `\projectZ\fns\Foo.dyalog` will now contain a copy of the function in text form.

2. Sourced namespace `Utils` contains functions and variables used everywhere. They constitute a set of utilities that must remain together. We want to store the namespace in file `U1.dyalog`:

   ```
   ⎕SE.SALT.Save 'Utils  \projectZ\utils\U1'
   ```

   Note that in this case we made the filename differ from the namespace's name.

3. Sourced namespace `GUIutils` is another set of utilities relying on the above `Utils` namespace. To save it and start using version numbers we add the `version` switch:

   ```
   ⎕SE.SALT.Save 'GUIutils   \projectZ\utils\U2  -version'
   ```

The next time we want to get those two namespaces (and their contents) all we need to do is

```
⎕SE.SALT.Load  '\projectZ\utils\U*'
```

There may be more sets of utilities starting with the letter `U`. If we don't want them all we must load the ones we want one by one. If there are dependencies we can tell SALT by using the SALT tag `⍝∇:require`. For example, if the above `GUI` namespace requires the `Utils` namespace to be present we should insert the line

```
⍝∇:require  \projectZ\utils\U1
```

preferably at the top of the source. If we know that `Utils` is always in the same folder as `GUI` we can use instead

```
 A∇:require  =\U1
```

with the `=` meaning *same folder as myself*. We then only need to load `U2`, even if both files are moved together to a different location.

Automatic update

Because SALT intercepts editor events, it can save changes on file right after editing an object, optionally prompting you for confirmation.

In other words, every time you edit a salted object, SALT will come up and ask you if you wish to save the changes back to file, making a new version if necessary.

Making life easier, the settings

Instead of always specifying the path of the objects to save or load you can tell SALT to look in specific places when a relative path (one that does not start with `\`) is given. To do so you use

```
 ⎕SE.SALT.Settings 'workdir  location1;loc2;…;locN'
```

SALT will *save* objects in `location1` if a relative path is given or *look* in each location until the file is found when a load is requested. Each full path must be separated from the next one by a semi-colon.

If you do not wish SALT to confirm with you when saving the changes every time you modify an object you should do

```
 ⎕SE.SALT.Settings 'edprompt  0'
```

This will skip the prompting and the changes will be made to file automatically.

Everyday examples

Let’s have a look at a typical use.

Mike has been gathering functions for years. They’re all over the place, in various workspaces, sometimes in duplicates, sometimes in triplicates. He wants to clean this up and start using a system to manage his code. SALT gives him two choices:

1. He can store all his functions, each one in a single script file, grouped by topic in folders of his choice.
2. He can start reorganising each workspace in namespaces and store each namespace in a separate file.

Whichever he chooses, he can snap his workspace straight into files like this:

```
⎕SE.SALT.Snap '\my\APL\folder'
```

Each function and namespace will be saved in a file with the same name followed by the `.dyalog` extension.

If he wants to start keeping track of versions he must add the `-version` switch as well. Let’s see both cases:

Method 1: store each function in a separate file

Let’s assume the workspace looks like this:

```
      )FNS
main    ublock  ucut    uopen   GUI_close       GUI_open
```

To store each function in separate files in the same folder we simply do

```
⎕SE.SALT.Snap  '\projectX\scripts\APL'
```

We can then use a ‘Disk Explorer’ type program to organise files in folders.

If we decide to reorganise the functions and, say, put the GUI functions together, the utilities (those functions whose names begin with `u`) together and the rest in the top folder, we can do

```
⎕SE.SALT.Snap  '\projectX\scripts\APL\GUI    -stem=GUI'
⎕SE.SALT.Snap  '\projectX\scripts\APL\utils  -stem=u'
```

We now save the remaining functions in `\projectX\scripts\APL`.

```
⎕SE.SALT.Snap  '\projectX\scripts\APL'
```

SALT keeps track of what has been saved and won’t save a new copy again.

Method 2: regroup some functions into namespaces first

Same thing. This time we organise the functions into namespaces first. We must create the namespaces and move functions into them. We can

a. create the namespaces:

```
        'GUI'  'utils' ⎕ns¨⊂''
```

   and then use Workspace Explorer to reorganise (move) the functions or

b. do it manually, e.g.:

```
        'utils' ⎕NS  list ← 'u' ⎕NL 3 4  ⋄  ⎕EX list
```

We need to make sure the functions do not appear in two places (hence the `⎕EX`).

Then we do all functions and namespaces at once:

```
⎕SE.SALT.Snap '\projectX\scripts\APL'
```

Note that everything (functions and namespaces) are saved in the same location. If you wish to separate them you could do

```
      ⍝ functions in folder 'base'
⎕SE.SALT.Snap '\projectX\scripts\APL\base   -class=3 4'
      ⍝ namespaces in folder 'groups'
⎕SE.SALT.Snap '\projectX\scripts\APL\groups -class=9'
```

When an object has been salted, modifications can be stored to file automatically after making changes via the editor. If you subsequently erase an object or redefine it the tagged information is lost and no automatic update can occur.

Important note

Only *sourced* functions and namespaces can be tagged. Non-sourced functions (e.g. derived functions) and namespaces cannot be tagged and although some of them can be saved (SALT generates source for them subject to constraints such as no GUI object in them), they cannot be edited and must be resaved manually or skipped. If you wish to convert a non-sourced namespace into a sourced one you should use the SALT utilities [3] provided. Saving it will then allow SALT to pick up subsequent changes automatically.

Saving new code

Once your code is in SALT your changes will be picked up automatically if you wish. If you disable SALT or if you decline to save changes when prompted (perhaps you want to test before filing the changes) you will end up with code that is not in SALT yet. You can `)SAVE` the workspace and resume later as all the tagged information is kept unless you deliberately remove it.

With SALT enabled you can pick up all the new code by using `Snap` again or you can use `Save` for an individual item.

Using your code

We've seen how to move code outside the workspace into text files. Now is the time to use that code. The function to bring the code in is `Load`. You give it a filename and it defines the code in the workspace, ready for use. Let's go back to Mike's code.

To bring everything back in we do

```
⎕SE.SALT.Load '\projectX\scripts\*'
```

If namespaces were saved there they will now also appear as namespaces in the root of the workspace.

If other needed namespaces were saved elsewhere they have to be brought in, separately:

```
⎕SE.SALT.Load '\projectX\scripts\APL\GUI\*'
⎕SE.SALT.Load '\projectX\scripts\APL\utils\*'
```

All these objects are tagged by SALT, and editing any of them will get the modifications saved back to file, if you wish. If you do *not* want the objects to be so tagged use the switch `nolink`, e.g.:

```
⎕SE.SALT.Load '\projectX\scripts\* -nolink'
```

You might typically do that in a production environment.

You might also prefer to keep some functions together, saved in a single file (a namespace), but to have them in the workspace (in the root by default) and not in a namespace. You use the `disperse` switch for that:

```
⎕SE.SALT.Load '\projectX\scripts\* -disperse'
```

You can even choose which objects to disperse:

```
⎕SE.SALT.Load '\projectX\scripts\* -disperse=fn1,fn2,opx,varX'
```

This method has the advantage of being able to define variables. A serious disadvantage, though, is that this version of SALT cannot keep track of where the objects came from: changes are *not* picked up and filed automatically.

Tackling variables

SALT cannot tag variables and save them individually on file.

There are two ways around this:

1. Initialise the variables in a function and call the function before doing anything else:

```
           ⎕SE.SALT.Load 'initfn' ⋄ ⎕VR 'initfn'
          ∇ initfn
     [1]   GlobalSetting←'PROD'
          ∇
```

2. Put the variables in a namespace and disperse its contents. If `globalVars` was saved thus:

```
:Namespace globalVars
   GlobalSetting←'PROD'
:EndNamespace
```

then

```
⎕SE.SALT.Load 'globalVars -disperse'
```

would define `GlobalSetting` in the workspace root.

Version control

One of the main reasons for using version control is to be able to go back in time and retrieve previous versions.

Let's say you had code working fine at version 3 and you made a series of changes that have brought you to version 5 and you now realise there is a problem or a difference in behaviour. If you want to check and run that previous code you can retrieve it simply by doing

```
⎕SE.SALT.Load 'mainCode  -version=3'
```

If you'd rather only see what the difference is you can use `Compare`:

```
⎕SE.SALT.Compare 'mainCode  -version=3 5'
```

SALT will use its own code to do the comparison but you prefer to use your favourite file-comparison program located, say, [4] in `[ProgramFiles]\X` then tell SALT [5]:

```
⎕SE.SALT.Compare'mainCode -v=3 5 -use=[ProgramFiles]\X\cm.exe'
```

SALT's way of tracking versions is very simple and each file can have its own version number. SALT has no locking mechanism and does not allow you to 'lock out' files but it will warn you if it detects that a file has been revised when you try to save back a modified object. If you see such a warning and you don't know why, then you should investigate.

For serious version control in a large system you should use a more robust system like SubVersion or CVS. In that case you should *not* be using SALT's versions at all.

If you decide to use versions you should keep in mind that SALT creates a new file each time a modification is made to an object. After a while you might end up with a large number of files, many of which you are uninterested in keeping. For example, let's say you started working on namespace `NmspX` at version 15. After having modified it sixty times you are now at version 75. If you are happy with the result and confident that versions 16 to 74 are useless you can use Explorer to get rid of the files `NmspX.16.dyalog` to `NmspX.74.dyalog`. There will be a gap between 15 and 75 but that might not matter.

If you would like to collapse those sixty versions into one, bringing the good version 75 to a version just above the current good version 15 (i.e. 16) you can get SALT to do it like this:

```
⎕SE.SALT.RemoveVersions 'mainCode  -version=>15 -collapse'
```

SALT will confirm with you the deletion of (here) 60 versions, delete 59 of them and rename the last one to version 16. The collapse switch is used to keep the last version. Without it the last 60 versions would be deleted (which may also be what you want to do).

Epilogue

You might not need SALT or any version control system at all. If all you have is a small system that runs well in 1 or 2 workspaces and needs little or no maintenance then SALT would not be very useful to you.

If you already have your own management system that takes care of all the maintenance problems and you don't need to use text files then, again, SALT would not improve your life much.

But if you don't have such a system, and your code requires a fair amount of maintenance, then SALT can help. It is already there and it is free.

As you have seen, SALT supports many ways to organise your code. It is good at saving and retrieving code, keeping track of versions and managing them. Coupled with a professional external version-control system it could solve many problems.

SALT comes with V12 but will be available online before 2009.

Check it out.

Notes

1. Sourced namespaces are those for which the `⎕SRC` function returns a canonical representation similar to what `⎕NR` returns for functions.

2. There are SALT utilities to do this.

3. You will find those utilities in `lib/NStoScript`. The main function is `Convert`. It takes a namespace as argument and turns it into a sourced namespace.

4. `[ProgramFiles]` is where Windows keeps all programs. SALT will replace it by whatever value is appropriate.

5. SALT uses a consistent syntax for each function: a single string argument that describes the command arguments and the switches that start by a dash. Each switch can be shortened to a non-conflicting length. Here there are no other switches starting with `v` so `-v` is sufficient to denote `-version`.

Functional calculation

1: Numerical ingredients

Neville Holmes, University of Tasmania
Neville.Holmes@utas.edu.au

Introducion

This article is the second in a series (numbered origin-zero *Ed.*) expounding the joys of functional calculation. Functional calculation does with operations applied to functions and numbers what numerical calculation does with functions applied to numbers. The functional notation used as the vehicle in this series is provided by a freely available calculation tool called J. This article reviews the numerical calculation capabilities of J which are the basis for its functional calculation capabilities, and which must be understood before the functional calculation capabilities proper can be understood. The capabilities described in this article will be illustrated and explained in detail in the next article.

Calculation

Calculation is generally reckoned to be the systematic manipulation of numeric values. Our schools, and our pocket calculators, are burdened with the idea that there are four kinds of arithmetic, called addition, subtraction, multiplication and division. The teaching of these traditional four arithmetic functions has not been blessed with success recently, and one of the reasons why it has not might be because there are in fact many arithmetic functions – some simpler than the traditional four, some more complex.

There are also many kinds of numbers, but this is another richness ignored by schools and pocket calculators.

This article reviews both the arithmetic functions provided by the functional calculating language J (as introduced in the first article in this series), and the numbers which are recognised by J. Using these functions, and their numbers, is not in itself functional calculation, but is part of the basis for it.

Numbers

Calculation changes numbers by systematic alteration or by combination. The apparent result of a calculation depends on up to three aspects of the numbers involved:

- what any starting numbers were keyed in as,

- what any numbers were stored as within the calculator, and
- what the result was displayed as.

Of course, it also depends on the functions applied to the numbers.

Number representation

Ideally, a calculator will store all the numbers it has to deal with as well as is possible. Just how they are stored depends on the underlying electronics, which typically used a binary representation that is unsuitable for direct use by humans. Anyone using a calculator should be as unaffected as possible by the method used to store numbers, and any decisions about the kind of representation to be used should be determined by the value to be stored, not by the user.

Although J interpreters shield their users from details of how numbers are stored, they nevertheless have to take numbers in from the keyboard and put them out to the display screen. The only practical way of doing this is using the unfortunate standard ASCII character set, unfortunate because of its poverty, lacking as it does even the × and ÷ symbols! The following table sums up the provisions of the J notation for expressing individual numbers.

`0 1 … 8 9`	decimal digits	`7 364 529081`	
`.`	fraction point	`0.7 3.64 52908.1`	must be digit to left of point
`_`	negative sign	`_7 _364 _52908.1`	underbar, not hyphen
`e`	scale point	`7e0 _3.6e4 5290e81`	scaling basis is ten
`r`	ratio point	`1r7 3.6r4 5290r_81`	

`j`
`ad`
`ar`
	cartesian point
	degree point
	radian point
		`1j7 3.6j4 5e90j_81`
		`1ad7 3.6ad4 5e90ad_81`
		`1ar7 3.6ar4 5e90ar_81`
			real `j` imaginary
			magnitude `ad/ar` angle
			a radian is about 57°

`p` `x`	pi point euler point	`1p2` (π^2) `2r3p_1` (2/3π) `1x2` (e^2) `2r3x_1` (2/3e)	
`b`	base point	`2b1111` (15) `16bff` (255)	

`_`
`_.`
	infinity
	indeterminacy
		`_ (1r0) __ (_1r0)`
		`_. (_-_)`
			these two notations
			are of special numbers

Furthermore, if an integer has an `x` suffix then it is stored exactly and exact arithmetic is used in conjunction with it if possible.

All the most commonly needed numbers can be succinctly expressed, as shown by the examples of the table, but several points need to be explained and emphasised.

- The elements in the list are applied in the sequence given. For example, the scale point is effective before the ratio point. So keying `1e2r3` in gives `33.3333`, while `1r2e3` gives `0.0005`. Also, keying `1.2e3` in by itself will give a display of `1200` but keying `1e2.3` in will display an error message *ill formed number* because the `e` sees `2.3` as its suffixing component but that component must be an integer.
- The elements not separated in the table by a horizontal rule are strict alternatives. So keying `1j1p1` in yields `1j3.14159`, but keying `1x1p1` in yields *ill formed number*.
- The symbol for the negative sign is not the same as the symbol for subtraction. Negativeness is a property, subtraction is a function, and always the twain should be distinct. An overbar is often used for the negative sign, but the ASCII character set lacks an overbar so the underbar must be used.
- The elements following the negative sign in the table are letters of the alphabet. They are, when immediately preceded and followed by numbers of lower form, arithmetic signs and not function symbols. They are, as is the fraction point, infixes, and in this the negative sign is an exception, being a prefix.
- Apart from the above, the letter `x` is used as a suffix to integers to signal that exact arithmetic is to be used on them if possible. This capability is very useful but was added late to the notation and is not yet fully and cleanly worked out.
- The last two lines of the table show some special uses of the underscore symbol, and the symbols given there are individual values, not elements.
- Only the elementary symbols given in the table may be used to compose an individual number. Parentheses and blanks are not allowed within individual numbers, though they may be used to separate individual numbers.

Any number may be keyed in in a variety of ways, but a number will, unless the user specifies a particular format, always be shown on the screen in the same way, however it may have been keyed in or calculated.

Some examples

For the most part, simple numbers like `123 4.5 6e7` and `8.9e10` will give unsurprising results, at least for people with the experience that tells them that the `e` signifies that the number to its left is to be scaled by the power of ten specified by the number to its right. Indeed, these particular numbers will display exactly as they are keyed in.

Some simple numbers, like `1e2` and `3e4`, will display in a simpler form, in this case as `100` and `30000` – not as `30,000`! Should you key `30,000` in, you will get `30 0` back. Keying `1,234` in to yield `1 234` seems less strange, but even that is not what it might seem to be, because keying `1+1,234` gives `2 235`. The explanation for this is that the comma is a function symbol in J, here standing for a function usually called *catenate*, so `1,234` specifies that the `1` is to be catenated with the `234` to give the two numbers `1` and `234`. So `2+30,000` will give `32 2`, but notice (this will be explained later) that `30,000+2` will give `30 2`.

Numeric functions

The numeric functions known as *scalar functions* are applied to one or two numbers and produce a single number as a result. The *primitive functions* are those functions which have a simple symbol for a name. Some of the scalar functions provided by J as primitive functions are given in the following table.

`+`	conjugate	add	`+.`		GCD	`+:`	double	nor
`-`	negate	subtract	`-.`	not		`-:`	halve	
`*`	signum	times	`*.`		LCM	`*:`	square	nand
`%`	reciprocal	divide				`%:`	square root	root
`|`	magnitude	residue						
`^`	exp	power	`^.`	$\log_e$	logarithm			
`=`		equal	`=.`		(local is)	`=:`		(global is)
`<`		less than	`<.`	floor	lesser	`<:`	decrement	not more
`>`		more than	`>.`	ceiling	greater	`>:`	increment	not less
						`~:`		not equal
`!`	factorial	choices						
`?`	roll		`o.`	pi times	circular	`p:`	prime	
`[`	(same)	(left)						
`]`	(same)	(right)	`".`	(do)		`":`	(format)	(format)

The functions are given by name, a name which is meant to suggest what the function does. If there is any doubt about this, experiment with the interpreter can be used to dispel the doubt. Again, some explanation is necessary to clarify the table.

- The dot (`.`) and colon (`:`) are used as character set extenders, and, when suffixed to a plain function symbol, effectively provide a new function symbol, though the symbols that differ only in their suffix are usually related in some way.
- A primitive function symbol may stand for two different functions, which is why there are two columns of names after the function symbol in the table. The name on the left is for a monadic function, that is, for a function which only has one argument, a right argument. The name on the right is for a dyadic function, that is, for a function which has two arguments, one on its left and one on its right.
- Some functions are restricted in the results they can produce. In particular, dyadic `= < > ~: <:` and `>:` can only produce a `0` or a `1`, though these results are ordinary numbers in the sense that there is no restriction on their use in further calculations.
- Some functions are restricted in the arguments they will take. For example, monadic `?` and `p:` take in only non-negative integers.
- The symbols `=.` and `=:` do not stand for functions at all. They are called *copulas* and are used for naming results. The name to the left of the copula is given to the value on the right, whatever that might happen to be. These symbols are sometimes called *gets* or *becomes*, and the global version should normally be used.
- Otherwise, the functions whose names are given in parentheses are not scalar functions, but are useful in connection with them. The functions that the brackets `[` and `]` stand for are very useful, though their result is always one of their arguments. The *format* function converts its right argument into a character string, optionally under control of its left argument, while the *do* function can convert the character string back to a number, though often not the same number.
- The circular functions have left arguments restricted to the values shown in the table below, and, where relevant, their right arguments are taken as radians. Actually, these functions include hyperbolic and pythagorean functions, as shown in the following table. Note that, in the following table, the left argument constant is separated from its function symbol by a blank, which is necessary to avoid the error message caused by an attempt to interpret the `o.` as part of a constant, the round character in the `o.` symbol being a lower case *o*. This is

not the case with the other function symbols introduced so far, except `p:`, because they use nonalphabetic characters.

						`0 o.`	sqrt(1-x^2)
`1 o.`	sine	`2 o.`	cosine	`3 o.`	tangent	`4 o.`	sqrt(1+x^2)
`5 o.`	sinh	`6 o.`	cosh	`7 o.`	tanh	`8 o.`	sqrt(_1-x^2)
`_1 o.`	arcsine	`_2 o.`	arccosine	`_3 o.`	arctangent	`_4 o.`	sqrt(_1+x^2)
`_5 o.`	arcsinh	`_6 o.`	arccosh	`_7 o.`	arctanh	`_8 o.`	sqrt(_1-x^2)

The hierarchy of functions

Clearly, J provides many primitive functions, as well as the ability to give a name to any, by the expression `sqrt=:%:` for instance. Dealing with these cleanly leads to a break with tradition that provokes a strong rejection from arithmetic traditionalists.

The primitive functions are traditionally considered to be arranged in a hierarchy of strength and direction, and the higher primitive functions are represented by notational peculiarities which imply a hierarchy. For J, the richness of primitive functions and the uniformity of expression forced by the ASCII character set and the linearity of its use, make a hierarchy practically impossible. Thus, `5+%|x` stands for *five plus the reciprocal of the magnitude of x*, while `a+7*b` stands for *a plus the product of seven and b*.

The absence of a hierarchy of functions leads to expressions having meanings which, though completely reasonable, are upsettingly untraditional. For example, `a*7+b` stands for *a times the sum of seven and b*, while `a-7-b` stands for *a minus the subtraction of b from seven*. If the traditional meanings are required for the arguments in that sequence, then parentheses must be used in J, as `(a-7)-b` and `(a*7)+b`. The gain is simplicity, the price is a break with tradition.

Summary

This article is like a list of ingredients that can be used for numerical calculation using the notation provided by J. At their simplest, these ingredients can be put together in the manner learned in elementary school, because the simplest of the expressions learned in elementary school, expressions like `1+2` and `7.2-5.75`, can be keyed in to the interpreter to give the result expected in elementary school.

However, the need to expand the number of elementary functions available, coupled to the restrictions of the ASCII character set, mean that some calculations and their results will not be quite like the results expected from elementary school. Nevertheless the changes are consistent and systematic, and their adoption allows elementary calculation to be extended in elementary ways, ways which allow simple use of an interpreter to evaluate expressions.

The next article in this series will explain and illustrate how numerical calculation is done with the J interpreter, as a preliminary to further treatment of functional calculation.

Footnote: This essay was written a decade ago to introduce J to classes of honours students as explained in the introductory essay "Tacit J and I". No attempt has been made to upgrade the text to incorporate subsequent changes to J, but I believe that the original design of J was exceptionally robust and will not have causes errors to crop up in the description above.

The ruler's edge revisited

In Session

Ray Polivka
polivkar@acm.org

"What a great inquisitive programming problem to give to a APL programming class!" was the thought that entered my mind when I read Stephen Taylor's article "The ruler's edge" that appeared on pages 118-120 in the January 2008 issue of *Vector*. I have recently taken a fancy to the term *inquisitive programming* as opposed to *software development programming*. The term was coined by Brian Hayes in his excellent paper entitled "Calculemus" in the *American Scientist* Vol.96 N°5, September-October 2008. One could also say that *inquisitive programming* is another term for personal problem solving. So be it. But this is the initial path that students should take when either just learning to program or learning a new programming language. Stephen's problem is a fine example to offer along this path.

Since I am a person who learns by doing, I decided to create my own solution. I wrote it in conventional defined-function form in APL2, since I am not very agile with the dynamic-function notation. Furthermore, I will be able to use it with any of the APL vendor's APL systems. Here are my two slightly different solutions.

```
[0]   Z←D RULERA L;I;N;T;IO;PW
[1]  ⍝L: Length of the ruler
[2]  ⍝D: Distance between ticks
[3]  ⍝Z: a ruler of length L with ticks every D places
[4]  ⍝   modelled after S Taylor's problem in Vector
[5]   ⎕PW←L+⎕IO←1
[6]   Z←Lρ((D-1),1)/'¯∧'
[7]   I←(Z='∧')/⍳L
[8]   N←⍉⍕,['']I
[9]   T←((↑ρN),L)ρ' '
[10]  T←[;I]←N
[11]  Z←T,[1]Z
    ∇ 2008-09-18 19.03.04 (GMT-4)
```

Or with a different ending:

```
[0]   Z←D RULERB L;I;N;T;IO;PW
      …
[9]   E←Lρ((D-1)ρ0),1
[10]  T←E\[2]N
[11]  Z←T,[1]Z
    ∇ 2008-09-18 19.03.19 (GMT-4)
```

Actually, I modified Stephen's problem slightly. Rather than having the number straddle the tick mark, the numbers appear vertically about the tick mark. This allows one to display a number every position in the ruler if one so chose. Thus,

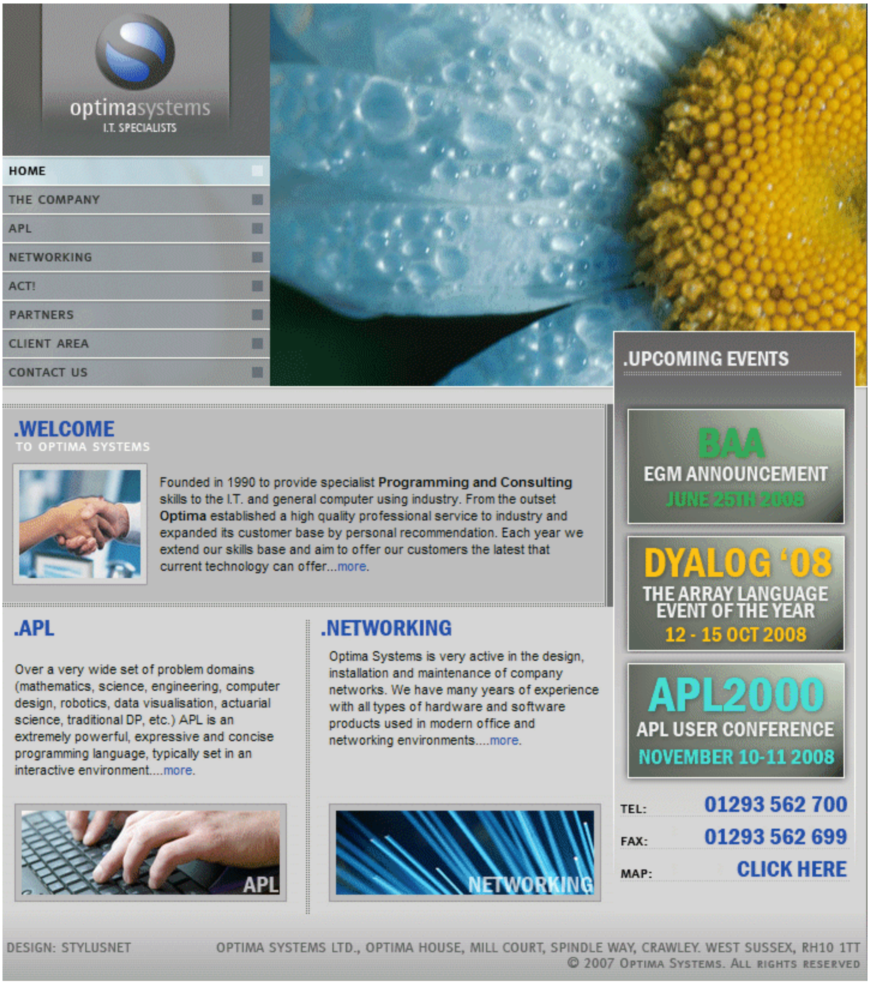
```
      3 RULERA 17
           1  1
  3  6  9  2  5
--^--^--^--^--^--
```

There are several ways in which this problem might be used in an APL programming class. In a more advanced class one could just state the problem and see what comes back. However, when the given solution illustrates some perhaps unusual approach or less-used aspects of APL, then presenting the solution directly may be better for an in-class discussion. Or one could present a solution with just comment symbols following all or selected lines of the function. The student now is asked to fill in a set of comments. This way the student can determine what the function does at specific lines while determining how the function achieves its purpose. Then too, the student can use the available debugging tools to ferret out the behaviour of the function. For the Dyalog APL programmers, another exercise might be to translate the defined function into its dynamic function form too.

As an additional exercise, the student could be asked to modify the function so that the ruler prints vertically using `|` and `+`. Thus

```
      3 RULERC 7
 |
 |
3+
 |
 |
6+
 |
```

This certainly can be a fun learning exercise, Stephen, as well as a morning honing exercise. Now where is my cup of tea?



.WELCOME
TO OPTIMA SYSTEMS

Founded in 1990 to provide specialist **Programming and Consulting** skills to the I.T. and general computer using industry. From the outset **Optima** established a high quality professional service to industry and expanded its customer base by personal recommendation. Each year we extend our skills base and aim to offer our customers the latest that current technology can offer...more.

.APL

Over a very wide set of problem domains (mathematics, science, engineering, computer design, robotics, data visualisation, actuarial science, traditional DP, etc.) APL is an extremely powerful, expressive and concise programming language, typically set in an interactive environment....more.

.NETWORKING

Optima Systems is very active in the design, installation and maintenance of company networks. We have many years of experience with all types of hardware and software products used in modern office and networking environments....more.

.UPCOMING EVENTS

BAA
EGM ANNOUNCEMENT
JUNE 25TH 2008

DYALOG '08
THE ARRAY LANGUAGE EVENT OF THE YEAR
12 - 15 OCT 2008

APL2000
APL USER CONFERENCE
NOVEMBER 10-11 2008

TEL:	01293 562 700
FAX:	01293 562 699
MAP:	CLICK HERE

DESIGN: STYLUSNET

OPTIMA SYSTEMS LTD., OPTIMA HOUSE, MILL COURT, SPINDLE WAY, CRAWLEY. WEST SUSSEX, RH10 1TT
© 2007 Optima Systems. All rights reserved

If you think J is complex try j

J-ottings 50

Norman Thomson

Abstract
This article is about the facilities available in J for handling complex numbers, something which is greatly helped by a few simple diagrams.

1. The two complex number constructors

Although complex numbers are readily input using numeric constants, e.g. `12.5j_7.9`, in meaningful applications the components are more likely to be expressions from which complex numbers are *constructed*. J has two tools for constructing complex numbers, namely `j.` and `r.` These correspond to their two possible representations, namely as 2-lists of Cartesian (that is x-y) coordinates, and as 2-lists of polar coordinates (that is {length, angle}). The way in which `j.` and `r.` work is illustrated by

```
   (2 j.1),(2 r.1)  NB. the two complex no. constructors
2j1 1.0806j1.68294
```

The second of these results shows that 2 times the coordinates of the end point of a radius of the unit circle at an angle of 1 radian are approximately (1.08, 1.68). For primary input in the form of 2-lists use *insert*:

```
   (j./2 1),(r./2 1)
2j1 1.0806j1.68294
```

Informally, `j.` compresses x-y coordinates into complex numbers, and `r.` converts polar representation to complex number form. Monadic `j.` is dyadic `j.` with a default left argument of 0, while monadic `r.` is dyadic `r.` with a default left argument of 1. It is not a coincidence that these defaults are the identity elements of addition and multiplication. `r.k` where `k` is real returns the Cartesian coordinates of the point on the unit circle whose polar coordinates are (1,k), for example

```
   r.1      NB. coords of radius at 1 radian
0.540302j0.841471
```

`r.k` is represented by e^{ik} in maths and thus by `^j.k` in J, an operation also available through the circle verb as `_12 o.k`. The fact that `r.` and `^j.` are synonyms links the two complex number constructors. More generally the circle functions with arguments in the ranges {9…12} and {_9…_12} are directly relevant to complex number construction, and they too have synonyms which will emerge as the discussion continues. The equivalence of `r.` and `^j.` will come to the fore later in section 6 when discussing complex powers.

2. The complex number deconstructors

The construction process is reversed (that is complex numbers are converted back to 2-lists) by `+.` for Cartesian coordinates and `*.` for polar coordinates:

```
   +.2j1 1.0806j1.682941    NB. +. reverses j./
     2       1
 1.0806 1.68294
   *.2j1 1.0806j1.682941    NB. *. reverses r./
2.23607 0.463648
     2        1
```

The *circle* verb provides the opportunity to obtain the components of `+.` and `*.` one by one:

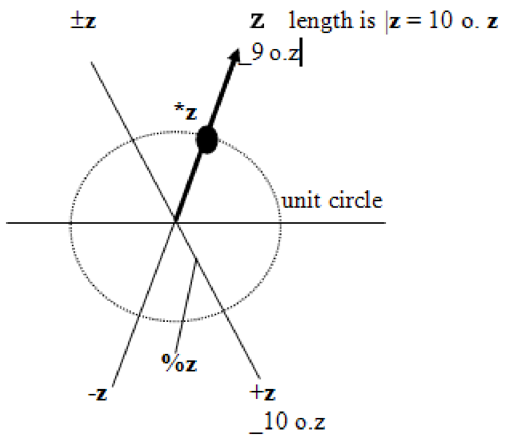
```
   9 11 o.2j1               NB. 9 o. is x. , 11 o. is y
2 1
   10 12 o.2j1              NB. 10 o. is length, 12 o. is angle
2.23607 0.463648
```

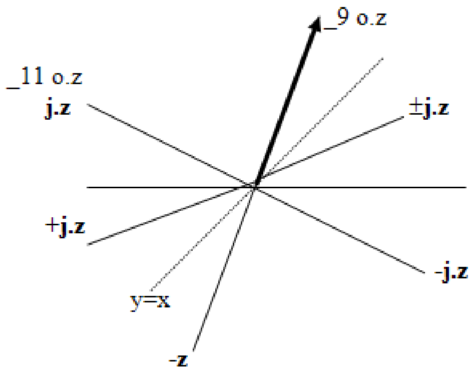
The following is a ‘rule of thumb’ table which summarises the meanings of the circle verbs and incorporates the above ideas :

n o.			n o.
_9	identity		
_10	conjugate		
	construct	deconstruct	
_11	j.	+.	9, 11
_12	r.(^j.)	*.	10, 12

3. Monadic operations with complex numbers

J provides alternative routes for several common complex numbers operations. In the diagrams below, a complex number z is represented by the arrowed line, and other points represent the results of the fundamental monadic arithmetic operations of addition, subtraction, multiplication and division, separately and in combination with `j.` as well as those of the circle functions which are synonyms.





The symbol `±` is used here to denote either of the verbs `+@-@j.` or `-@+@j.` since they are equivalent. The points `z ±z -z +z` represent a rectangle formed by reflections in the x and y axes with vertices visited anti-clockwise, while the points `j.z ±j.z -j.z +j.z` represent a rectangle formed by reflections in the diagonal axes with vertices visited clockwise. In addition to the three circle function synonyms shown for circle functions, `_12 o.` is a synonym for `r.` as noted earlier.

The symmetries of rectangles can be represented by groups of verbs of order 4 in which `I` is the identity verb :

Rotations	`{I - j. -j.}`	`j. -j.` are anticlockwise/clockwise rotations of π/4
Reflections	`{I - + ±}`	`+ ±` are reflections in main axes,
	`{I - ±@j. +@-j.}`	`±@j. +@-j.` are reflections in diagonal axes

From these as starting points the full order-8 group table for the symmetries of the square can easily be obtained.

4. Basic dyadic operations

The basic operations `+ - * %` behave as expected, and rules such as the following are obeyed:

```
   |5j2*3j4                    NB. modulus of a product is ..
26.9258
   (|5j2)*(|3j4)               NB. .. the product of moduli
26.9258

   12 o. 5j2*3j4               NB. the angle of a product ..
1.3078
   +/12 o. 5j2 3j4             NB. .. is the sum of the angles
1.3078
```

Also

```
   +/5j2*3j4
7j26
```

is equivalent numerically to

$$\begin{vmatrix} 5 & -2 \\ 2 & 5 \end{vmatrix} \quad \begin{vmatrix} 3 \\ 4 \end{vmatrix} \quad = \quad \begin{vmatrix} 7 \\ 26 \end{vmatrix}$$

and

$$\begin{vmatrix} 3 & -4 \\ 4 & 3 \end{vmatrix} \quad \begin{vmatrix} 5 \\ 2 \end{vmatrix} \quad = \quad \begin{vmatrix} 7 \\ 26 \end{vmatrix}$$

showing that multiplication of complex numbers is equivalent to the inner product `+/ .*` for matrices of the form

$$\begin{vmatrix} a & -b \\ b & a \end{vmatrix}$$

When multiplication takes the angle outside the range [-π,π] `*.` and `12 o.` automatically make a wraparound to bring the angle back into range, for example

```
wrap=.3 :0
t=.(o.2)|y.
if.t>o.1 do.t=.t-o.2 end.
)
    +/12 o._5j2 _3j4        NB. sum of angles exceeds pi
4.97538
   12 o._5j2*_3j4
_1.3078
   wrap 4.97538             NB. 4.975.. + 1.307.. = 2pi
_1.3078
```

Complex numbers raised to real powers are the subject of de Moivre's theorem. This depends on the fundamental relation

$$e^{i\theta} = \cos\theta + i\sin\theta$$

which in J expresses the fact that the verbs `^@j.` and `j./@(2 1&o.)` are equivalent. De Moivre's theorem says that

$$(re^{i\theta})^n = r^n e^{in\theta} = r^n \{\cos n\theta + i\sin n\theta\}$$

so to raise complex z to the power n its modulus should be raised to the power n and its angle multiplied by n. To see this in action raise `2j1` to the powers 1, 2 and 8 in first Cartesian and then polar form:

```
   +.2j1^1 2 8
   2    1                   NB. modulus = sqrt(5)
   3    4                   NB. modulus = 5
_527 _336                   NB. modulus = 625 = 5^4

   *.2j1^1 2 8
2.23607 0.463648            NB. (length,angle) for 2j1
      5 0.927295            NB. (length,angle) for 2j1 ^2
    625   _2.574            NB. (length,angle) for 2j1 ^8
```

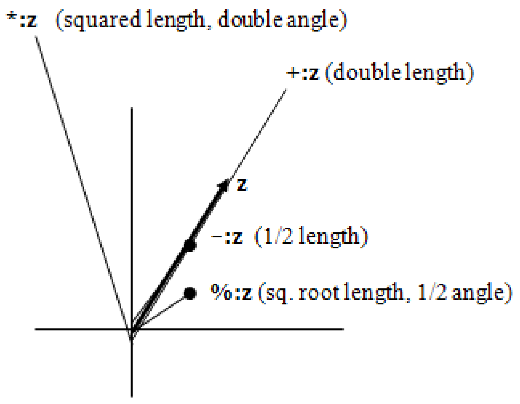
The angle in the last line above can be confirmed by

```
   wrap 8*0.463648
_2.574
```

It may be tempting to use the circle function `_3 o.` *arctan* to obtain angles, but this only works in simple cases because the range of arctan is [-π/2, π/2]. The range for complex numbers is double this because arctan makes no distinction at all between (-x)/y and x/(-y), whereas the difference between second and fourth quadrants *is* significant in dealing with complex numbers.

5. The enhanced arithmetic operations

The second diagram illustrates the actions of the J verbs which are obtained from the basic arithmetic operations by adding 'colon' to make a di-gram:


*:z (squared length, double angle)
+:z (double length)
z
-:z (1/2 length)
%:z (sq. root length, 1/2 angle)

Only one of these four forms – namely `%:` – extends to the dyadic case, for example `4%:z` means (4th root of length, ¼ angle).

In the case of di-grams formed by adding full-stop `1-.z` is the same as with real numbers and `%.` and `%` are exactly equivalent. If either `*.` or `+.` are applied to real scalar numbers the results are 2-lists made by joining zeros. This can be used as a method of stitching 0s as in

```
   +.3 4
3 0
4 0
```

With real numbers dyadic `*.` and `+.` are LCM and GCD respectively, but these should not be used with complex arguments in the expectation of obtaining the separate GCDs and LCMs of their components. For example

```
   (5j6 +. 10j3),(5j6 *. 10j3)
0j1 75j_32
```

6. Complex powers and logarithms

To understand complex powers, start with the synonym relationship between `r.` and `^j.` (or `_12 o.`), which at first sight should lead to `j.` and `^.r.` also being synonyms. This is indeed true in some cases:

```
   (j.2j1),(^.r.2j1)
_1j2 _1j2
```

but not always:

```
   (j.12j10),(^.r.12j10)
_10j12 _10j_0.566371
```

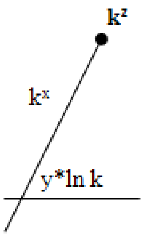
This is because, unlike in real arithmetic, the logarithm of a complex number is not a single-valued function. In Cartesian coordinates, x and y values stretch out indefinitely in both directions, but in polar coordinates angles wrap around in cycles of 2π in the manner defined by the verb `wrap` above. In mathematical notation,

$$\ln(z) = \ln(re^{\pi i\theta}) = \ln(r) + i(\theta + 2k\pi)$$

where k is an integer. As a matter of arbitrary (but natural!) choice, J returns the unique angle which lies in the range [-π, π]. The same wrapping process applies when real numbers are raised to complex powers :

```
   *.2^4j4.5 4j4.6
16  3.11916                 NB. unwrapped
16 _3.09471                 NB. wrapped
```

More specifically, if k is real and z=x+iy then k^z is illustrated by



```
   *.2^2j1 3j2
4 0.693147                  NB. k to power x, y times ln(k)
8  1.38629                  NB. with angle doubled
```

The cases 'complex raised to real' (de Moivre's theorem) and 'real raised to complex' have now been covered, leaving only the case 'complex raised to complex' to be dealt with. An interesting starting point is the number i^i which at first sight should be about as complex as it gets:

```
   0j1^0j1                  NB. i to the power i
0.20788
```

Not so! To explain this result, consider first ln(i^i)=i×ln(i). Using the formula

$$\ln (re^{\pi i\theta}) = \ln(r) + i(\theta + 2k\pi)$$

and choosing k=0 (as J does) to make the logarithm single-valued, gives

$$\ln i = 0 + i\pi/2$$

which, when multipled by i, gives -π/2.

i^i must therefore be $e^{-\pi/2}$, which has the value 0.20788 to 5 decimal places. This sequence of calculations is confirmed by

```
   (^.0j1), (^.0j1^0j1), (^-o.0.5)
0j1.5708 _1.5708 0.20788
```

Here is the i^i calculation spelt out in a single line:

```
   ln=.(^.@{. , }.)@*.      NB. log length, angle
   (^@*j./@ln)0j1           NB. i to the power i
0.20788
```

The verb `ln` fulfils the familiar 'reduce multiplication to addition' property of logarithms of real numbers, that is log(ab) = log a + log b, for example:

```
   ln 1j2*3j4               NB. ln ab
2.41416 2.03444
   +/ln 1j2 3j4             NB. ln a + ln b
2.41416 2.03444
```

For powers where both w and z are fully complex (that is, have non-zero imaginary parts) the following sequence of equivalences

$$w^z = (e^{\ln w})^z = (e^z)^{\ln w} = e^{z\ln w}$$

leads to, for example

```
   2j1^1j3                  NB. 2j1 to the power 1j3
_0.537177j0.145082

   ^1j3*j./ln 2j1           NB. e to the power ln w
_0.537177j0.145082
```

It is not easy, perhaps impossible, to visualise the relationship of wz to w and z diagrammatically, that is the link of `_0.537177j0.145082` with `2j1` and `1j3` is a numerical rather than a graphical one. The same is true for other functions which can accept complex arguments, for examples trig ratios and their inverses. Also the logarithms concerned must be to the base e. e is one of the five most fundamental numbers in the universe, namely 0, 1, e, π and i, which are connected by the equation $1+e^{\pi i}=0$. It is reasonable to suppose that advanced intelligent communicators from outer space (if such there be) would certainly try to convey this set of numbers to us as an immediate *lingua franca*. This equation can be expressed in J in the following three equivalent ways:

```
   (1+^o.j.1),  (1+_12 o.o.1),  (1+r.o.1)
0 0 0
```

The equation 1+eπi=0 can be rewritten ln(-1)=πi, which, since ln(-r)=ln(r)+ln(-1), means that the natural logarithms of negative real numbers are obtained by appending `j`π to the logarithm of the corresponding positive number. For example:

```
   ^.5.2 _5.2   NB. (ln r), (ln -r)
1.64866 1.64866j3.14159
```

7. Extension to quaternions

Given a matrix of the form

$$\begin{vmatrix} a & -b \\ b & a \end{vmatrix}$$

where a and b are real numbers, e.g.

```
   M=.2 2$2 _3 3 2
```

and an inner product of `M` with a 2-list such as

```
   M +/ .* 2 _1
7 4
```

the same information could be obtained by multiplying two complex scalars:

```
   2j3*2j_1
7j4
```

Similarly finding the determinant of `M` is equivalent to a couple of operations on a complex scalar:

```
   (det=.-/ .*)M            NB. determinant of M
13
   *:10 o.2j3               NB. sum of squares of 2 and 3
13
```

Thus complex numbers can be used as a means of reducing the rank level of some operations. An ‘obvious’ question is then: if complex data can reduce rank-2 operations to rank 1, can it correspondingly reduce rank-3 operations to rank 2?

This speculation is not unique to J, in fact the question was answered in the mid-19th century by Sir William Hamilton, who discovered that this progression is multiplicative rather than additive; that is, that the next step up is not from 2 to 3 but from 2 to 4.

First define a verb which transfoms a 2-list of real scalars into a matrix of the above form:

```
   r2ltom=.,._1 1&*@|.      NB. matrix from real 2-list
   r2ltom 2 3
2 _3
3  2
```

Next observe that is possible to have a matrix with complex coefficients which nevertheless has a real determinant, for example:

```
   C =. 2 2$4j3 6j_2 _6j2 4j3
   det C
39
```

The matrix `C` has the form

$$\begin{vmatrix} PjQ & (-R)j-S \\ RjS & PjQ \end{vmatrix}$$

which is the form

$$\begin{vmatrix} a & -b \\ b & a \end{vmatrix}$$

extended to complex elements. Straightforward arithmetic shows that the determinant of a matrix of the above form has a real part ($P^2-Q^2+R^2-S^2$) and an imaginary part 2(PQ+RS) and so, if PQ = –RS, as is the case with `C`, the determinant is real, otherwise not. In the ‘all real’ case, Q = S = 0 and `det`(M) = P^2-R^2.

If the form is now changed to

$$\begin{vmatrix} \mathrm{PjQ} & (-\mathrm{R})\mathrm{jS} \\ \mathrm{RjS} & \mathrm{Pj}-\mathrm{Q} \end{vmatrix}$$

that is,

$$\begin{vmatrix} a & -b \\ b & a \end{vmatrix}$$

where the dashes denote complex conjugates, then the determinant is arithmetically guaranteed to be real for *all* values of P, Q, R and S. If this is now taken as a standard form then four fundamental matrices obtainable by setting each of P, Q, R and S to 1 and the other three to zero correspond to unit points on the axes of a four-dimensional geometrical space as denoted by (1,0), (0,1), (i,0) and (0,i). A verb analogous to `r2ltom` which constructs the above matrix from a 2-list of complex scalars is

```
   c2ltom=.,.+@|.@(*&1 _1) NB. matrix from complex 2-list
   c2ltom 2j3 4j5
 2j3 _4j5
 4j5 2j_3
```

Define variables to correspond to (1,0), (0,1), (i,0) and (0,i):

```
   ]'I i j k'=.c2ltom &.> 1 0 ;0 1;0j1 0;0 0j1
 +---+----+-------+-------+
 |1 0|0 _1|0j1    0|  0 0j1|
 |0 1|1  0|  0 0j_1|0j1   0|
 +---+----+-------+-------+
```

and self-multiply each of these matrices:

```
   times=.+/ .* &.>          NB. matrix multiply
   times~ i;j;k              NB. squares of i, j and k
 +-----+-----+-----+
 |_1  0|_1  0|_1  0|
 | 0 _1| 0 _1| 0 _1|
 +-----+-----+-----+
   times~^:2 i;j;k           NB. 4th powers of i, j and k
 +---+---+---+
 |1 0|1 0|1 0|
 |0 1|0 1|0 1|
 +---+---+---+
```

which shows that $i^2 = j^2 = k^2 = -I$ and $i^4 = j^4 = k^4 = I$ where I is the identity matrix. Now multiply i, j and k by each other:

```
   (i;j;k)times(j;k;i)      NB. result is k;i;j
+-------+----+--------+
|  0 0j1|0 _1|0j1    0|
|0j1   0|1  0|  0 0j_1|
+-------+----+--------+

   (j;k;i)times(i;j;k)      NB. result is (-k);(-i);(-j)
+--------+----+--------+
|   0 0j_1| 0 1|0j_1   0|
|0j_1    0|_1 0|   0 0j1|
+--------+----+--------+
   det&> i;j;k              NB. determinants equal 1
1 1 1
```

If i, j and k are raised to third powers a further three matrices not previously encountered arise:

```
   ]'ci cj ck'=.(i;j;k)times(i;j;k)times i;j;k
+----+--------+---------+
| 0 1|0j_1   0|   0 0j_1|
|_1 0|   0 0j1|0j_1    0|
+----+--------+---------+
```

but now, however much the set of eight matrices `I, -I, i, j, k, cj, ck, ci` are intermultiplied, the result is always another member of the set, for example:

```
   (i;j;k)times cj;ck;ci    NB. result is ck;ci;cj
+--------+----+--------+
|   0 0j_1| 0 1|0j_1   0|
|0j_1    0|_1 0|   0 0j1|
+--------+----+--------+
```

The set of eight matrices possesses the properties of a *group*, more specifically the *quaternion group*. Although there are eight elements in the group these are all related to each other, and the whole set of seven excluding the identity matrix can be generated from any two. For example if `i` and `j` are chosen as generators, `k` is `i j`, `ci` and `cj` are defined as powers of `i` and `j`, and `ck` is `cj` multiplied by `ci`. The seventh matrix is the common value of i^2 and j^2.

If, analogous to `j`, J were to contain two further independent number constructors `k` and `m` such that `2j3k4m5` were a scalar, and the same rules $i^2 = j^2 = k^2 = -I$ and $i^4 = j^4 = k^4 = I$ applied where I = `1`, i = `0j1k0m0`, j = `0j0k1m0`, k = `0j0k0m1` then these scalars would be recognised mathematically as *hypercomplex numbers*. The four basic hypercomplex numbers for which the real elements are (1 0 0 0), (0 1 0 0), (0 0 1 0) and (0 0 0 1), would follow the same multiplication structure as the set of eight matrices, and form a group isomorphic with the quaternion group.

A scalar such as `0j3k4m5` whose first element is zero corresponds to a *pure quaternion* and so pure quaternions exactly match points in three-dimensional space, or quantities such as E, B, H which define electro-magnetic fields as three-dimensional entities. Such equivalences are of course only useful if the operations employed on them are guaranteed to produce further pure quaternions.

PROFIT

Partitions of numbers

An efficient algorithm in J

R.E. Boss
r.e.boss@planet.nl

Abstract
A partition is a way of writing an integer as a sum of positive integers. [1] [2] So 5=3+1+1 is a partition of 5. The subject of this paper is an efficient algorithm in J [3] to generate all partitions of a given number.

Introduction

This analysis was inspired by a remark of Roger Hui [4] “This computation of partitions in 3 lines is the neatest I have seen over the years (although it is not the most efficient).”

Since you can add any number of zeros to a partition, the zeros are left out. Furthermore, in this paper a partition is given in non-ascending order, to identify the uniqueness of the partition. A partition can usually be given as a sequence of the (smaller) numbers with any separator, such as 3,1,1.

If you start generating partitions, you get for the numbers 1, 2 and 3 the partitions 1; 1 1 and 2; 1 1 1, 2 1 and 3 respectively.

Let $P(n)$ be the set of all partitions of n and $P(n,k)$ be the partitions of n starting with k or less [5]. Obviously $P(n,n) = P(n)$. In formula this looks like

$$P(n) = \{\ x = (x_0, x_1, \dots) \mid n \geq x_0 \geq x_1 \geq x_2 \geq \dots \geq 0;\ n = \Sigma x_i\ \}$$

and

$$P(n,k) = \{\ x = (x_0, x_1, \dots) \mid k \geq x_0 \geq x_1 \geq x_2 \geq \dots \geq 0;\ n = \Sigma x_i\ \}$$

We can group the partitions of n according to their starting number, which can be 1 to n. If a partition of n starts with number, say k, the rest is a partition of n-k starting with number k or less. This can be put in formula (A) as follows.

(A) $P(n) = \{\ (k, x_0, x_1, \dots) \mid 0 < k \leq n;\ x \in P(n\text{-}k,k)\ \}$ where $x = (x_0, x_1, \dots)$

Notice that for $k \geq ½n$ we have $k \geq n\text{-}k$, so $P(n\text{-}k,k) = P(n\text{-}k)$.

We develop a J program, based on this analysis.

Using smaller partitions

In generating *P(n)*, we suppose for all *0<k≤m<n* that *P(m,k)* is known already. Applying equation (A), that is, for all *0<k≤n* we generate *P(n,k)* from *P(n-k,k)* by prefixing *k*, and since *P(n,n) = P(n)*, we are done.

Assume the partitions of *n* to be boxed according to the starting number; see Fig. 1. So we start (at the right hand side) with `<<i. 1 0` being the partition of 0, followed by the partition of 1 etc.

Now suppose we have a list of boxes, each box containing the (boxed) partitions, as in Fig. 1, and we want to generate the partitions of the next number, *n=5*. This is done in a few steps.

Figure 1.

```
+---------+-------+-----+---+--+
|+-------+|+-----+|+---+|+-+|++|
||1 1 1 1|||1 1 1|||1 1|||1||||
|+-------+|+-----+|+---+|+-+|++|
||2 1 1  |||2 1  |||2  ||   |  |
||2 2 0  ||+-----+|+---+|   |  |
|+-------+||3    ||     |   |  |
||3 1    ||+-----+|     |   |  |
|+-------+|       |     |   |  |
||4      ||       |     |   |  |
|+-------+|       |     |   |  |
+---------+-------+-----+---+--+
```

First we generate the list of starting numbers of the partitions of 5, here 1…5, which is produced by `>:@i.@#` applied to the given list of boxes. This will be the left argument of our verb to be developed, the right argument being the list of boxes. See Fig. 2, where both arguments are depicted.

Figure 2.

```
            +---------+-------+-----+---+--+
            |+-------+|+-----+|+---+|+-+|++|
1 2 3 4 5   ||1 1 1 1|||1 1 1|||1 1|||1||||
            |+-------+|+-----+|+---+|+-+|++|
            ||2 1 1  |||2 1  |||2  ||   |  |
            ||2 2 0  ||+-----+|+---+|   |  |
            |+-------+||3    ||     |   |  |
            ||3 1    ||+-----+|     |   |  |
            |+-------+|       |     |   |  |
            ||4      ||       |     |   |  |
            |+-------+|       |     |   |  |
            +---------+-------+-----+---+--+
```

Second, with these left and right arguments, the numbers left are matched to the boxes right and within that box, this number is used to select the first partitions. So from the first, only the first box is selected, from the second the first 2 etc. Of course, no more sub boxes can be selected than available. So the selections are done by `(<.#){.]` and are visualised in Fig. 3.

Figure 3.

```
+---------+-------+-----+---+--+
|+-------+|+-----+|+---+|+-+|++|
||1 1 1 1|||1 1 1|||1 1|||1||||
|+-------+|+-----+|+---+|+-+|++|
|         ||2 1  |||2  ||   |  |
|         |+-----+|+---+|   |  |
+---------+-------+-----+---+--+
```

Then the numbers of the left argument are appended with the content of the selected partitions in the boxes by `>:@i.@# ([ ,.&.> (<.#){.])&.> ]` producing the boxes as in Fig. 4.

Figure 4.

```
+-----------+---------+-------+-----+---+
|+---------+|+-------+|+-----+|+---+|+-+|
||1 1 1 1 1|||2 1 1 1|||3 1 1|||4 1|||5||
|+---------+|+-------+|+-----+|+---+|+-+|
|           ||2 2 1  |||3 2  ||     |   |
|           |+-------+|+-----+|     |   |
+-----------+---------+-------+-----+---+
```

But the items in the sub boxes should be taken together since they start with the same number, which is done by `>:@i.@# ;@ ([,.&.> (<.#){.])&.> ]` resulting in the output from Fig. 5.

Figure 5.

```
+---------+
|1 1 1 1 1|
+---------+
|2 1 1 1  |
|2 2 1 0  |
+---------+
|3 1 1    |
|3 2 0    |
+---------+
|4 1      |
+---------+
|5        |
+---------+
```

Now this outcome is exactly the set of partitions of 5 in the required form, so must be boxed and then can be prefixed to the boxes from which we started. Then we are in the same situation, but now with one number further, see Fig. 6.

Figure 6.

```
+----------+--------+------+-----+---+--+
|+---------+|+-------+|+-----+|+---+|+-+|++|
||1 1 1 1 1|||1 1 1 1|||1 1 1|||1 1|||1||||
|+---------+|+-------+|+-----+|+---+|+-+|++|
||2 1 1 1  |||2 1 1  |||2 1  |||2  ||   |  |
||2 2 1 0  |||2 2 0  ||+-----+|+---+|   |  |
|+---------+|+-------+||3     ||     |   |  |
||3 1 1    |||3 1    ||+-----+|     |   |  |
||3 2 0    ||+-------+|       |     |   |  |
|+---------+||4      ||       |     |   |  |
||4 1      ||+-------+|       |     |   |  |
|+---------+|        |       |     |   |  |
||5        ||        |       |     |   |  |
|+---------+|        |       |     |   |  |
+----------+--------+------+-----+---+--+
```

So, the whole process is given by

```
,~ <@(>:@i.@# ;@([ ,.&.> (<.#){.])&.> ])
```

Starting with the initial empty partition of 0 and repeating this the required number of times, this finally leads to

```
(,~ <@(>:@i.@# ;@([ ,.&.> (<.#){.])&.> ])) ^:y <<i.1 0
```

If we distinguish the different processes we get

```
nrs0=: >:@i.@#                      NB. numbers to be generated
aps10=: [ ,.&.> (<.#){.]            NB. select partns, append nmbrs
new0=: ,~ <@( nrs0 ;@aps10 &.> ])   NB. new partitions to be added
part0=: 3 : ';;{. new0^:y <<i.1 0'  NB. the complete script
```

An alternative with single boxing

We change the foregoing construction by representing all partitions of number *n* in a matrix with *n* columns, where shorter partitions are padded with zeros, like in Fig. 7, the analogue of Fig. 6, with the order of the boxes and the order in the boxes reversed. At the left hand side the partitions of 0, than of 1, etc. until finally the partitions of 5 are given.

Figure 7.

```
++-+---+-----+-------+---------+
||1|2 0|3 0 0|4 0 0 0|5 0 0 0 0|
|| |1 1|2 1 0|3 1 0 0|4 1 0 0 0|
|| |   |1 1 1|2 2 0 0|3 2 0 0 0|
|| |   |     |2 1 1 0|3 1 1 0 0|
|| |   |     |1 1 1 1|2 2 1 0 0|
|| |   |     |       |2 1 1 1 0|
|| |   |     |       |1 1 1 1 1|
++-+---+-----+-------+---------+
```

Now the processes are adapted as follows.

```
nrs1=: (-i.)@#                        NB. numbers to be generated
apsl1=: [,. (>: {."1)# ]              NB. select partns, append nmbrs
new1=: , [: <@; nrs1 apsl1 &.> ]      NB. new partitions to be added
part1=: 3 : '> {: new1 ^:y <i.1 0'    NB. the complete script
```

Not all smaller partitions are needed

Let's have a closer look at this example of the partition of 5. If you look at Figures 0 and 2, it is obvious that not all smaller partitions are selected. From the partitions of 4 only the first one is used and from those of 3 only the first 2. So why not consider only those partitions which are needed for the final result?

From equation (A) we conclude that for generating all partitions of *n*, we need only the partitions *P(k)* and *P(n-k,k)* with $0 < k \leq ½n$ to combine with *n-k* and *k* respectively. As an example, we take *n=6*, for *P(6)* we need the partitions *P(5,1)*, *P(4,2)*, *P(3,3) = P(3)*, *P(2)*, *P(1)*.

First we determine max(*k,n-k*) for *k = 1…n-1*, which is done by `(](-<.>:@])i.)@<:` and append these numbers to the original number, see Fig. 8.

Figure 8.

```
   (],(](-<.>:@])i.)@<:) 6
6 1 2 3 2 1
```

Now these numbers, taken from right to left, are used to generate the partitions *P(6)*, *P(5,1)*, *P(4,2)*, *P(3,3) = P(3)*, *P(2)*, *P(1)*, also in the order from right to left. This is done as follows:

```
6 1 2 3 2 1 P(0)
6 1 2 3 2 P(1) P(0)
6 1 2 3 P(2) P(1) P(0)
6 1 2 P(3) P(2) P(1) P(0)
6 1 P(4,2) P(3) P(2) P(1) P(0)
6 P(5,1) P(4,2) P(3) P(2) P(1) P(0)
P(6) P(5,1) P(4,2) P(3) P(2) P(1) P(0)
```

It is obvious that *P(5,1)* and *P(4,2)* are quite a bit less numerous than *P(5)* and *P(4)*.

Since we want to apply *scan* instead of *power*, we have to box these numbers first and then append the empty partition (of 0): see Fig. 9.

Figure 9.

```
   (<<i.1 0),~<"0(],(](-<.>:@])i.)@<:) 6
+-+-+-+-+-+-+--+
|6|1|2|3|2|1|++|
| | | | | | ||||
| | | | | | |++|
+-+-+-+-+-+-+--+
```

Scanning these boxes from right to left generates *P(0)*, *P(1)*, *P(2)*, *P(3)* = *P(3,3)*, *P(4,2)*, *P(5,1)*, *P(6)* subsequently. To explain this process, suppose we have reached *P(3)*, as depicted in the last box of Fig. 10, together with the remaining parameters 6, 1, 2.

Figure 10.

```
+-+-+-+---------------+
|6|1|2|+-----+---+-++|
| | | ||1 1 1|1 1|1|||
| | | ||2 1 0|2 0| |||
| | | ||3 0 0|   | |||
| | | |+-----+---+-++|
+-+-+-+---------------+
```

The process applies to the last two boxes, being

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

Opening the boxes first and selecting the items from the right as indicated by the left: `({.)&.>` we get

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

From the left argument, the (smaller) integers are generated (1 2) and from each of the boxes in the right argument those partitions are selected by

```
   (>:@i.@[ (((>:{."1) # ])&.>) {.)&.>
```

giving

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

The numbers from the left side now are prepended with those on the right side and the appropriate unboxing and boxing is done. This gives

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

This output is prepended to the right hand side. In J this becomes

```
   (],~ >:@i.@[ <@;@:(([,.(>:{."1)#])&.>) {.)&.>
```

The corresponding output is in Fig. 11.

Figure 11.

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

If this is applied to all the elements of the initial noun – see Fig. 9 – then we get the script

```
(],~ >:@i.@[ <@;@:(([,.(>:{."1)#])&.>) {.)&.>/ …
                      (<<i.1 0),~<"0(],(](-<.>:@])i.)@<:) 6
```

and the output as in Fig. 12. The first box contains the desired partitions of 6.

Figure 12.

```
+-----------------------------------------+
|+-----------+---------+-------+-----+---+-++|
||1 1 1 1 1 1|1 1 1 1 1|1 1 1 1|1 1 1|1 1|1|||
||2 1 1 1 1 0|         |2 1 1 0|2 1 0|2 0| |||
||2 2 1 1 0 0|         |2 2 0 0|3 0 0|   | |||
||2 2 2 0 0 0|         |       |     |   | |||
||3 1 1 1 0 0|         |       |     |   | |||
||3 2 1 0 0 0|         |       |     |   | |||
||3 3 0 0 0 0|         |       |     |   | |||
||4 1 1 0 0 0|         |       |     |   | |||
||4 2 0 0 0 0|         |       |     |   | |||
||5 1 0 0 0 0|         |       |     |   | |||
||6 0 0 0 0 0|         |       |     |   | |||
|+-----------+---------+-------+-----+---+-++|
+-----------------------------------------+
```

If we split this script into comprehensible verbs we get

```
NB. initial sequence
init=: (<<i.1 0) ,~ <"0@(] , (] (- <. >:@]) i.)@<:)
NB. select partns, prepend numbers
ppsl=: >:@i.@[ <@;@:(([ ,. (>: {."1) # ])&.>) {.
exit=: >@{.@>                               NB. desired partition
part2=: [: exit [: (],~ ppsl)&.>/ init  NB. complete script
```

Performance

Performance is measured for *n* = 5 10 15 20 25 30 35 40 45 50. If we measure only the relative performances of `part0` and `part1` with respect to the third verb `part2`, we get the following figures for execution time respectively:

```
   5     10     15     20     25     30     35     40     45     50
1.49   2.19   2.86   3.28   3.08   2.52   2.18   2.17   1.94   1.97
1.32   1.84   1.80   2.28   2.48   2.98   3.11   3.56   3.24   3.40
```

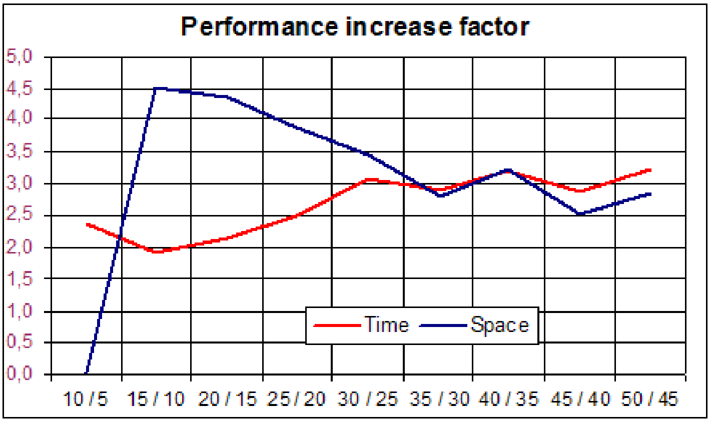
with averages 2.4 and 2.6 respectively.

For execution space the relative performances of `part0` and `part1` compared to `part2` are respectively:

```
   5     10     15     20     25     30     35     40     45     50
1.32   1.31   1.35   1.37   1.43   1.52   1.65   1.65   1.74   1.75
1.10   1.17   1.36   1.57   1.75   1.88   2.06   1.97   2.13   2.14
```

with averages 1.5 and 1.7 respectively.

The factor of increase of execution time and space of `part2` with each step of 5 extra, is shown in the following chart.



As can be seen, the partition of each 5 more elements costs about 3 times more.

Notes

1. "Partition" Eric W Weisstein, in *MathWorld*, a Wolfram Web Resource, http://mathworld.wolfram.com/Partition.html

2. *The Art of Computer Programming* , Donald Knuth, Vol. 4, Fascicle 3, p.37. Knuth uses non-negative integers for the addends, although "... the zero terms are usually suppressed."

3. http://www.jsoftware.com

4. http://www.jsoftware.com/pipermail/general/2005-June/023191.html

5. So all the numbers of any element of *P(n,k)* are less than or equal to *k*.

About polynomials

Part 1

Gianluigi Quario
giangiquario@yahoo.it

Abstract
This is the first part of an article dedicated to polynomials: some thoughts about polynomials, their evaluation, their functional treatment; how APL helps us to understand the strict ties between polynomial evaluation and division; and a new insight into polynomial division. The second part of the article addresses the problem of finding zeros.

A univariate polynomial p(Y) of degree N is defined to be a formal expression in the CMN (common mathematical notation) of the form:

$$p(Y) \Leftrightarrow c[N]\times Y^{N} + c[N-1]\times Y^{(N-1)} + \ldots + c[1]\times Y^{1} + c[0]$$

Here the expression is in descending order, Y is a formal symbol, and every power of Y is just a placeholder for a coefficient c[k].

Suppose that in the APL environment `⎕IO←0`, `c` is a numeric vector, `c[0]` is the constant term in the polynomial and `c[k]` is the coefficient of Y^k.

One can associate with every polynomial a polynomial function, whose values are obtained everywhere by replacing symbol Y by a numerical value.

The reason that mathematicians distinguish between polynomials and polynomial functions is subtle and not worth investigating here. We shall later consider the polynomials as vectors representing polynomial functions and ponder about their functional status.

Nonetheless we'll consider the problem of their evaluation.

Some hints were given by J's dictionary and by suggestions from Phil Last, Stephen Mansour and Stephen Taylor. The APL expressions are written in Dyalog.

Handy definitions

A polynomial of degree 0 is a *constant polynomial*.

When c[0]=0 and N=0 the polynomial is called a *zero polynomial* or *null polynomial*.

An *integer polynomial* is a polynomial where all elements of `c` are integer.

A polynomial, all of whose `c` elements are zeroes with exception of one element, is called a *power polynomial* or *monomial polynomial*.

For example: Y^3 or $-5×Y^5$ or $3×Y$

A polynomial is said to be *primitive* if the greatest common divisor of its coefficients is 1.

For any field F (Integers, Rationals, Reals, Complexes, etc.) the polynomials with coefficients in F form a ring which is denoted by F{}.

A polynomial p(Y) in F{} is called irreducible over F if it is non-constant and cannot be represented as the product of two or more non-constant polynomials from F{}.

This definition depends on the field F.

For example: p(Y) ⇔ Y^2+1 is irreducible over Real field R but not over Complex field C.

APL evaluation of a polynomial function

A polynomial function can be evaluated in several ways.

Let `poly` be a numerical vector of coefficients of polynomial p(Y); let `point` be a numerical scalar or array.

Then the value of polynomial p(Y) at `point` is

Ruffini-Horner (a)	`poly[0] + point × poly[1] + point × poly[2] + … point × poly[N]`
Sum of power monomials (b)	`(point∘.*⍳⍴poly)+.×poly`
Base value (c)	internal Ruf-Horn method valid also when poly is a null polynomial `(point∘.+,0)⊥⌽poly`
Ruffini-Horner (d)	`⊃{⍺+point×⍵}/poly`
Ruffini-Horner (e)	`⊃+∘(point∘×)/poly`
Ruffini-Horner (f)	`⊃point{⍺+⍺⍺×⍵}/poly`

Chain of forks (g)

```
PolyEval←(poly[0]∘+)∘
              … (×∘+∘(poly[1]∘+)∘
              … (×∘+∘(poly[2]∘+)∘
              … (×∘+∘(poly[3]∘+)∘
              … (poly[4]∘×)⍨)⍨)⍨)
PolyEval point
```

Form (a) shows the superiority of Iverson notation to the Common Mathematical Notation (CMN):

poly[0] + point × (poly[1] + point × (poly[2] + … point × poly[N]) …))

The Iverson notation is neat, does not use any parentheses, and shows a strict symmetry between addition and multiplication. If we could couple the primitive *reduce* operator with an array of functions `+ ×` we could write:

```
(+ ×)/poly[0],point,poly[1],point,poly[2], … ,point,poly[N]
```

The J language does not provide for arrays of functions, but includes the wonderful idea of gerunds.

The polynomial can be evaluated in J by means of conjunction *tie* in this way:

```
+'*/poly[0],point,poly[1],point,poly[2], … ,point,poly[N]
```

Form (f) stresses the prominence and simplicity of Horner's algorithm and shows a greater semantic clarity than the form (c), which internally uses – but also hides – this algorithm.

Form (g) is an example of another way to define the evaluation function of `poly`.

It is reported here because it is a *direct assignment* function, to which the inverse operator `⍣¯1` can be applied (it could be useful for finding a root of `poly`).

Functional background of polynomials

The set of all polynomials of degree ≤N, over a field (or a ring) F, is denoted F{N}.

With the usual algebraic operations, F{N} is a vector space, because it is closed under addition (the sum of any two polynomials of degree ≤N is again a polynomial of degree ≤N) and scalar multiplication (a scalar times a polynomial of degree ≤N is still a polynomial of degree ≤N).

There is a simple isomorphism between F{N} and the vector space $F^{(N+1)}$.

The standard basis for F{N} is the base {1, Y, Y^2, … Y^N}.

This basis is also called the *monomial basis* and comes from the standard basis for $F^{(N+1)}$.

If F is the field of Integers/Rationals/Reals, the APL vector `poly` (`⍴poly` ←→ N+1) of coefficients of any polynomial can represent a unique element of the vector space F{N} .

Hence we can think of a numerical vector as a generic polynomial: furthermore it is straightforward to imagine an isomorphism between functional programming (over polynomials) and traditional data programming (over APL numerical vectors) … *sometimes Functional Programming can be carried out with no need to be mixed up with functions*!

The vector space of all polynomials with coefficients in F forms a commutative ring denoted F{} and is called the ring of polynomials over F. The symbol Y is commonly called the *variable*, and this ring is also called the ring of polynomials in one variable over F, to distinguish it from more general rings of polynomials in several variables. This terminology is suggested by the important cases of polynomials with real or complex coefficients, which may be alternatively viewed as real or complex polynomial functions.

However, in general, Y, and its powers Y^k, are treated as formal symbols, not as elements of the field F. One can think of the ring F{} as arising from F by adding one new element Y that is external to F and requiring that Y commute with all elements of F. In order for F{} to form a ring, all powers of Y have to be included as well, and this leads to the definition of polynomials as linear combinations of the powers of Y with coefficients in F.

If F is the field of Integers/Rationals/Reals, the set of all APL numerical vectors (any length) can represent the set of the ring F{}. Note that a null polynomial can be represented by a zero-length vector or by the vector `,0`.

Functional management of polynomials

Sum and product

A ring has two binary operations, addition and multiplication.

In the case of the polynomial ring F{}, let `poly1` and `poly2` be the vectors representing two polynomials, then these operations are explicitly given by the following definitions:

```
        polySUM  ← {deg←⊃⌈/ρ∘,¨α ω ◊ (deg↑α)+deg↑ω }
        polyPROD ← {+⌿(⎕IO-ιρ,α)⌽α∘.×ω,1↓0×α}
```

where the arguments and results of `polySUM` and `polyPROD` are all vectors.

The identity element for addition is the null polynomial.

The identity element for multiplication is the polynomial `,1`.

Let us use the following function for polynomial evaluation:

```
        PolyEval ← {⊃ω{α+αα×ω}/α}
```

Then the functional highlight is that – for any point `point` – the following pairs of expressions are equivalents:

```
        (poly1 PolyEval point)+(poly2 PolyEval point)
        polySUM PolyEval point

        poly1 PolyEval poly2 PolyEval point
        polyPROD PolyEval point
```

Note that this definition of the multiplication of two polynomials is assimilable to the ∘ function-composition operator or – in maths speech – the 'Discrete Convolution' of the polynomial functions. In fact the following two are also equivalent:

```
        (poly1∘PolyEval)∘(poly2∘PolyEval)
        polyPROD∘PolyEval
```

Note that the product of two primitive polynomials is also primitive.

Over the Complex field, every non-constant polynomial can be unambiguously factored into linear factors. In the CMN:

p(Y) ⇔ poly[N] × (Y-z1) × (Y-z2) × … × (Y-zN)

or (in APL), p(Y) is given by:

```
        PolyMult ← {+⌿(⎕IO-ιρ,α)⌽α∘.×ω,1↓0×α}
        poly[N] × ⊃PolyMult/-(z1,z2 … zN),¨¯1
```

where `poly[N]` is the leading coefficient of the polynomial and the `z`s are the zeroes of p(Y).

Hence, over the Complex field, all irreducible polynomials are of degree 1: this is the fundamental theorem of algebra.

So, over the Complex field, the polynomial represented by vector `poly` can also be represented by `poly[N]` jointly with vector `z1, z2 … zN`. The J language has chosen to represent any polynomial either with `poly` or the nested vector:

```
(poly[N])(z1,z2 … zn)
```

Division

When the ring F{} has the *zero-product* property (this is true for Integers, Rationals, Reals, Complexes), it is possible to define a division between the polynomials of the ring F{}.

If f and g are polynomials and g is not the null polynomial, then there exist unique polynomials q and r such that:

$$f = qg + r$$

where the degree of r is smaller than the degree of g.

This division is called division with remainder. The polynomial f is said to be divisible by g when remainder r is the null polynomial.

At school we learned Ruffini's 'simple-division' rule to obtain the quotient polynomial q and the remainder polynomial r.

The following function performs this algorithm; it is a transcription of a C-language procedure.

```
      ∇
 [0]   Z←u PolyDivi_0 v;n;nv;q;r;k;j;⎕IO    …
                          ⍝ division of 2 general polynomials
 [1]  ⍝ cfr Numerical Recipes in C :the art of scientific computing
 [2]  ⍝ u v  ⍝numerator and denominator polynomials
 [3]  ⍝ q r  ⍝quotient and remainder
 [4]  ⍝ void poldiv(float u[], int n, float v[], int nv, …
                                          float q[], float r[]
 [5]  ⍝ the coefficients of quotient poly are returned in …
                                          q[0..n] and the coeff
 [6]  ⍝ of remainder poly are returned in r[0..n]
 [7]  ⍝ the elements r[nv..n] and q[n-nv+1..n] are returned as zero
 [8]   ⎕IO←0
 [9]   n←¯1+⍴u  ⍝degree of numerator
 [10]  nv←¯1+⍴v ⍝degree of denominator
 [11]  r←u
 [12]  q←(⍴u)⍴0
 [13]  :For k :In ⌽⍳1+n-nv
 [14]     q[k]←r[nv+k]÷v[nv]
 [15]     :For j :In ⌽k+⍳nv
 [16]        r[j]-←q[k]×v[j-k]
 [17]     :EndFor
 [18]  :EndFor
 [19]  r[nv+⍳1+n-nv]←0
 [20]  Z←q r
      ∇
```

For example

```
      (q r) ← 5 4 6 4 1 PolyDivi_0  1 3 1
      q
 2 1 1 0 0
      r
 3 ¯3 0 0 0
```

Given a divisor polynomial *g*, every polynomial in F{} is associated to two polynomials *q* and *r*, and

(degree *f*) ≥ (degree *q*) + (degree *r*)

But we can define a slightly different polynomial division where, given a divisor polynomial, every polynomial in F{} is associated with a single polynomial in F{} of the same degree.

For that purpose it is necessary to look at Ruffini's rule and adapt its algorithm by means of Horner's algorithm coupled with the 'Accumulating reduction' D-operator

```
        acc ← {⊃αα{(⊂α αα⊃θρω),ω}/1↓{ω,⊂θρω}¯1⌽ω}
```

Let us start with the simple case where the divisor *g* is a monomial (1-degree) polynomial.

Then consider the result `s` of:

```
        s ← (-⊃g) {α+αα×ω} acc f
```

We obtain a vector `s` of the same length as `f`, composed by the catenation or the remainder polynomial `r` and the quotient polynomial `q`.

The first element of `s` is also the value of polynomial `f` evaluated at point `⊃g`.

For example

```
        1 {α+αα×ω} acc 5 4 6 4 1
  20 15 11 5 1
```

The remainder of dividing `5 4 6 4 1` by `¯1 1` is `20` and the quotient is `15 11 5 1`.

By means of the simple expression `{α+αα×ω}` and operator `acc`, it is possible to implement Ruffini's 'simple division' rule between a polynomial and a monomial: we have neither iterations nor recursions.

Funny! The polynomial division is achieved through sum and product.

Note that the APL notation emphasises that Ruffini's rule is just a more general form of the polynomial evaluation function:

```
        (-⊃g) {α+αα×ω} / f
```

In the case where the degree of divisor `g` is greater than one, the polynomial division is usually executed by means of the 'long division' algorithm. Yet it is enough to extend the 'simple division' rule above; now we can perform it for any divisor:

```
PolyDivi←{
  ⍝ division of polynomial ⍺ by  polynomial ⍵
  ⍝ higher degree synthetic division algorithm
  ⍝ return:  vector (remainder,quotient) polynomial
  ⍝ purge highest_degree_terms when coefficient is 0
    ZeroClean←{(-+/∧\⌽⍵=0)↓⍵}
    num den←ZeroClean¨⍺ ⍵
    0∊⍴num:num num  ⍝ NULLpoly ÷ poly or NULLpoly ←→ NULLpoly
    0∊⍴den:'DENOMINATOR ERROR'⎕SIGNAL 11
    1=⍴den:num÷den
    (⍴num)<⍴den:num
    acc←{                         ⍝ accumulating reduction
        ⊃⍺⍺{(⊂⍺ ⍺⍺⊃⍬⍴⍵),⍵}/1↓{⍵,⊂⍬⍴⍵}¯1⌽⍵}
    extRuffini←{  ⍝ extended Ruffini-Horner cfr {⍺+⍺⍺×⍵}
        ((⍴⍺⍺)↑(⊃⍺),⍵)+⍺⍺×¯1↑⍵}
    synt←(-¯1↓den÷¯1↑den)extRuffini acc (⍴1↓den),/num
    (⊃synt),⊃¨⌽¨1↓synt
}
```

For example

```
      5 4 6 4 1 PolyDivi 1 3 1
3 ¯3 2 1 1
```

Into the bargain, polynomial division with a fixed polynomial divisor can be considered as something more than an operation that returns a quotient and a remainder: it can be seen as a one-to-one correspondence between F{} and F{}.

Given the polynomial *f* with degree N we obtain a polynomial *s* with degree N, thus there is also a one-to-one correspondence between F{N} and F{N}.

Derivative and Integral

Inside any ring F{} or F{N} it is possible to define another correspondence, the *derivative*: this is a many-to-one correspondence.

The following function returns the derivative `s` of any polynomial `f`:

```
      PolyDerivative ← {(1<⍴⍵)↓⍵ ×-⎕IO-+⍳⍴⍵}
```

The degree of `s` < degree `f` (except when `f` is a constant or null polynomial).

For example:

```
      PolyDerivative  5 4 6 4 1
      4 12 12 4
```

We can also have a one-to-many correspondence inside F{} : the *integral*.

Since a single polynomial can have an infinity of integral polynomials, it is not possible to define a suitable APL function. But we may anchor any number of the field F and build a function that defines a one-to-one correspondence:

```
      PolyIntegral←{⎕IO←1 ⋄ α,ω÷⍳⍴ω}
```

The degree of `s` > degree `f` (except when `f` is a null polynomial).

For example:

```
      33∘PolyIntegral 5 4 6 4 1
 33 5 2 2 1 0.2
```

Perspective

Polynomial evaluation is the first step to the old problem of root finding. The second part of this paper will address the task of defining a stable and wide range function for the extraction of complex roots.

Further reading

1. http://en.wikibooks.org/wiki/Abstract_algebra
2. http://mathworld.wolfram.com/topics/Algebra.html
3. http://www.jsoftware.com/jwiki
4. http://www.dyalog.dk/dfnsdws/n_contents.htm

Subscribing to Vector

Your *Vector* subscription includes membership of the British APL Association, which is open to anyone interested in APL or related languages. The membership year runs from 1 May to 30 April.

Name:	____________________
Address:	____________________
	____________________
Postcode/Zip and country:	____________________
Telephone number:	____________________
Email address:	____________________

UK private membership	£20	__
Overseas private membership	£22	__
+ airmail supplement outside Europe	£4	__
UK corporate membership	£100	__
Overseas corporate membership	£110	__
Sustaining membership	£500	__
Non-voting UK member (student/OAP/unemployed)	£10	__

Payment should be enclosed with your membership form as a sterling cheque payable to "British APL Association" or by quoting your Visa or Mastercard number.

I authorise you to debit my Visa/Mastercard account

Number: ______________________ Expiry date: ____ / ____

for the membership category indicated above,

- __ annually, at the prevailing rate, until further notice
- __ for one year's subscription only

Signature: ______________________

Send this completed form to:

BAA, c/o Nicholas Small, 12 Cambridge Road, Waterbeach, Cambridge CB25 9NJ