

Contents

Editorial		3
News		
Dyalog		5
Kx		10
Optima		11
General		
BAA: Chairman's Report	Paul Grosvenor	14
BAA: AGM Minutes	Chris Hogan	16
Report on BAA London	Phil Last	18
Phil Last reports from his moot	Phil Last	20
APL		
Three blind mice	Paul Grosvenor	24
Our first steps into the APL world	Greeley, Gutsell, Sidiki	26
User Commands in Dyalog	Dan Baronet	30
Dyalog's public User Commands	Dan Baronet	41
Regular Expressions in Dyalog (⎕R and ⎕S)	Dan Baronet	61
Bayesian modelling in APL	Devon McCormick	72
Hui juggles with 88 hats	Roger Hui	84
Function design	Kai Jaeger	90
J		
Backgammon tools in J:		
3. Two-sided bearoff probabilities	Howard Peelle	104
Savitzky-Golay interpolation for smoothing values and derivatives	Porter & Reiter	107

Quick reference diary

20-24 October 2013

Florida

Dyalog User Conference

Dates for future issues

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

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

EDITORIAL

August 1985, I was handed my first copy of *Vector* Vol.1 No.1. I had just joined BUPA, an escapee from a COBOL environment, where an analyst at a keyboard was deemed time wasting. An incidental exposure to VSAPL led to lunch times learning the language, and a subsequent career switch.

Coming from that relative isolation into a proper APL shop the wide and active community of APLers soon became apparent. Since then I have not been disappointed. APL has given me some great friends; work experiences; intellectual stimulation; and pleasure. The fun just goes on with other array-processing languages such as J and K offering another rich vein to explore.

Stephen Taylor will be a tough act to follow. He has set the bar very high but has also given *Vector* the means and methods to continue producing an outstanding journal. Stephen's efforts have been immense, taking on the role of an entire editorial committee. As editor he went beyond writing editorials, actively trying to obtain material to support series or collect related articles; as webmaster he built the website and archive to support online publication; as typesetter he also rebuilt the production process to get *Vector* in print.

Meet the three young apprentice APL Programmers in their article 'Our first steps into the world of APL'. Funded by Optima Systems (UK) and Dyalog Limited in an initiative to introduce new people to the APL community. The apprentices were given a warm welcome in October by delegates at the Dyalog conference in Elsinore, Denmark. Here, as part of a presentation 'Three Blind Mice' by Paul Grosvenor, delegates heard the apprentices describe their first experiences and impressions of APL. I am reminded of my own early journey.

In the archive is a wealth of material and amongst that material are some gems that are worthy of being reprinted, hopefully stimulating further interest and work. In this edition I have begun with a reprint of 'Bayesian financial dynamic linear modelling in APL' in *Vector* 21.2 by Devon McCormick, and hope that Devon will follow this up with implementation in J. I will welcome suggestions for other candidate reprints.

This year sees the fourth anniversary of the BAA London symposia, these well fit either definition of a meeting in which the participants form an audience and make presentations" or the alternative "convivial meeting for drinking and intellectual discussion". From a personal point of view I have never attended one where I haven't learned something useful. Phil Last gives a potted history in his article "BAA London" in this issue.

John Jacob

NEWS

Industry news

Dyalog Ltd

Looking East and West

The Annual Dyalog Programming Contest is in its 4th year. In the first three years, we have had entries from all over the world, with winners and runners-up coming from New Zealand, the USA and Germany. This year, the best submissions were from the East (as seen from the United Kingdom): we are very pleased to announce that the winner of the Grand Prize in 2012 (USD 2,500 and an expenses-paid trip to the Dyalog' 12 Conference in Denmark) is 橋本隼人 (Hayato Hashimoto), a 21 year old undergraduate student at Kyoto University in Japan – and the runner-up Татьяна Иванова (Tatiana Ivanova) from Russia.

At the same time, we are delighted to announce that Pat Buteux has joined Dyalog Ltd as VP of Sales & Marketing for the USA. Pat joins Dyalog Ltd with one of the finest pedigrees in software development and software consulting, and she is particularly experienced in working with business development, strategic branding and positioning. Pat's career started at General Electric in the timesharing days. She helped transform STSC into an independent software vendor and was responsible for development, marketing and sales of APL*PLUS. She also built STSC's international network of resellers. We're incredibly pleased to have Pat join the Dyalog team and we're looking forward to working with her on expanding the APL market to our West.

Dyalog APL Versions 13.1 and 13.2

We are currently about halfway between Version 13.1, which was released in April, and 13.2, which is due to be released at the end of January 2013. As the decimals suggest, these are incremental updates, with particular focus on performance or capacity enhancements, and (ahem) the odd bug fix – while we work on version 14.0 (targeting end of 2013) which is intended to add very significant new core-language features and more fundamental performance improvements and features to support parallel hardware.

That being said, versions 13.1 and 13.2 will not be without new functionality:

Summary of Version 13.1 enhancements

- New system function `□FHIST` – The new File History system function reports information about which user created and most recently updated the file (and when it happened). This feature has been added to support clients who have previously been running the SHAREFILE/AP system on mainframes.
- New system constant `□DMX` – Version 13.1 adds significant new error reporting functionality, in the form of a system variable which has been named `□DMX`. `□DMX` is designed to make error handling simpler, more complete and more accurate.

The key benefits of `□DMX` are: the provision of more detailed information, including the relevant operating-system error where appropriate; that it is thread safe (`□DMX` is local to each APL thread – which is not the case for `□DM` and `□EN`) and that it is local to the code which is invoked to handle an error, in such a way that successfully handled (trapped) errors will leave no trace of the handled event.

- Performance Improvements: including much faster processing of lines which contain no code, speedups to most cases of dyadic iota (Index Of), a couple of new idioms `0=⌵p` and `0≠⌵p`, and finally significant speed-ups to the performance of expressions of the form `(n↑1)`.
- URL string support: URLs in the session or tracer/editor are underlined. Control- and left-click on the URL string will cause the URL string to appear in the default browser (or the default e-mail client for mailto: URLs).
- Enlist in selective assignment: Enlist (`ε`) may be used in selective assignment expressions.
- Large arrays: version 13.1 marks the final step in adding support for unlimited array sizes. All primitives and the component file system now support arrays of any size.

Release Notes for version 13.1 (and all other releases) can be viewed online at <http://help.dyalog.com>, and can be downloaded in PDF format from <http://docs.dyalog.com>.

Planned v13.2 Features

We would like to thank the participants of the 2012 BAPLA AGM & Moot, which was held at the YHA Lee Valley Youth Hostel, Cheshunt, in April (<http://moot.aplwiki.com/>). In 2006, version 11.0 of Dyalog APL introduced support for object-oriented programming in APL, and this included support for editing both classes and namespaces in the form of a single script. As the use of scripted APL code has slowly spread in the APL community, it has become apparent that the editor needs further enhancements (and some bug fixes). The participants of the moot were invited to discuss these issues, and we would like to acknowledge this valuable input, which will be a driver for work going into both 13.2 and future releases. Other candidates for new functionality to be delivered as part of 13.2 include:

- A number of new GUI features to provide support for gradient colouring, transparent effects, and other modern GUI features (in order to allow clients a bit more time to see which way the Microsoft.NET cookie is going to crumble).
- Speedups to scans, summations, and the regular expression operators `⎕R/⎕S`.
- Support for all the selective assignment syntax allowed by IBM APL2.
- Callbacks into APL from `⎕NA` calls.

The planned release date for Version 13.2 is January 31st, 2013

The Dyalog File Server

Due to be released shortly after version 13.2, the Dyalog File Server (DFS), will provide secure, multi-user access to APL component files and native files – similar to the functionality provided by products like SHAREFILE/AP which was used with mainframe APL systems. The TCP-based client/server architecture employed by DFS avoids the need for direct network access to the underlying component files, doing away with the need for network shares.

DFS provides significantly enhanced security, compared to the use of component files in the form of shared files on a LAN. DFS supports username/password style authentication as well as Integrated Windows Authentication (IWA) on Windows platforms. Secure communications can be used between the client and server using Conga's Transport Layer Security (TLS) support. As a fringe benefit of this project, recent versions of Conga allow you to provide single-sign-on

functionality in your own application, if your organization uses Windows domains.

In many typical scenarios the DFS also performs better than the existing component file system. Multiple file services can be configured to help balance the usage loads, and a web-based console will provide logging, usage/performance monitoring, and administrative functions – including the ability to perform full and incremental backups and restore operations without taking the system down.

The DFS will be sold separately from Dyalog APL, and targets enterprise systems. It will initially only be available for Microsoft Windows, but is designed to be portable.

TryAPL.ORG

As announced in our previous news item, we released an on-line interactive APL workbench at <http://tryapl.org>, early this year. We are not quite seeing Google-sized numbers but the site has seen an average of 10-20 distinct user sessions per day, and at times the numbers have been spectacular, as they were in the days following the discovery of TryAPL by “reddit”.

We had about 8,000 distinct sessions over a weekend following http://www.reddit.com/r/programming/comments/uu2br/try_apl_is_weird_but_fun/. The server crashed a few times due to a WS FULL in our log file handling code, but increasing MAXWS quickly resolved that! In June, the site was used to teach an afternoon’s APL workshop for a group of students running a variety of operating systems including Windows, Linux and Apples OSX, without requiring the installation of any software!

APL Training Courses

Speaking of workshops: as was the case the last time we submitted a news report to *Vector*, Bernard Legrand has just completed another introductory APL course at our head office in Bramley. We are very pleased to be able to report a steady increase in demand for APL training, and expect to be running a minimum of two courses per year in Bramley, in addition to the in-house courses that Bernard regularly teaches at client sites. If you have an APL-related training requirement anywhere in the world, please contact Karen Shaw in Europe or Pat Buteaux in North America or reach all the relevant people at once with an e-mail to sales@dyalog.com.

Karen and Pat's email addresses are <firstname>@dyalog.com as indeed is almost everyone else at Dyalog too.

More Details

In-depth articles on many of the issues mentioned above are available in the last two issues of the Dyalog newsletter, which can be viewed at <http://dyalog.com/news.htm> – look for the “newsletter” links in the left column. The best place to catch up on all things Dyalog is of course to attend our annual User Conference. The next conference will be held at the Embassy Suites Hotel on Deerfield Beach, between Boca Raton and Ft. Lauderdale on Florida's Gold Coast, October 20-24, 2013. Recordings of talks from the recent conference in Denmark will be posted to <http://videos.dyalog.com> and on YouTube (search for 'Dyalog Conference').

Kx releases new version of kdb+

Palo Alto (24 Sep 2012) – Kx Systems, the leader in high-performance database and timeseries analysis, has announced the release of kdb+ v3.0. The key benefits of the new version include a considerable improvement in processing speeds when running on Intel's recent processors, support for WebSockets, GUIDs/UUIDs (unique identifiers, which facilitate the design of distributed systems) and simplified storage of billions of records.

The optimized code in kdb+ utilizes the processor specific instructions available at run-time: testing saw very significant speed increases when running calculations using Intel's Advanced Vector Extensions (AVX) and SSE instructions, available on Intel's latest generation of Sandy Bridge family of processors.

The growing volumes of derivatives and trading volumes in FX and equity markets, as well as regulatory requirements, all result in institutions having to store and analyse vast quantities of data. The simplified storage in v3.0 makes the design and implementation of large systems much less complex. While kdb+ has always been able to handle far more than 2 billion records, this has been made much simpler in the new release.

Kx chief strategist, Simon Garland, explains: "This enhancement in v3.0 simplifies the design and implementation of large systems which have to handle more than a trillion records, allowing for a more elegant architecture."

The addition of UUIDs as a basic data type means that distributed systems are now easier to write. UUIDs can be used to uniquely identify distinct records and are a valuable tool for managing distributed systems. In highly complex systems, which are spread across different regions and continents, UUIDs make distributed processing more efficient and system design more straightforward. At the same time, storing and processing transaction IDs, such as order and confirmation IDs, is easier and more efficient.

Garland continues: "Managing multiple servers across different countries and continents can be a challenge and requires some complex programming. UUIDs make this much more straightforward, as individual records can be uniquely identified; combined with the speed enhancements in v3.0 and ease of handling hundreds of billions of records, more efficient systems can be designed. This is an important step forward, especially in the face of ever-growing data volumes."

Another new feature in kdb+ v3.0 is the introduction of support for WebSockets, which allow for a direct, bi-directional, full-duplex connection between a browser and an application. This is a much more efficient approach than HTTP/AJAX, offering greater scalability and much faster processing. It is particularly useful for high-performance browser-based applications, for example applications visualizing real-time data.

Daryan Dehghanpisheh, global director, Financial Services Team at Intel, says: "The increased complexity of the markets and continued race towards automation, across more asset classes and venues, mean that the enormous growth of data will continue. The new version of kdb+ running on the latest Intel processors, such as the Intel Xeon Processor™ server platforms - which are optimized to support AVX instructions, to further increase overall performance - provides market participants and technologists with new capabilities, levels of performance and flexibility. The result is a powerful tool in the hunt for Alpha, while ensuring maximum stability and reliability."

Optima Systems Ltd

Firstly, I am pleased to be able to welcome John (Jake) Jacob to Optima as a full-time member of staff. Many of you will already know John for his work with *Vector* and within BAA London. We have a few more challenges for him now and we hope to keep him very busy.

Secondly, as you may already heard, we have in collaboration with Dyalog taken on three trainee APL programmers; James Greeley, Samuel Gutsell and Shaquil Sidiki. They started with us in August and have attended their first APL course and APL conference already. Take a look at their blog[1] and see how they are getting on.

We have had a fantastic year and have never been busier. Our COSMOS product which some of you have seen me demo looks like it has found its first customer and we are hopeful of a number of others in the UK and US. Development proceeds apace and is keeping us focused shall we say.

Interest in our products and proposals is currently at an all-time high and we have now just started to work with UK Trade & Investment and market ourselves properly outside of the UK. There will be more news on this in the next edition of *Vector*.

Other work, coming in from new clients, is becoming increasingly web based and we are now actively looking at the new technologies to see how they might help us. APL remains very much the cornerstone of our work but now we find ourselves interfacing to an ever increasing toolset.

These days it's all about finding enough hours in the day. With this in mind I am also very pleased to be able to announce that Peter Merritt is now our APL Team Leader and Gilgamesh Athoraya our R&D and Technical Analyst.

References

1. <http://threeblindmiceapl.wordpress.com>

GENERAL

BAA: Chairman's Report

October 2012

Paul Grosvenor (paul@optima-systems.co.uk)



Well here we go again. Another *Vector* falls off the production line but this time we have recruited a number of new people to assist with the process and now hopefully we can get more regular issues out to you all.

We are all aware of the worldwide recession we find ourselves in but even so the activity within the APL community remains very high. It has been a few years since I can recall quite so many good things going on.

At our AGM this year in the Lee Valley we made a few decisions regarding the way in which *Vector* was to be produced and managed;

- We would not be bound to 4 issues per membership year. Instead we would produce editions as and when we could but aim for no fewer than three.
- With the Chairman, Secretary, Editor and many of the *Vector* production crew all coming from Optima Systems staff I asked the attendees if there was any objection to Optima taking on such a large role in the production of *Vector*. There were no objections. However it is only right that I reaffirm that Optima will keep *Vector* running as an independent publication and not as a mouthpiece for its own purposes.

I have been very fortunate in the past year to be able to attend a number of the APL meetings and conferences around the world;

In April APL2000 ran their annual conference in New York. As always we were met by the friendly face of Sonia and her crew. The presentations were interesting and lively, food was good and APL speak went on long into the night – no surprises there then.

In September Stephen Taylor ran his Iverson College workshop in Cambridge. A fantastic time was had by all with twenty-four keen APLer's chewing the fat, writing code, putting the world to rights, punting, cycling and oh yes, talking long into the night [1].

October saw Dyalog holding their conference in Elsinore Denmark. The group met, consumed vast quantities of fine food, listened to fascinating presentations even danced the Samba! then talked long into the night.

And every month of course BAA London meets at the Albion pub to hear what is going and give their opinions – yes, long into the night.

I feel sure you are picking up a consistent theme here and I am sure I'm getting too old for it !

Nevertheless it is so nice to have such a friendly and accommodating community; something that is very rare these days. So let me finish with a comment from Devon McCormick after his trip to Lee Valley in April;

"The day I got back to NYC, I was in a bar talking to someone who was very knowledgeable about beer. When I told him I was just back from England where I'd attended a gathering of APLers, he told me that his wife used to program in APL. I joked that I probably knew who she was. Of course, as it turned out, I did"

References

1. sites.google.com/site/iversoncollege/home

Minutes of the British APL Association Annual General Meeting

by Chris Hogan (chris.hogan@4extra.com)

The 2012 AGM was held as the opening session of the APL Moot held at the Lee Valley Youth Hostel (Windmill Lane, Cheshunt, Hertfordshire, EN8 9AJ) which is situated within the River Lee Country Park.

The Minutes of the 2011 AGM were accepted by general consensus of those present.

Report from the Chairman:

Vector Production Team: Stephen Taylor is willing to continue in some form as long as he isn't full time editor. Kai Jaeger is now in charge of production of the printed Vector and John Jacob is webmaster. Assistance is given by Phil Last, Beau Webber and to a lesser extent by Chris Hogan and others.

Dyalog and Optima are willing to contribute to ensure the continued production of Vector, as is (probably) Kx systems according to Stephen Taylor.

There is a need for sub-editors for the different flavours, APL2000, Dyalog, MicroAPL, K, J etc. to ensure the quality of submitted material and resultant articles.

The trend is an increase in articles on KDE, Q and Dyalog and a decrease in J.

The Annual Award was not given at the AGM. The meeting voted to extend the recipients of the award to any of the array languages, e.g. J, K, Q, etc. It was also decided to poll the membership for nominations for next year's award.

Issues of Vector published each year: We normally expect 4 issues per volume of Vector. In the past there has been precedent for double issues, which meant fewer physical issues. The membership has been tied to this number so subscriptions have been collected every 4 printings rather than annually. It was decided that the physical printing of Vector was important, rather than simply having the articles published to the website, but that we should print 3 issues per year to ensure the association's ability to meet this commitment from its current income.

Monies owed by the BCS to the BAA: The BCS suffered a vote of no confidence from other special interest groups suffering in the same way as the BAA. The BCS committee spent further funds to fight this vote, which to those present at the meeting didn't seem within their powers. Their attitude is that any monies given to the BAA was general BCS funds. This will cost too much to fight in court. Chris Hogan to write one last time on behalf of HMW Computing as a sustaining member. The matter will be kept open, but this is a drain on the committee's time given the result of other similar actions taken against the BCS.

Financial Situation: A brief statement was made by the Treasurer/Membership Secretary (before the official end of year so full audited accounts are not available). Three sustaining members have yet to pay their dues, but assuming a schedule of three print runs of Vector per annum the treasurer can see no reason why the association cannot continue in its current state. No details of membership were available at present.

Reappointment of Committee: As everyone in a position was willing to continue and no one put forward any other names Paul suggested the the same committee continue, proposed by Jane and seconded by Ray.

Appointment of Auditor: The current auditor (Chris Hogan) was proposed by Paul Grosvenor and seconded by Ray Canon. Accepted by those present.

AOB: One question was raised about sustaining membership and it was clarified that a SM can nominate up to 5 individuals to represent them at meetings.

The meeting closed. It was followed by two presentations:

- Jeremy Sutton - Park Ranger: The history and background of the park.
- Kai Jaeger - Independent Consultant: FiRe: a FInd and REplace utility for Dyalog APL

BAA London

by Phil Last

On 8 November 2008 a post under the banner 'A message from three APL enthusiasts' was sent to `comp.lang.apl` and the Dyalog and MicroAPL forums from Chris Hogan, John Jacob and me inviting those within reach of the City of London to come to an inaugural meeting of what was billed as an informal meeting of APLers and was to become BAA-London.0

The meeting took place in the upstairs gallery at the Edgar Wallace pub in Essex Street on the 21st and ten people turned up, slightly more than what has become the average but not exceptional.

Three days later Stephen Taylor created our own on-line forum `groups.google.com/group/baa-london` and posted a suggestion that our next meeting, by invitation of Mike Hughes, could coincide both temporally and spatially with the IPSA Christmas reunion at the Plumbers Arms at Victoria. Thanks are due to the ex-Sharpies for putting up with us then and each Christmas since.

In January we were back at the "Edgar" where eleven of us each gave our ideas of what the meetings, much later to be renamed 'symposiums' by suggestion of Jane Sullivan, ought to be.

Chris (Ziggy) Paul gave us our first formal presentation in February entitled 'Education in APL'.

March gave us another web-presence when Ellis Morgan added some pages to the APL wiki. His monthly notes, `aplwiki.com/CategoryBAALondonMeeting`, a very useful but unsung contribution, continued until late 2010.

Presentations have been given and discussions led by a large number of members and guests including: Dan Baronet, Brian Becker, Dick Bowman, Nicolas Delcros, Walter Fil, Chris Hogan, Mike Hughes, Roger Hui, Morten Kromberg, Ellis Morgan, Chris Paul and Stephen Taylor. My apologies if I've inadvertently missed you off the list.

Given that any properly instituted organisation has been lacking, indeed resisted, what our meetings have turned out to be is precisely what those attending have brought with them. In Stephen's words "We are the agenda".

A number of projects have been started by suggestions made at the meetings. The Phrasebook pages of the APL wiki aplwiki.com/PhraseBook. The APL2010 Berlin open forums aplin2020.org.

A number of original ideas have been aired at the meetings but have not necessarily been given the exposure they deserve, 26 June 2009 Stephen Taylor with his suggestion for Direct Development not least among them.

After a few months at the "Edgar" we moved to a slightly quieter upstairs room in "The Knights Templar" in Chancery Lane. This was small but satisfactory until the month that I forgot to arrange it with the landlord (and failed to turn up) and those attending found themselves jostled among a large and noisy crowd watching an international soccer match in the main bar.

We held a couple of meetings in the "Punch Tavern" in Fleet Street that had a very convenient room but no WiFi and very poor reception for our own mobile broadband.

In April 2010 we moved to "The Albion" in New Bridge Street where we have a large, quiet, private room. For our second meeting there we hosted the British APL Association Annual General Meeting. We've been there ever since and about four months ago we finally worked out how to plug our computers into the large TV monitor on the wall so Chris Hogan no longer has to cart his 'luggable' projector to the meetings.

Three of our meetings have been elsewhere than in City of London pubs. The first at a domestic venue in Cheshunt, Hertfordshire in August 2009; the second in July 2010 when we were generously hosted by Dyalog Limited in their offices in Bramley in Hampshire; and most lately the 2012 BAPLA AGM and Moot at the Lea Valley Youth Hostel in Cheshunt, Hertfordshire in April/May.

It will be four years all but a few weeks between our original announcement and your reading this. I believe it's been a modest success.

BAA Moot 2012

Phil Last (phil.last@ntlworld.com)

“In the proud tradition of APL Mooting from Ray Cannon’s legendary early moots via Paul Mansour’s Kefalonian and Tuscan extravaganzas to Stephen Taylor’s inspired Iverson College in Cambridge we bring you the 2012 BAA AGM and Moot to be held at the YHA Lee Valley Youth Hostel, Cheshunt, on Friday-Sunday, 27-29 April.”

Thus was announced the first Moot to be held under the aegis of the BAA for seven years.

In 2010 BAA-London had been asked to organise the BAA AGM and hosted it in the same venue as our regular meetings. We did the same in 2011 but this year, 2012, in my joint role as chief culprit of BAA-London and activities officer of the BAA I wanted to make a change. Following the precedent of 2009 when the AGM was held as a part of the BAA Conference at Reading, I decided to organise a Moot and have the AGM as the opening event.

Kai Jaeger was kind enough to set up a wiki moot.aplwiki.com so all the planning and propagation of information could take place in one place and in full view.

I was unfortunate enough never to have attended one of the earlier moots that Ray Cannon held in a variety of Village Halls. I did attend both of Paul Mansour’s European adventures and Stephen Taylor’s working week. I assumed that of all of them Ray’s were the events closest in spirit to the workshop meetings I half remembered from among the regular meetings that the BAA used to hold at Imperial College. I hoped to pitch the event somewhere financially and comfortably in the middle. I thought perhaps not too many people would want to camp out or bed down in a barn. A Villa or full board in a Hotel was likely to put it beyond the price range of many, some of whom might also be paying to take themselves to other conferences that year. So a Youth Hostel with conference facilities and catering seemed like a good compromise. There being one not a mile from my house and that in some of the most beautiful Green Belt North of the Thames helped me to decide.

The BAA-London meetings had developed into a random alternation between fairly formal presentations and totally informal discussions. For this I was hoping for more of a workshop atmosphere but the attendees would be the ones to decide what happened.

The concept of a Moot was less universally comprehended than I'd thought. Within a few days of the announcement this appeared in one of the APL discussion forums:[1] Rohan Jayasekera: "You might attract more people if you were to tell them what a moot is. moot.aplwiki.com doesn't explain it either. (Google didn't help me either.)" The ensuing discussion continued for a week with many contributors.

Insufficient effort was put in to attract attendees specialising in dialects J, K, Q et cetera but after Devon McCormick's interjection "Do you think someone who currently works mostly in J would find this useful?" a good contingent of J-users came and contributed greatly to the proceedings.

The weekend began on the Friday afternoon with the BAA AGM after which Jeremy Sutton, one of the head rangers of the Lee Valley Park, the locality of the venue, introduced us to the park and Kai Jaeger gave us a preview of his FiRe utility written in Dyalog APL. This was rounded off with a barbecue that was devoured at tables in the park still wet from the recent rain. The evening was spent renewing old acquaintances, making new ones and discussing recent developments.

The fullest day was Saturday that started out with fifteen people sitting around an inward facing square of tables wondering who was going to start. Most got out laptops and some actually got to work or at least read their mail. Until Dick Bowman suggested that he could have sat and worked in silence more easily at home without the bother of negotiating the public transport system. So he got up and treated us to an interesting introduction to WPF and XAML for generating GUI forms in APL. This was what *I* hoped it would be like.

Andy Shires and John Daintree of Dyalog Ltd. arrived mid-morning. Andy introduced them and they were then patient enough to sit through a long catalogue of unresolved bugs and interface problems that Kai Jaeger and others had found in the Dyalog APL development environment. A wide ranging discussion about many aspects of the development interface showed above everything else that although we are all in some ways dissatisfied, one person's requirement would not necessarily please everyone if implemented just so. One of Kai's points was the lack of a simple way to align trailing comments in the function editor. Among fifteen people there were at least six mutually incompatible suggestions of how to achieve the goal and half-a-dozen of precisely what the goal was. One generally accepted goal being that all comments are left aligned at some minimum distance beyond the longest line of code being commented. Not completely compatible with Jane's suggestion of right

alignment. I liked Dick's idea to left-align them off the right-hand-side of the screen. John gave us an inkling of the difficulty of pleasing everyone but was pleased to have had such an exposure to ordinary users and left us believing that we might have had some influence on the Dyalog development process.

Fifteen of us attended at least a part of the weekend, including visitors from Switzerland and the USA, listed at moot.aplwiki.com/Attendees. During the rest of the weekend we worked, learned, ate, drank and laughed together. Many of us gave impromptu demonstrations of recent work and I hope all went away on the Sunday afternoon not having wasted their weekend. I thank you all.

References

1. Linked in discussion group: APL – A Programming Language
<http://www.linkedin.com/groups/Invitation-Moot-1805002.S.93876375>

A P L

Three blind mice or A tale of three new APL trainees or A story of collaboration

Paul Grosvenor (paul@optima-systems.co.uk)

At the Dyalog conference held in Elsinore in October 2012 I presented a short talk on the Apprentices that Optima and Dyalog had jointly taken on. The key points of that discussion are given here.

We have talked for many years about bringing new blood into the APL community but it has always been easier said than done. Two key problems have always gotten in the way:

- Where do we find our trainees?
- How can a small company provide a proper training infrastructure?

With the introduction of the UK Government's Apprenticeship scheme we found that both of these issues were solved at the same time. By collaborating with the Government and local colleges we were able to utilise the apprenticeship portal, which allows anyone to see what training opportunities are available around the country. In addition the Colleges are able to provide the formal training required for our apprentices to achieve a properly recognised qualification. In this case the qualification would be an NVQ.

In 2012 Optima joined the UK apprenticeship scheme and, together with our local college, advertised a position for an APL trainee. Once the applicants had been thinned out and short listed we gave them some tasks to perform. They were asked to logon to the Dyalog "Try APL" site www.tryapl.org and investigate a few small problems prior to interview.

At this point Dyalog showed an interest in what we were doing and, at one of our regular update meetings, we decided to take on three trainees between us and share in the collective load of teaching them APL.

By now we had spent about 10 weeks from start to finish. We had spent very little money only time and had a very effective collaborative group (Government, College and Corporation) providing for the needs of our trainees over the next 12 months.

Before starting this initiative our aims were broadly;

- Bring new people into our community
- Find talent
- Generate a resource stream
- Explore the interest in the general community
- Generate opportunity for the trainees and for ourselves
- Co-operation with other companies

So far all of our expectations have been met and we are very hopeful for the future. If the coming months work out well we hope to do the same thing next year and meet one of our aims, which is to generate a resource stream.

The collaborative concept is working very well indeed and we very much want to expand this idea out to other companies who also might want to 'share' or 'loan' staff or maybe simply give their existing APL staff experience of other working environments.

So where are we going next;

- Continue the training of our apprentices and identify their strengths, likes, dislikes etc.
- Hopefully the apprentices will complete their year, gain a solid background in APL and have an industry recognised qualification
- If successful repeat next year and subsequent years
- Encourage a staff share with Dyalog
- Expand the concept of a staff share or training share with other companies
- Spread the word

We hope that there will be regular updates on this story over the coming months and that it will generate considerable interest with all of our community.

And by the way, the apprentices have created a blog which aims to record their experiences; please take a look: <http://threeblindmiceapl.wordpress.com>

Watch this space – the boys are coming!

Our first steps into the APL world

Sam Gutsell (samuel@optima-systems.co.uk)

Shaquil Sidiki (shaquil@optima-systems.co.uk)

James Greeley (james@optima-systems.co.uk)

This is a companion article to "Three Blind Mice" by Paul Grosvenor in this issue. Here three apprentice APL programmers introduce themselves and give their first impressions of APL and the community that uses it.



Hard at work learning Dyalog APL on the course taught by Bernard Legrand at Dyalog's Bramley offices

From left to right: Sam Gutsell, James Greeley, Shaquil Sidiki

Sam Gutsell

About me:

I am an apprentice APL programmer at Optima Systems, I come from an academic and conceptual background of programming and have achieved an A-Level in Computing; in completing my A-Level I gained experience with Visual Basic, HTML, MySQL, PHP and Java.

My impressions of APL:

At first glance APL was quite daunting because of the amount of symbols, but with time I started to develop an understanding of how the language is constructed and how the language works. I have adapted to the concepts of APL quite well due to the style of my experience. I completed a four-day course taught by Bernard Legrand which helped give me a better understanding of APL. I have learnt with the little experience I have that the language is a very powerful and unique one and can be uniquely used by everyone using it. I am sure that with the help of the very experienced people around me at Optima Systems I will develop a great understanding of the language and hopefully one day be a skilled APL programmer.

My reflections about the Dyalog'12 Conference

Thank you to everyone who made our first conference so enjoyable, to those of you who led the workshops and presentations thank you very much. We all found them both useful and interesting and hope that you will find our blog both useful and interesting too. The whole experience of meeting everyone was very interesting and we would like to thank everyone for being very welcoming and kind to us absolute beginners.

Shaquil Sidiki

About me:

I'm seventeen years old. I am originally from Lisbon, Portugal and I moved to England when I was fifteen. I am currently working at Optima Systems as an Apprentice APL Programmer. My hobbies are to play and watch football, go to the gym, meet up with my friends and to watch videos on YouTube. I have GCSEs in Mathematics, English Language, English Literature, Chemistry, Physics, Spanish, French, Portuguese and an OCR Double Award in Information Technology.

My impressions of APL:

My impressions of APL so far are great. I think it's a very powerful and interesting language as it uses symbols instead of ordinary words. Also everyone has a different style and therefore some problems have more than one solution. Furthermore, in my point of view APL is just like a spoken language. I say this because for me the symbols are like "the words" and it's not enough to memorise what each symbol means we need to know how to put the symbols together to make the code to work, as in any spoken language it's not enough to know what each word means, we need to know how to put a sentence together.

My reflections about the Dyalog'12 Conference:

It was my first conference and I loved it. In my opinion the conference itself was very well planned. I liked all the presentations; however it was very hard to follow some of them. I must say, the presentations I enjoyed the most were: "Introducing Dyalog Version 13.2" (John Daintree), because I'm really looking forward to this new version as I like updates; "Potential Version 14" (John Scholes and Roger Hui), because once again I like updates and I like to look forward to improvements and new features; "Optimisation across networks" (Paul Grosvenor) because it's related to the company that I'm working for (Optima Systems); "Building an Android Application" (Illse Nell and Danie Maré), because as a young adult I think that phones are very important and I like new technology; and "APL and Raspberry Pi" (Liam Flanagan) because again I love new technology and also it's good to the growth of APL and I also think that the iPad app was very good and interesting, and of course "Three Blind Mice by" by us! I would like to make a special reference to John Scholes for his fantastic presentation "State-Free Programming".

Well done to everyone who made a presentation though. I also would like to thank everyone at the Dyalog conference who made us feel very welcome into the APL community. Overall my experience of the Dyalog'12 Conference was fabulous and hopefully next year I can go again. A special thank you to Optima Systems (Paul Grosvenor) for taking us to the Conference.

James Greeley**About me:**

I am 20 years old and work for Optima Systems as an Apprentice APL Programmer A Levels: Biology, Chemistry, Economics, Business. Interests: Web Design, Good UI, Computers, Gym, Video Games

My impressions of APL:

Having only previously dealt with HTML APL was somewhat a jump into the deep end of the pool. At first its syntax and incomprehensible symbols baffled me, what the hell am I looking at? Why have I got this error? However, having spent more time with APL I find myself understanding the symbols without even thinking, as if it were just another word in my vocabulary.

My main interests about APL would be using its power and quick development capabilities to deploy as back ends to web applications. The perfect example of this was demonstrated to me by Gilgamesh Athoraya who showed me the Cosmos system being developed by Optima Systems. This system used MiServer with a flash front end.

I being one of three apprentices taken on by Optima recently went on a four day course at Dyalog taught by Bernard Legrand it was fascinating to meet the author of the book we'd spent so much time reading and a really fantastic week that I would recommend to all.

I'd like to take this opportunity to thank everyone at Optima Systems for an amazing opportunity and being such a nice company to work for!

User commands

by Dan Baronet (danb@dyalog.com)

Prologue

With Version 12.1 Dyalog introduced “User Commands”. Like system commands, user commands are tools which are available to developers at any time, in any workspace. Unlike system commands, user commands are written in APL. Dyalog APL is shipped with a set of user commands, with APL source code that you can inspect and modify – or use as the basis for writing completely new user commands of your own. User commands are intended to make it easy to write and share development tools. User commands began life as Spice commands in version 11.

This article assumes you know how to create a user command in Dyalog APL. To see how to create one see *Vector* article “Spice for beginners” in *Vector* 24:1. The rest of this article will concentrate on technicalities and tricks instead. If an input line begins with a closing square bracket `]`, the system will interpret the line as a user command, temporarily loading the required code into the session namespace where it cannot conflict with any code in the active workspace, and executing it. For example:

```
)load util
util saved whenever
]fns S*
SET SETMON SETWX SM_TS SNAP
```

Help is easily accessible for user commands:

```
]?fns
Command "fnsLike"
Syntax: accepts switches -regex -date=
```

```
Arg: pattern; returns names of fns matching the pattern given
-regex    uses full regular expression
-date     takes a YYMMDD value preceded by > or <
```

```
Script location: C:\ProgramFiles\D121U\SALT\spice\wsutils
```

As we can see above, the full name of the command is `fnslike`, but unambiguous abbreviations are allowed. The source code is in a file called `wsutils.dyalog` in the folder which is identified in the above output. New user commands can be installed simply by dropping new source code files into the command folder, making them instantly accessible without restarting

any part of the system. A full list of installed user commands is available at any time:

```

    ]?
73 commands:

aplmon  calendar cd          commentalign  cfcompare  compare
(more lines here) ...
varslake          wscompare          wsloc    wspeek  xref

Type "]?+" for a summary or "]??" for general help or "]?CMD"
for info on command CMD

```

Implementation

When an input line begins with a closing square bracket, the system will look for a function named `SE.UCMD` and call this function passing the rest of the input line as the right argument. The default session files (.DSE) contain a function which passes the command to the Spice command processor, which is based on SALT[1]. As a result any Spice commands that you may have developed before are now available as user commands in version 12.1.

Dyalog's user commands are similar in concept to those implemented in other APL systems in the past[2] – but the text-based implementation is intended to allow much easier sharing of development tools.

Using user commands

All *user commands* are entered in the session starting with a right bracket, in the same way that *system commands* start with a right parenthesis.

To execute command *xyz* type

```
]xyz
```

To find all available commands type

```
]?
```

To get a summarized list of all commands type

```
]?+
```

To get more general help type

```
]?? or ]help
```

To find all the available commands in a specific folder type

```
]? \folder\name
```

To get info on command XYZ type

```
]?xyz or ]help xyz
```

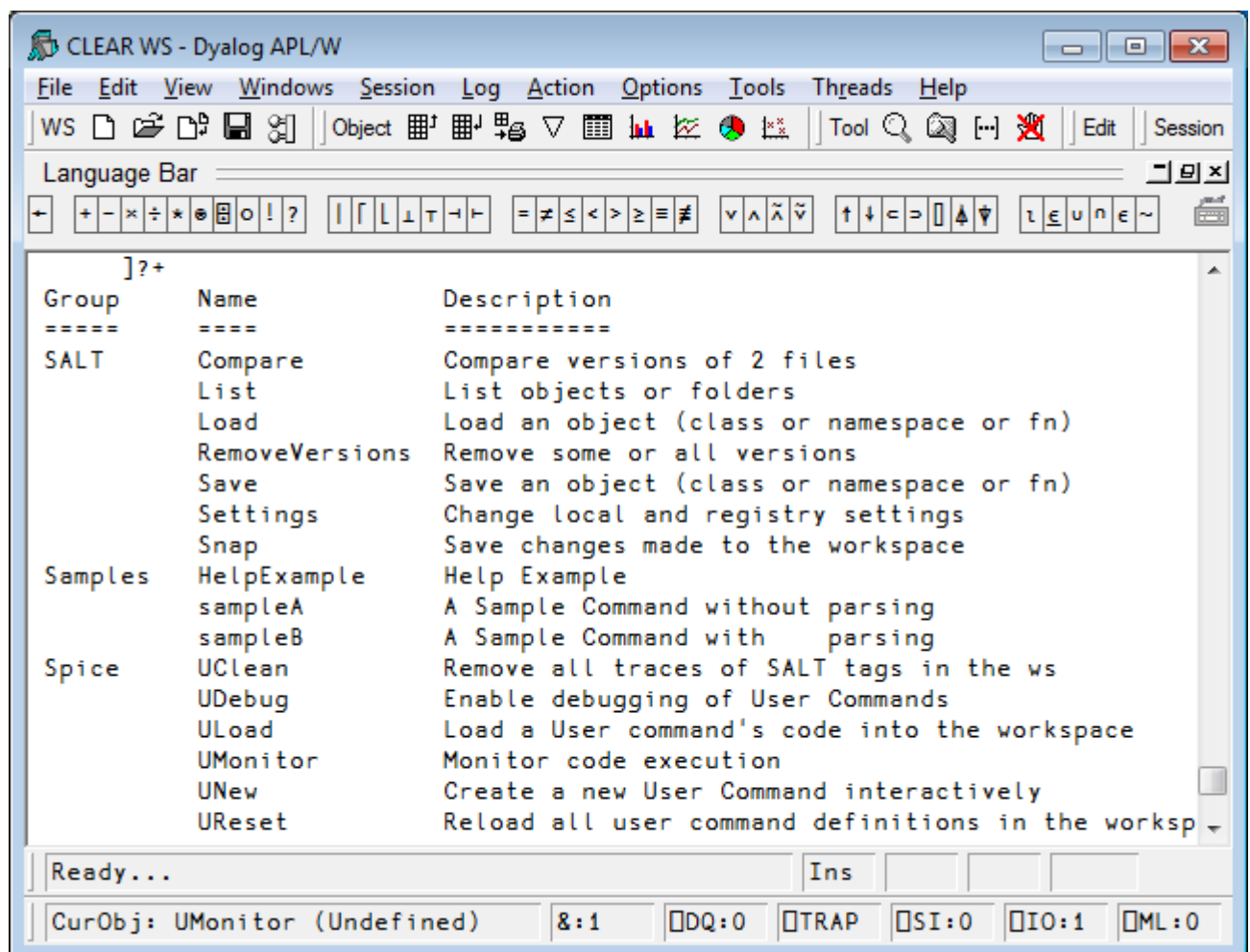
To get detailed help/info on command XYZ type

```
]??xyz
```

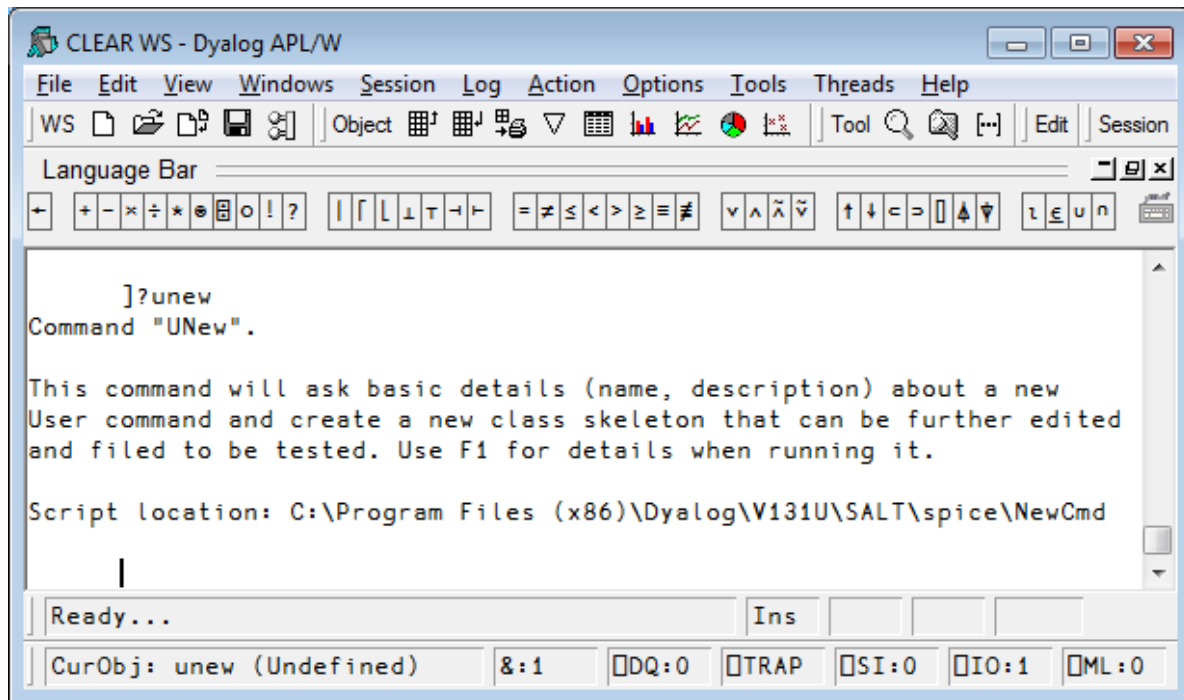
To assign the result of a command to a variable type

```
]nl←cmdx ...
```

Example:



To view help on a particular command type `]?cmdname`. For example, to find help on command `UNew`:



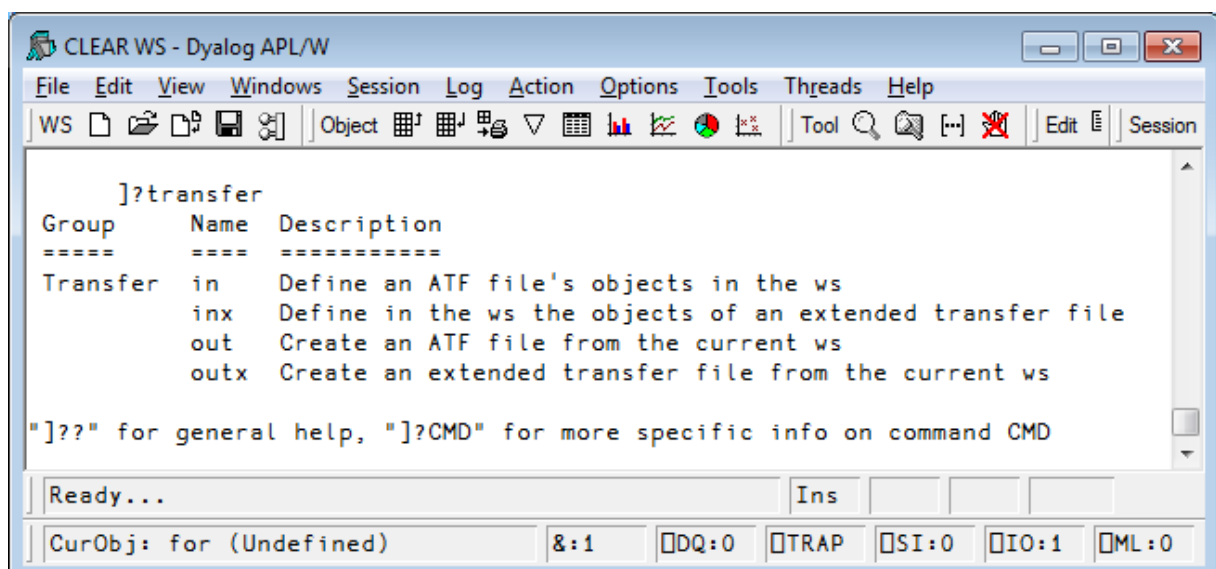
The names of commands are case insensitive, so `UNew` and `unew` are the same command.

Upon hitting Enter, the line is sent to the user command processor which determines which command has been selected, brings in the code, runs it, and then cleans up.

Groups

Commands with common features can be regrouped under a single name. To find all the commands related to a particular group type `]?grpname`

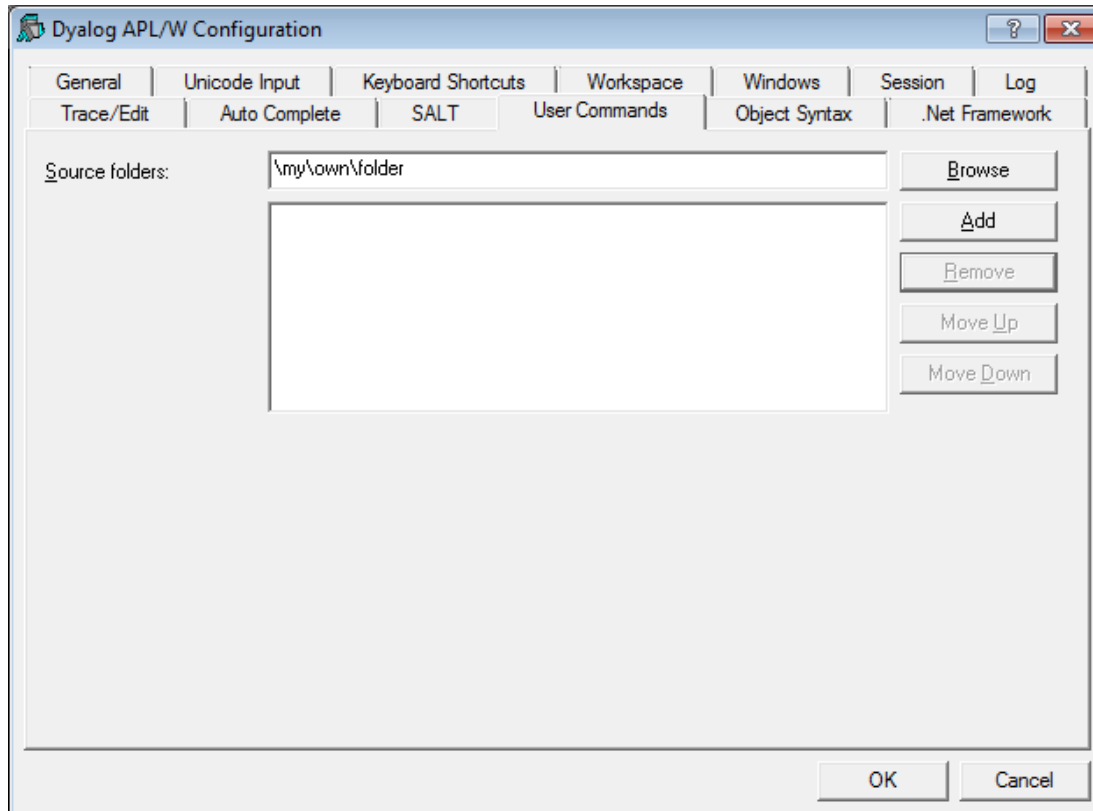
For example, to list all the commands in the `transfer` group:



Locations of commands

By default, the files defining user commands are located in the folder SALT\spice below the main Dyalog program folder. You can change that by specifying a new location.

You can change the location using *Options/Configure User Commands Tab*, just remember the change won't become effective until the next APL restart:



You can also change the location of user commands immediately (no need to restart APL) using the command `]settings`.

`]settings` takes 0, 1 or 2 arguments. With no argument, it displays the current value of ALL settings. With one argument it shows the value of that particular setting. With two arguments it resets the value of the setting specified.

The setting to use for the user command folder is `cmddir`. Thus

```
]settings cmddir
```

will report the folder(s) currently in use. The installed default is `[SALT]\spice`, where `[SALT]` is shorthand for the SALT program folder. If you wish to use another folder, e.g. `\my\user\cmds` you should type

```
]settings cmddir \my\user\cmds
```

Note that this will change the setting *for the duration of the session only*. If you wish to make this permanent you should use the `-permanent` switch:

```
]settings cmddir \my\user\cmds -permanent
```

More than one folder can be specified by separating the folders with semicolons, e.g.

```
]settings cmddir \my\user\cmds;\my\other\goodies
```

The folders will be used in the order specified. If a command with the same name appears in more than one folder, only the first occurrence will be used.

Because spaces are important in folder names you must take care not to introduce any spaces inappropriately.

If you replace the command folder with your own, you effectively disable most installed commands. Only the commands which are part of the SALT and Spice framework will remain active. See below for details on those.

If you wish to add to the existing settings you can either retype the list of folders including the previous ones or precede your new folder with a comma to mean “add” (in front), e.g.

```
]settings cmddir ,\my\spice\cmds;\my\other\goodies
```

will add the two folders specified to any existing setting.

If your folder includes spaces or a dash you should use quotes:

```
]settings cmd '\tmp\a -b c;\apl\with spaces'
```

When you change the command folder it takes effect immediately. The next time you ask for `]?` or a command it scans the new folder(s) specified to cache the info related to all commands: name, description, parsing rules.

Advanced topics

By default, all errors in user commands are trapped, possibly making it difficult to debug commands as you are working on them. To prevent this, you can set the `DEBUG` mode `ON`, as follows:

```
]debug ON
```

Tracing user commands

You can trace into a user commands just like any other APL expression. Because there is a setup involved in executing a user command it can take quite a few keystrokes to get to the actual code: first the `UCMD` function is called then the

Spice processor, and finally your `Run` function. To speed up the process you can ask Spice to stop just prior to calling `Run` by adding a dash at the end of your command expressions, e.g.

```
]command arguments -
```

The dash will be stripped off and APL will stop on the line calling your `Run` function, allowing you to trace into your code.

This will only work when the `DEBUG` mode, as shown above, is `ON`.

Parsing the input

If desired, your input line can be broken down into arguments and switches for you. To do so simply specify in function `List` in your script what are the parsing rules for your command (see article ‘Spice for beginners’ in *Vector* 24:1 for details). The framework will build a namespace containing variable `Arguments` and a variable for each switch mentioned. `Arguments` will be a VTV (a vector of string vectors) containing all your arguments. This namespace will also contain some other utility functions and will be passed as second argument to your `Run` function. If you do not wish to have your input line parsed simply leave the parsing rules empty (‘’) and the framework will set your second argument to whatever you entered on the command line, minus the command name itself.

In the text that follows, `A2` will represent that namespace passed as second argument to your `Run` function.

Default values for switches

A switch always has a value, either 0 if not present on the command line, 1 if present without a value or a *string* matching the value of the switch. For example, if you use `- X=123` then `A2.X` will be a 3-element character vector, not an integer.

If you wish to default a switch to a specific value, you can either test its value for 0 and set it to your desired default, e.g.

```
:If X≡0 ♦ X←123 ♦ :EndIf
```

or you can use the function `Switch` which is also found in your namespace (in the second argument).

Monadic `Switch` returns the value of the switch as if it had been requested directly except that it returns 0 for invalid switches (an error normally).

Dyadic `Switch` returns the value of the left argument if the switch is undefined (0) or the value of the switch if defined but with a twist: if the value of the default is numeric it assumes the value of the switch should also be numeric and will transform it into a number, so if `-X=123` was entered, then (remember `A2` in the following text is your function `Run`'s second argument, a namespace containing all the switches)

```
99  A2.Switch 'X' a default to 99 if undefined
```

will return `(,123)`, *not* `'123'` [3].

Restricted names

If possible, avoid using switches named `Arguments`, `SwD`, `Switch`, `Propagate` or `Delim`, as these names are used by the parser itself (remember that switch names are case sensitive) and included in the second argument. You *can* use these names, but they will not be defined as variables in the argument namespace. They will only be available through function `Switch`, for e.g. `A2.Switch 'SwD'` will return the value of switch named `SwD`.

Long arguments

There are times when arguments contain spaces. The user can put quotes around related elements. For example, if the user command `newid` accepts two arguments, say *full name* and *address* you would set `Parse` to 2 and the user would use, e.g.

```
]newid 'joe blough' '42 Main str mycity'
```

If the command needed arguments *name*, *surname* and *address* (three arguments), the user would not need the quotes around 'joe' and 'blough', but would need them for the third argument to keep the four parts of the address together.

If you want the *last* argument to contain "whatever is left", then you can declare the command as '*long*'. If there are too many arguments, the "extra" ones will be merged into the last one (with a single space inserted between them). To do this, append an `L` after the number of arguments, for example, here, `3L` (plus switches if any).

An example of a logging command requiring one compulsory *long* argument would be coded `1L`:

```
]log    all this text is the argument.
```

Note that if there are multiple blanks anywhere in the text, they will be converted into single spaces.

Short arguments

There are times when you only know the maximum number of arguments. For example there may be 0, 1 or 2 but no more. In that case you would code the parse string as `2S` for two short arguments.

Another example is when you have a single argument which can be defaulted if not supplied. You would then use `1S` (plus switches if any) as parse string. If the user enters no argument (`0=pA2.Arguments`) then your program takes the proper action (e.g. default to a specific value).

Forcing a reload of all commands

When you use a command which the framework does not recognize, it can scan the command folder(s) to see whether new commands have been added. This is the default behaviour when the `setting newcmd` is set to 'auto'. However, if you change this setting to 'manual' or make a change to the short help or the parsing rules you will need to use the command `]ureset` to force a complete reload of all user commands.

SALT commands

Because SALT is part of the user command framework, the commands which implement SALT itself are always available, even if you remove the default command folder from the `cmddir` setting. The commands in question are `load`, `save`, `compare`, `list`, `settings`, `removeversions` and `snap`. If you "shadow" these with your own command with the same names, you will effectively make them invisible, but you will always be able to call them directly by using the functions in `[]SE.SALT`, for example `[]SE.SALT.Load`.

Detailed help

It is possible to provide several levels of help for your commands. When the user enters `]?xyz` the framework calls your `Help` function with the name of the command (here `xyz`) as right argument. If your command accepts a left argument it will be given the number 0 for "basic help".

It is possible to use more than one '?' to specify the level of help required. Entering `]??xyz` is requesting more help than `]?xyz`. `]???xyz` even more so. In effect the left argument to your `Help` function is the number of extra '?' See command `]HelpExample` for details.

More Implementation Details

User commands are implemented through a call to `[]SE.UCMD` which is given the string to the right of the `]` as the right argument and a reference to calling space

as the left argument. For example, if you happen to be in namespace `#.ABC` and enter the command

```
]XYZ -mySwitch=blah
```

APL will make the following call to `⌈SE.UCMD`:

```
#.ABC ⌈SE.UCMD 'XYZ -mySwitch=blah'
```

preserving the command line exactly. The result returned by `UCMD` is displayed in the session.

This means that application code can invoke user commands by calling `⌈SE.UCMD` directly and that if you erase the function, you will disable user commands completely.

By default, `⌈SE.UCMD` calls `Spice`, which implements user commands as described in this document. Its right argument is simply passed on to `Spice` using (here) the call:

```
⌈SE.SALTUtils.Spice 'XYZ -mySwitch=blah'
```

`Spice` will make `UCMD`'s left argument available to your command via global `##.THIS` so you can reference the calling environment if you need to.

Assigning the result of a command

It is possible to assign the result of a command by simply inserting an assignment between `]` and the command name. For example to assign the result of `list` to 'a':

```
]a←list -raw
p⌈←a
<DIR> lib      2010 7 8 17 38 10 879
<DIR> SALT     2010 7 8 17 38 10 889
<DIR> spice    2010 7 8 21 0 52 715
<DIR> study    2010 7 8 17 38 10 930
<DIR> tools    2010 7 8 17 38 10 948
5 5
```

To discard the result, do not specify a variable:

```
]←list -raw
```

To display line by line (as opposed to block by block) use `quad`:

```
⌈pw←30
]disp 2 32p';'
;;;;;;;;;;;;;
;;;;;;;;;;;;;
```

```

;;
;;
]⍵←disp 2 32p';'
;;;;;;;;;;;;;
;;
;;;;;;;;;;;;;
;;

```

Detecting if the result will be captured

It may be interesting to know if the result of a command will be used. For example, the command `]fnsLike` returns a matrix result but can be formatted to be seen like `)FNS`.

Since most calls issued in the session would require one to use `-format` to format the result a la `)FNS`, it is easier to assume this is always the case and make the command show a formatted result whenever called from the session.

For this the framework supplies a Boolean variable, `##.RIU` (Result Is Used), which tells whether the result is captured either because `⍵SE.UCMD` was used specifically or `]Z←cmd` was entered.

In the case above where we would not want `]fnsLike` to format the result we can always use `]⍵←fns` (quad assign the result).

Source file of the command

Should you need to know which file your command came from, the global `##.SourceFile` will provide that information.

Epilogue

The user command implementation of Dyalog is changing but pretty stabilized. Dyalog will continue working on the framework but the community can add to the existing user command stock. There is a project underway to put general interest commands on the web.

References

1. For details on SALT see articles SALT: A Simple APL Library Toolkit in Vector 23/3 and SALT II in 23/4
2. APL/PC was the first to offer user commands. This was carried on to the rest of the APL+ family. It was using component files instead of text files to hold the code and data.
3. The result is always a vector with Switch, this makes it easy to subsequently tell between 0 (switch not there) and ,0 (value supplied by the user)

Public user commands in Dyalog APL

Dan Baronet (danb@dyalog.com)

Version 13.1 is the third release which includes user commands. The user command mechanism is stable but should still be considered experimental to some extent: while the intention is that user commands built with version 13.1 will continue to work in future releases, the mechanism may be extended and many of the user commands shipped with the product are likely to be renamed, moved or significantly modified in the next couple of releases. V13 and 13.1 already have some changes which are mainly additions to the 12.1 set.

Public user commands in Dyalog APL

Use `]?` To list the commands currently installed.

Commands are divided into groups. Each group is presented here along with its commands.

To get examples or more information use `]??` command. For example, to get detailed info on command `wslocate` do

```
]??wslocate
```

The commands are divided into groups. Commands are usually regrouped under a same script with the same group name but it does not have to be this way. A script may regroup several commands with different groups and a group may comprise several commands distributed over several scripts.

Group SALT

This group contains commands that correspond to the SALT functions of the same name found in:

`□SE.SALT`: Save, Load, List, Compare, Settings, Snap and RemoveVersions.

Example:

```
]save myClass \tmp\classX -ver
```

This will do the same as

```
□SE.SALT.Save 'myClass \tmp\classX -ver'
```

Group Sample

There are commands in this group used to demonstrate the use of help and parsing user command lines. You should have a look at the classes and read the comments in them to better understand the examples.

Group Spice

This group contains nine commands: `UClean`, `UDebug`, `ULoad`, `Umonitor`, `UNew`, `USetup`, `UReset`, `Uupdate` and `UVersion`. They are used to manage the user command system itself.

Command `UClean`

This command removes any trace of SALT in the workspace by removing all tags associated with SALT with each object in the workspace. Once you run it the editor will no longer put changes back in the source file(s).

Command `UDebug`

This command turns debugging ON and OFF in order to stop on errors when they happen. Otherwise the error will be reported in the calling environment. It also enables the 'stop before calling the run function' feature which consists in adding a dash at the end of the command as in

```
]mycmd myarg -
```

`UDebug` can also turn system debugging on and off[1]. For example, to turn the `w debug` flag on use

```
]udebug +w
```

to turn it off use

```
]udebug -w
```

Command `ULoad`

This command is used to bring in the workspace the namespace associated with a user command. It is typically used when debugging a user command and you need the code in order to work with it.

Example: load the code for the `CPUTime` command:

```
]uload cpu  
Command "CPUTime" is now found in <#.Monitor>
```

The namespace `Monitor` containing the code for the `CPUTime` user command was brought in from file. We can now edit the namespace and modify the command. When we exit from the editor, the namespace will automatically be

saved back to the script file from whence it came. There is no need for a `usave` command since SALT's `save` command already saves code and subsequent changes are handled by the editor's callback function. However, there is a command for creating a new command, `UNew`, described below.

Command UMonitor

This command turns monitoring ON or OFF. When ON, all active functions see their `□CR` and their `□MONITOR` information paired in the global variable `#.UCMDMonitor`

Results are set in `#.UCMDMonitor` after each invocation of a command.

`-var=` sets the name of the variable to store the result instead of `#.UCMDMonitor`

Command UNew

This command is used to create a namespace containing one or more user commands.

It saves you from having to create a new script from scratch.

It creates a form which is used to input all the basic information about the commands contained in a Spice namespace: the command names, their groups, their short and long description and details of switches.

Each command's information is entered one after another.

When finished it creates a namespace which you can edit and finally save as a file.

Command UReset

Forces a reload of all user commands – this may be required e.g. after modifying a command's description or parsing rules which are kept in memory.

Command UUpdate

This command will update your current version of SALT and user commands to the latest version. This command must be run manually as prompts are issued to do the work although the `-noprompt` switch allows to bypass them.

Command UVersion

This command reports various version numbers: for APL, SALT, .NET and UCMD itself. If given the name of a file containing a workspace it will display the minimum version of Dyalog necessary to `)LOAD` the workspace.

Group svn

This group contains a series of commands used as cover to SubVersion functions of the same name. For example, `svnci` is equivalent to `svn ci` and commits changes made to the current working copy. This is only useful if you have SubVersion installed and in use.

Group SysMon

This group contains three commands for measuring CPU consumption in various ways: `CPUTime` simply measures the total time spent executing a statement, `Monitor` uses `MONITOR` to break CPU consumption down by line of application code, and `APLMON` breaks consumption down by APL Primitive.

Command APLMON

From version 12.0, Dyalog APL provides a root method which allows profiling of application code execution, breaking CPU usage down by APL primitive rather than by code line. The `APLMON` command gives access to this functionality.

As with `Monitor`, you can either run the command with the switch `-on` to enable monitoring, run your application, and then run the command again with the switch `-report` to produce a report, or you can pass an expression as an argument, in which case the command will switch monitoring on, run the expression, and produce a report immediately. The only other switch is `-filename=`, which allows specification of the `APLMON` output file to be used. If it is omitted, a filename will be generated in the folder which holds your APL session log file.

Examples:

```
]aplmon p{+/1=ωvιω}"ι1000 -file=\tmp\data
1000
Written: C:\tmp\data.csv
```

The above command generated a log file name, enabled `APLMON` logging, ran the expression and switched `APLMON` off again. You can report on the contents of this file using the “aplmon” workspace, or send it to Dyalog for analysis.


```

)load aplmon
InitMon '\tmp\data.csv'
Total CPU Time    = 0.15 seconds
Total primitives =      5,003

```

		count	sum	hitcount	sum time	pct	time
1.	or	7		1,000	0.136557	94.03	
2.	equal	6		1,000	0.00454	3.13	
3.	iota	1		1,001	0.003087	2.13	
4.	plus slash	6		1,000	0.001038	0.71	

Command CPUTime

This command is used to measure the CPU and Elapsed time required to execute an APL expression. There are two switches, `-repeat=` which allows you to have the expression repeated a number of times and/or some period of time and `-details=` which specifies how much details should be included. By default, the expression is executed once. The report always shows the average time for a single execution.

It can also accept a combination of both iterations and period, for example the maximum between 10 iterations and 1000 milliseconds. If 1 second is not enough to run the expression 10 times it repeats until the expression has been executed 10 times. On the other hand if the expression ran 10 times in less than 1 second it continues to run until 1 second has gone by. It would be specified this way: `-repeat=10 [1s`

With `-details=none` only the numbers are returned as a 2 column matrix (CPU and elapsed), 1 row per expression.

With `-details=ai` only the same numbers plus the 2 `AI` numbers are returned (Nx4 matrix).

With `-details` or `-details=all` nothing is returned; instead, a report that includes the number of times repeated and the `AI` and `MONITOR` numbers is shown.

Examples:

```

]cputime {+/1=ωvιω}¨ι1000
* Benchmarking "{+/1=ωvιω}¨ι1000"
  Exp
CPU (avg):    63
Elapsed:      67

]cputime {+/1=ωvιω}¨ι1000 -repeat=1s
* Benchmarking "{+/1=ωvιω}¨ι1000", repeat=1s
  Exp

```

```
CPU (avg):    54.6
Elapsed:      55.2
```

```
]cpu {+/1=ωvιω}~ι100 ι~ι1e6 -details=ai -rep=50
0.64 0.46 0.64 0.46
30.58 30.52 30.58 30.52
```

The last example shows the 2x4 result for the 2 expressions tested.

Command Monitor

This command is used to find out which lines of code in your application are consuming most CPU. You can either run the command with the switch `-on` to enable monitoring, run your application, and then run the command again with the switch `-report` to produce a report, or you can pass an expression as an argument, in which case the command will switch monitoring on, run the expression, and produce a report immediately. Other switches are:

Switch	Effect
<code>-top=n</code>	Limits report to the n functions consuming the most CPU
<code>-min=n</code>	Only reports lines which consume at least n% of the total, either CPU or Elapsed time
<code>-fns=fn1,fn2,...</code>	Only monitors named functions
<code>-caption=text</code>	Caption for the tab created for this report

Examples:

```
]monitor -on
Monitoring switched on for 44 functions
```

```
5↑[1]NTREE '[]SE'
[]SE (Session)
├─Chart (Namespace)
│   ├──CheckData (Function)
│   ├──Do (Function)
│   └─DoChart (Function)
```

```
]mon -rep -cap=NTREE
```

(Pops up the following dialog)

Function	Count	CPU	Elapsed	%CPU	Code
NTREE	392	673	488	100.0	R-([D]NTREE ON;C;N;[IO];[M]
	392	16	3	2.4	[18] [M]+3 A For APL2-style parti
	392	0	7	0.0	[26] -((=ON)ε,"[SE] '# ' '.)pL3 A ...avoid a few [NC] f
	392	15	1	2.2	[42] L3: A We have a fully-qualified name and we know what cle
	392	16	7	2.4	[45] N=(C+2)>11+N
	215	47	5	7.0	[47] N-ON [MG]Type' A Get 'Namespace' or G
	392	16	10	2.4	[51] R-(φ\φ(Cs0)∨1,'. '≠1+ON)/ON A Just tail of name un
	392	0	5	0.0	[52] R+R,' (' ,N,')' A Name and type
	215	0	7	0.0	[60] C-ONε'[M] 1 2 3 4 9' A Children
	54	486	336	72.2	[63] C-(D-1)NTREE"C A Recurse
	54	15	17	2.2	[71] C[(εpC),"ε"((1N),"1)]-Nt'L' A Last child: 'L', gra
	54	0	6	0.0	[81] R-([2]~' ('εR)ε"ε[2]R A Split names and type
	54	0	5	0.0	[83] R-([1="R),' ' ,(' '∨,εC)/C A Align all names and

Command Profile

This command is used to fine tune your application. This is a more complex command that will be the subject of a separate article.

Group System

This group contains operating system related commands.

Command assemblies

This command lists all the assemblies loaded in memory.

Command cd

This command will change directory in Windows only. It reports the previous directory or the current directory if the argument is empty.

Example: switch to directory `\tmp` for the remaining of the session:

```
]cd \tmp
C:\Users\Danb\Desktop
```

Command EFA

This command will associate Windows® file extensions `.dyapp` and `.dyalog` with a specific Dyalog APL version. This is useful only if you have several versions installed and wish to change the current association made with the latest install.

Group Tools

This group contains a series of commands used to perform tasks related to everyday activities.

Command calendar

This command is similar to Unix' cal program and displays a calendar for the year or the month requested.

Example:

```
]cal 2010 3
    March 2010
Su Mo Tu We Th Fr Sa
   1  2  3  4  5  6
  7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

Command Demo

Demo provides a *playback* mechanism for live demonstrations of code written in Dyalog APL

Demo takes a script (a text file) name as argument and executes each APL line in it after displaying it on the screen.

It also sets F12 to display the next line and F11 to display the previous line. This allows you to rehearse a demo comprising a series of lines you call, in sequence, by using F12.

For example, if you wish to demo how to do something special, statement by statement you could put them in file `\tmp\mydemo.txt` and demo it by doing

```
]demo \tmp\mydemo
```

The extension TXT will be assumed if no extension is present.

The first line will be shown and executed when you press Enter. F12 will show the next which will be executed when you press Enter, etc.

Command dinput

This command is used to test multi line D-expressions.

Example:

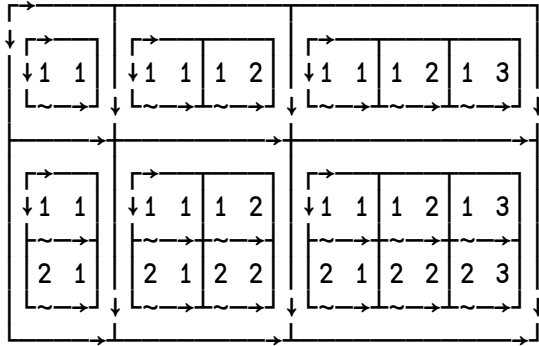
```
]Dinput      A multi-line expression
....{        A dup:
.....ω ω
....}{      A twice:
.....αα αα ω
....}7
7 7 7 7
```

Command disp

This command will display APL expressions using boxes around enclosed elements as per the familiar `disp` function.

Example:

```
disp ⍝⍝2 3
```

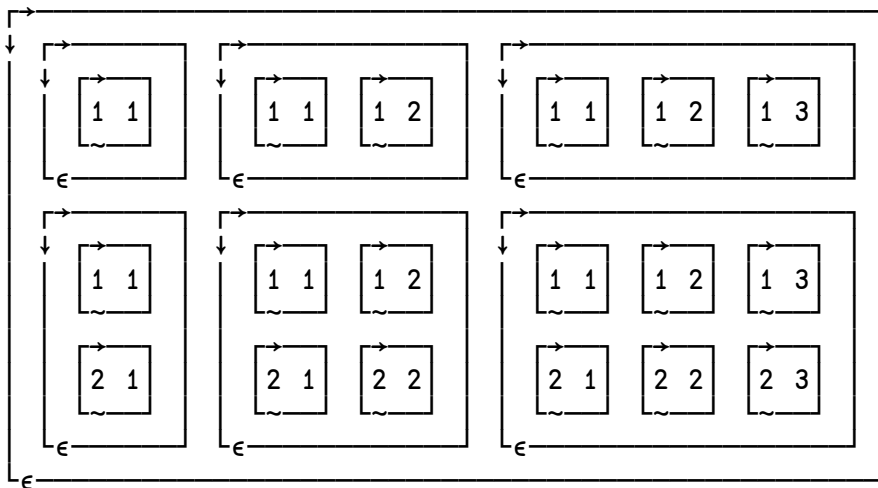
**Command display**

This command will display APL expressions using boxes around enclosed elements as per the familiar `DISPLAY` function.

Note that this command is different from the `disp` command just like the 2 functions `disp` and `DISPLAY` are different and you must enter at least 'displ' to use it.

Example:

```
]display ⍝⍝2 3
```

**Command dmX**

This command will provide more detailed information about the last APL error. It uses `⍎DMX`, the new Display Message eXtended system variable to do its work.

Command factorsof

This command will return the factors that constitute a number.

Example:

```
]fac 123456789
3 3 3607 3803
```

Command FFind

This command searches the *.dyalog* files in, by default, the current SALT working directory for the string given as argument in SALT script files. It needs one long argument which may be a *.Net* regular expression[2].

It reports all the hits in each script file where found.

To search different directories use the switch `-folder` to specify the new location.

`-options` will accept a value of `I` (insensitive), `S` (singleline mode) or `M` (Multiline mode – the default) to change search behaviour.

`-types` will accept the extensions to use when searching (default *.dyalog*)

`-regex` will consider the argument to be a regular expression

Example:

```
]ffind \b(abc|\w{7})\b -folder=\tmp -typ=txt log -r
```

will find *abc* or all 7 letter words in all *.txt* or *.log* files in *\tmp*, and below. `-r` is short for `-regex`; without it the exact text above would be looked for.

Command FnCalls

This command is used to find the calls made by a program in a script file or in the workspace.

It takes one or two arguments: the function name and the namespace or filename where it resides (default current namespace). With switch `-details` it can provide extra details on all the names involved such as locals, globals, unused, recursively called, etc.

With switch `-treeview` it will show the result in a treeview window instead of the session log.

If the switch `-file` is provided the namespace is assumed to be the name of a file.

Example:

```

]fncalls Spice '\Dya -APL\12.1\SALT\SALTUtils' -fil
Level 1: →Spice
A Handle KeyPress in command window
A The function can also be used directly with a string
F:isChar          F:isHelp          F:isRelPath
F:BootSpice       F:GetSpiceList     F:SpiceHELP

Level 2: Spice→isChar
...
Level 2: Spice→BootSpice
A Set up Spice
F:GetList         R:Spice

Level 3: BootSpice→GetList
A Retrieve the list of all Spice commands
F:getEnvir        F:lCase          F:splitOn        F:ClassFolder

Level 4: GetList→ClassFolder
A Produce full path by merging root and folder name
...
```

NB: the argument '\Dya -APL\12.1\SALT\SALTUtils' is surrounded by quotes because it contains a dash

At each level the calling function is followed by the called function which is detailed. It lists each function called preceded by either an F (for function) or an R (for recursive call). We can see at the 1st level that function `Spice` calls 6 other functions and at the second level function `isChar` calls nothing and `BootSpice` calls 2 functions: `GetList` and `Spice`, recursively. At the third level `GetList` calls 4 functions and so on.

With the switch `-full` the output repeats for already shown functions. This may produce output where the same function calls may be different if objects are shadowed up the stack.

With the switch `-details` each object is preceded by either F or R as above or a character meaning:

```

o: local
G: global
!: undefined local
↑: glocal (global localized higher on the stack)
L: label
l: unreferenced label
*: previously described in the output
```

NB The abbreviation `*: previously described in the output` won't happen when switch `A-full` is used.

With the switch `-isolate` all the object names required to run the function given as argument are returned in matrix form. Moreover, if `-isolate` takes an unused APL name as value (e.g. `-isolate=newNs`) then all the objects are copied into the new namespace. This allows you to modularize code by isolating individual sections.

Command **FReplace**

This command searches the `.dyalog` files in, by default, the SALT current working directory for the string given as first argument in SALT script files and replaces occurrences by the second (long) argument. It needs two arguments which may be *.Net* regular expressions[3] if the switch `-regex` is used.

To work on a different directory use the switch `-folder` to specify the new location.

`-options` will accept a value of `I` (insensitive), `S` (singleline mode) or `M` (Multiline mode – the default) to change search behaviour.

`-types` will accept the extensions to use when searching (default `.dyalog`)

`-regex` will consider the arguments to be regular expressions

Example:

```
]frepl Name:\s+(\w+)\s+(\w+) Name: $2, $1 -f=\tmp -r
```

will reverse every occurrence of two words (`-r` means this is a regular expression) when they follow `Name: ,` i.e

```
Name: Joe Blough
```

will become

```
Name: Blough, Joe
```

in every file it finds in the directory `\tmp`.

Command **fromhex**

This command will display a hexadecimal value in decimal

Example:

```
]fromhex FFF A0
4095 160
```


Command ftttots

This command will display a number representing a file component time information into a TS form (7 numbers).

```
]ftttots 3[]frdci 4 1
2011 3 10 23 16 28 0
```

Command GUIProps

This command will report the properties (and their values), *childlist*, *eventlist* and *proplist* of the event given as argument or, if none provided, the object on which the session has focus (the object whose name appears in the bottom left corner of the session log).

Command latest

This command will list the names of the youngest functions changed (most likely today, otherwise of the last changed day), the most recently changed first.

Command open

This command opens a specific file with the proper program, e.g. an *Excel* spreadsheet, or it open *Explorer* (under Windows®) to view the files in the main SALT folder. It replaces `]explore` with has been decommissioned.

Command Summary

This command produces a summary of the functions in a class in a script file. It takes a full pathname as single (long) argument. If the switch `-file` is provided the name is assumed to be a file.

```
]summary []SE.Parser
name      scope  size  syntax
fixCase           24   n0f
if             24   n0f
init          PC   4500   n1f
xCut           532   r2m
Parse          P   5748   r1f
Propagate      S   1220   r2f
Switch         1152   r2f
```

Scope shows S if shared, P if public, C if constructor and D if destructor

Size is in bytes

Syntax is a 3 letter code:

```
[1] n=no result, r=result
[2] # of arguments (valence)
[3] f=function, m=monadic operator, d=dyadic operator
```

Command tohex

This command will display a number in hexadecimal value

Example:

```
]tohex    100 256
64 100
```

Command tohtml

This command will produce HTML text that displays a namespace or a class in a browser. It accepts five switches

-title= will take a string to be displayed at the top of the page, e.g.
-title=<center>My best Class</center>

-full will include the full HTML code, including the <head> section before the <body>

-filename= write the result to the file specified

-clipboard will put the result on the clipboard, ready to be pasted elsewhere

-xref will produce a Cross-reference of the names used in the class in relation to all the methods

Command WSpeak

Executes an expression in a temp copy of a workspace. As its name suggests, WSpeak is used to view, rather than to change, a saved workspace; any changes made in the copy are discarded on termination of the command. A `wsid='.'` means the saved copy of the current workspace.

```
]WSpeak wsid [expr ...] A expr defaults to ±[]LX
```

Example: execute the `queens` program from the 'dfns' workspace

```
]wsp dfns 0 disp queens 5
```

Command Xref

This command returns a Cross-reference of the object given. If the switch `-file` is provided the name is assumed to be a file.

It produces a very crude display of all references on top against all functions to the left. At the intersection of a function and a reference is shown a symbol denoting the nature of the reference in relation to the function: *o* means local, *G* mean global, *F* means function, *L* means label.

Example:

```

]xref  \Program Files\Dyalog\SALT\lib\rundemo -file
      ccfkllnpsssszzzFFILNPPS
      llieaiaa_cn...iinieaoc
      .lybnms ri.NRllinxtr
      Tes eet ip.eaeetethni
      e . . e p .sw . . . p
      x . . . t .t. N . . t
      t . . . . . . . . .
      - - - - : - - - - : -
Edit   . . . . o . . G . : G
Init   . . . .o: . . . .F:GG
Load   .o. .o. : oGGG.F. G G

```

As can be seen in this report, name `script` is a local in function `edit`. The characters *dot*, *dash* and *semi colon* only serve as alignment decorators and have no special meaning.

Group Transfer

This group contains four commands: *in*, *out*, *inx* and *outx*. *In* and *Out* read or write APL Transfer Files in the standard ATF format, and should be compatible with similarly named user or system commands in other APL implementations. *inx* and *outx* use a format which has been extended to represent elements of a workspace which have been introduced in Dyalog APL since the ATF format was defined.

See the Dyalog APL Windows Workspace Transfer v12.1 for more details.

Group wsutils

This group contains several commands used for workspace management and debugging. Some of the commands take a filter as an argument, to identify a selection of objects. By default, the filters are in the format used for filtering file names under Windows or Unix, using `?` as a wildcard for a single character and `*` for 0 or more characters. For example, to denote all objects starting with the letter “A” you would use the pattern `A*`.

Regular expressions can be used to select objects. You indicate that your filter is a regular expression by providing the switch `-regex`.

The commands which accept filters are *fnslike*, *varslike*, *nameslike*, *reordlocals*, *sizeof* and *commentalign*. They all apply to *the current namespace*, i.e. you cannot supply a dotted name as argument.

Also, very often the same command will accept a `-date` switch which specifies the date to which the argument applies. This will typically be used when functions

are involved, for example when looking for functions older than a date, say 2009-01-01, you would use `-date=<90101`. The century, year and month are assumed to be the current one so if using this expression in 2009 using `-date=<101` would be sufficient. You can use other comparison symbols and `-date=#80506` would look for dates different than 2008-05-06. Ranges are possible too and `-date=81011-90203` would look for dates from 2008-10-11 to 2009-02-03 included.

The value of *date* is `<90101`, the `<` is included which is why the syntax includes both `=` and `<`.

Command `cfcompare`

This command compares 2 APL component files. It accepts the same switches as `varcompare` plus switch `-cpts` which list the components to compare. It accepts two arguments: either the path of files or their tie number if already tied. They can be optionally followed by *passnumber* if the file(s) require one for reading

Example:

```
]cfcompare \tmp\abc:123 27 -cpts=5 7, 9-99
```

This will compare file `\tmp\abc` with the file using 27 as tie number. The *passnumber* 123 will be used for file `\tmp\abc` to read the components. Only components 5, 7 and 9 to 99 will be compared.

Only those components with the same number will be compared; if the 1st file's components range from 1 to 10 and the second's range from 6 to 22 then only components 6 to 10 will be compared. In this case because a set has been specified only components 7, 9 and 10 would be compared.

If either the path of the name of the second file is the same as the 1st's then `=` can be used to abbreviate the name. For example to compare files ABC and XYZ in folder `\tmp\long\path` you can enter

```
]cfcomp \tmp\long\path\ABC =\XYZ
```

See command `varcompare` for the list of other accepted switches.

Command `CommentAlign`

This command will align all the end of line comments of a series of functions to column 40 or to the column specified with the `-offset` switch.

The arguments are DOS type patterns for names which can be viewed as a regular expression pattern if switch `-regex` is supplied. The `-date` switch can also be applied.

The result is the list of functions that were modified in column format or in `)FNS` format if switch `-format` is supplied.

Example:

```
]commentalign HTML* -format -offset=30
```

This will align all comments at column 30 for all functions starting with *HTML* and display the names of all the functions it modified in `)FNS` format

Command `fncompare`

This command will show the difference between 2 functions, including time stamps. It can handle large functions and has switches to trim the functions first, exclude the time stamps, etc.

Example:

```
given:      ∇fna                      ∇fnb
            [1] same line              [1] same line
            [2] fna line 2              [2] fnb line 2
            [3] same line 3              [3] same line 3
            [4] A comment deleted        [4] new common line
            [5] new common line          [5] A new comment
            ∇                          ∇
```

```
]fncomp fna fnb
←[0] fna
→   fnb
[1] same line
←[2] fna line 2
→   fnb line 2
[3] same line 3
←[4] A comment deleted
[5] new common line
→   A new comment
```

Switches

- `-normalize` removes excess space at the ends of each line
- `-delins` change the delete/insert characters
- `-exts` exclude timestamps in comparison
- `-zone` specify how many lines to show before and after each difference
- `-nolastline` exclude the last line of each function (ex ignore SALT tag lines)

Command `fn diff`

This command will show the different lines between two functions by showing the differences side by side. It is more suited for small functions. With the same example functions as in `fn compare`:

Example:

```

]fn diff fna fnb
.fna.      .      .      .      .      | .fnb.      .      .      .      .
.fna.line 2 .      .      .      | .fnb.line 2 .      .      .      .
A comment deleted      .      | A new comment      .      .      .

```

Command `fn like`

This command will show all functions names following a pattern in their names. It accepts the `-regex`, `-date` and `-format` switches.

Command `names like`

This command will show all objects following a same pattern in their names. Each name will be followed by the class of the name.

It accepts the `-regex`, `-date` and `-format` switches.

Example: find all names containing the letter *a*:

```

]names like *a*
aplUtils.9      disableSALT.3      enableSALT.3
commandLineArgs.2 disableSPICE.3    enableSPICE.3

```

Command `reorder locals`

This command will reorder the local names in the header of the functions given in the argument. The argument is a series of patterns representing the names to be affected. It accepts the `-regex`, `-date` and `-format` switches.

Command `sizeof`

This command will show you the size of the variables and namespaces given in the argument. The argument is a series of patterns (including `none=ALL`) representing the names affected. It accepts the `-class` switch to specify the classes involved and the `-top` switch to limit the number of items shown.

Example:

```

)obs
NStoScript      aplUtils      test
)vars
CR      DELINS Describe      FS
]size -top=4 -class=2 9
NStoScript 132352

```

```
aplUtils      40964
test          31996
Describe      10128
```

Command supdate

This command will update a namespace script with newly added variables and functions.

This can come in handy when you've added code and data inside a scripted namespace.

Example:

```
]load myns
)cs myns
V ← 19
[]fx 'myfn' '2+2'
A Now update the script to include these new objects
]supdate
Added 1 variables and 1 functions
```

Command varcompare

This command will compare two variables including namespaces which contain functions and other variables and namespaces. For this reason it includes the same switches as command `fncompare` plus the following:

```
-exnames=    exclude names from the comparison
-nlines=     show only the 1st n lines of each variable not in the
              other object
-show=       show only specific sections of the comparison report
-nssrc       force the use of source for namespaces if they exist
              instead of comparing object by object
```

See `]??varcompare` for details

Command varslike

This command will show all variables following a same pattern in their names. It accepts the `-regex` and `-format` switches.

Command wscompare

This command will show the difference between two workspaces. It is a combination of the commands `fncompare` and `varcompare` being run on entire workspaces. The workspaces are first copied into temporary namespaces and the comparison then performed. It includes the switches of `fncompare` and `varcompare` plus the following:

-exstring= exclude object containing this string
 -gatheroutput gather all the output and return it as a result
 (can be quite large)

Command `wslocate`

This command will search strings in the current namespace. It accepts a number of switches that allow it to screen out hits in comments, text, etc. It accepts normal and regular expressions and will perform replacement on most objects. It is a very comprehensive command. For example it allows you to find where 3 names are assigned 3 numbers. See its documentation for details: `]?wslocate`

Example: look for the words ending in *AV* (syntactically to the right), regardless of case, in text only (exclude Body and Comments):

```
]wslocate AV -syntactic=r -insensitive -exclude=bc
Search String (Find and Replace) for Dyalog V6.01
```

```
▽ #.xfrfrom (3 found)
[57]  Ɑ(ΔΔtrans=2)/oNS, 'ΔAV←bUf '
           ^
[72]  ΔΔCodT←ΔΔCodT, (ΔΔtrans=2)/ '%□av[ %ΔAV[ '
                        ^      ^
```

References

1. System debug flags are used to debug the interpreter itself. See the User Guide for details on this topic.
2. To look for or replace strings in the workspace use command `WSLOCATE`
3. For *.Net* regular expressions see
[http://msdn.microsoft.com/en-us/library/az24scfc\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/az24scfc(VS.71).aspx)

␣S and ␣R

Dan Baronet (danb@dyalog.com)

With version 13 Dyalog introduced two new system operators to deal with regular expressions. This article discusses these operators ␣S and ␣R

The implementation uses the open-source regular-expression search engine PCRE (Perl Compatible Regular Expressions), a library, which is built into Dyalog APL. The regular expression syntax which the library supports is not unique to APL nor is it part of the language. Dyalog introduces new features and the way to use it is more like APL.

␣S and ␣R are the new system operators that use the PCRE engine to do their job. Like all other functions, ␣S/R work with Unicode.

Who should read this

This article assumes the reader knows a little bit about regular expressions. Although some basic syntax is given, few details will be provided. It will focus on the APL part of it instead and the examples will use simple regular expressions. Some advanced examples are shown towards the end for those interested in that sort of thing.

Basics

␣S is used to report information on string matches and ␣R is used to make replacements in situ. They use regular expressions for that.

A regular expression, often called a *regex*, is a string representing a pattern, a way to match text (e.g. words) in another piece of text. Here are 2 examples.

1. Find the *offset* of all single characters followed by 'at'

```
( 'at' ␣S 0) 'The cat sat on the mattress'
4 8 17
```

2. Change to uppercase all occurrences of single letters followed by 'at'

```
( '.at' ␣R '\u&') 'The cat sat on the mattress'
The CAT SAT on the MATtress
```

␣S (and ␣R) being a dual operator, it takes 2 operands and returns a function. The left operand is always the pattern(s) and the right operand is the transformation, if any, to apply. It can either be a code (example 1 above: 0

means return the offset of each match), a transformation pattern (example 2: '\u&' [1] means Uppercase the whole match), or a user function discussed below. The result, a function, is applied to an argument which holds text or a number tied to a native text file (including Unicode encoded files).

Regular expressions, or regexes, can specify more complex patterns and an expression to find the length AND offset of *cat* or *lion* would be:

```
      ('cat|lion' s 1 0) 'the cat sat on the medallion'
3 4      4 24
```

The result is 2 pairs of numbers: 'a 3 long match at offset 4' and 'a 4 long match at offset 24'.

Note that *lion* was found in *medallion*. Regexes allow you to specify whether a letter should be on a word boundary by using \b before and/or after it. For example to look for the offset of exact words *cat* and *lion* (using - this time to separate the right operand from the argument):

```
      p '\bcat\b|\blion\b' S 0 - 'the cats sat on the medallion'
0
```

Nothing found. And this is right, the words cat and lion nowhere appear by themselves.

There are many rules which are out of the scope of this article. They can be found either on the Net or in Dyalog's manuals; too many to list them all here. It suffices to say that you can look for patterns in a very flexible and concise manner. Here is a short list of pattern characters and what they mean:

.	stands for any char but line delimiter
{n}	stands for N times
?	stands for 0 or 1 time
+	stands for 1 or more times
*	stands for 0 or more times
	means OR
()	group elements or alternatives
[]	group possible characters
\b	means "at a word Boundary"
^	means the beginning of a line
\$	means the end of a line

For example to look for a string representing a number smaller or equal to 2012 (remember we are dealing with characters here) the pattern “\b (1\d{3} | 200\d | 201[012]) \b”² would do it. It means: look for either

a ‘1’ followed by 3 digits (\d is any numeric digit – 0 to 9, {3} means repeat 3 times)

OR (the vertical bar |)

‘200’ followed by a digit (\d)

OR (|)

‘201’ followed by either ‘0’, ‘1’ or ‘2’ (square brackets regroup the set of characters that the next character should be in)

The parentheses regroup the 3 alternatives and the \b before and after ensures the string found is not part of a bigger number (like 120009)

You can look for patterns that repeat, recurse, and do all kinds of things.

But like anything else you need to spend a bit of time to do more fancy things.

Again, if you want to delve into this, type ‘regular expressions’ in your favourite search engine and let yourself loose. If you want APL examples see “Tools, Part 4: Regular Expressions” in Vector 21:2, p.126.

We will now look into what Dyalog did with this.

Transformations

There are 3 types of results we may want to get from ⍷s :

1. Frequent values like offset, length, line number for which codes (small ints) can be used
2. Transformation (text) pattern like “uppercase the hit”
3. More complex expression for which you must provide a function.

We have seen examples of 1 and 2 above. If you cannot do what you want with those you’ll need to write a monadic function which will accept a namespace as argument. This namespace will contain a series of variables related to the current match:

Block	the text being searched, in line mode ³ this is the line we’re looking at
BlockNum	the line number (⍷IO←0), in line mode again
Pattern	the pattern used
PatternNum	the pattern number (⍷IO←0)
Match	the string that matches the pattern

Offsets	the offset of each sub pattern
Lengths	the length of each sub pattern
Names	the names of each sub pattern
ReplaceMode	whether we are using <code>QR</code> (1) or <code>QS</code> (0)

Examples

1. Find all but the 1st digit of all unsigned integers found in a text

```
'\d(\d+)' QS '\1' - 'dsa 1233 30 3 xyz'
233 0
```

Explanation: the pattern `\d(\d+)` means “find a digit (`\d`) followed by “more-than-1 digits” (`\d+`). The parentheses capture that sub pattern and since it is the 1st (and only) sub pattern it is #1 (#0 is given to the whole match). So ‘1233’ is the 1st match found (and sub pattern 0), in it is the sub pattern ‘233’ which is returned because this is what the right operand (`\1`) says to do. The same thing happens with the second match: 30. So `QS` returns 2 results: ‘233’ and ‘0’. Note that ‘3’, the third integer in the string does NOT match the pattern as there is no “more-than-1 digit” (`\d+`) after the 1st one.

2. Reverse and return all the words in a sentence

This one is impossible to do with a transformation string so we use a function:

```
'\w+' QS {φω.Match} 'The cat sat on the mat'
eht tac tas no eht tam
```

Explanation: each time the `QS` derived function finds a match, e.g. ‘The’, it gives the right operand function a namespace as argument which contains the variables mentioned above. In it is found ‘Match’ which can be referred to and, in this case, flipped and returned. This is done each time a match is found (6 times). So here `QS` returns a vector of 6 text vectors (a VTV).

QR

So far we’ve only seen searches. `QR` is the other new system operator introduced in V13.

It is very much like `QS` except that it performs replacement in the argument string. If we were to run the last 2 examples with `QR` instead we would get this:

1. Remove the 1st digit of all integers >9 :

```
'\d(\d+)' QR '\1' - 'dsa 1233 30 3 xyz'
dsa 233 0 3 xyz
```

Explanation: the pattern `\d(\d+)` means “find a digit (`\d`) followed by “more-than-1 digits” (`\d+`) just like before. The parentheses capture that sub pattern. So ‘1233’ is the 1st match found, and it is replaced by the sub pattern ‘233’ because this is what the right operand (`\1`) says to do. The same thing happens with the second match: 30. So `⊠R` returns the argument minus the 1st digit of each match.

2. Reverse all the words in a sentence

```
'\w+' ⊠R {⊠ω.Match} 'The cat sat on the mat'
ehT tac tas no eht tam
```

Explanation: each time the `⊠R` derived function finds a match, e.g. ‘The’, it gives the right operand function a namespace as argument which contains, among other things, ‘Match’, which can be referred to and, in this case, flipped and used to replace the original match. This is done each time a match is found (6 times) resulting in a modified argument. Note that there is no need to separate the right operand function from the argument since it binds with `⊠R` first.

Details

Normally a regex engine like PCRE works on a string in two possible modes: as a whole, called single line, or as a series of lines delimited by a line delimiter (like NL), called multiline. This affects the way some searches are performed: in single line mode the `^` and `$` only match at the beginning and end of the searched string. In multiline mode they match at the beginning and end of every line in it. For example, an expression like `^a.*b$` would find all lines starting with ‘a’ and ending with ‘b’. The `^` means “the beginning of the line”, the dot (`.`) means “any character but NL”, the star (`*`) means “as many times as possible” and the `$` means “the end of the line”.

`⊠S/R` have the same 2 modes called Document mode (=PCRE *single line* mode) and Mixed mode (=PCRE multiline mode) plus another mode called Line mode. Line mode splits the searched text into logical lines so searches cannot span across them. It effectively treats every line as a different text to be searched. This is something not available with PCRE. Line mode is less prone to WSFULLs. It is the default mode.

APL being array oriented it also works on VTVs (vectors of text vectors). With VTVs each vector represents a line in the text separated (by default) by CRLF and all results are the same in every mode, whether you use a VTV or `(⊖,LineDel)⌈⊃,/LineDel∘,“VTV`.

The Variant operator ¶

A regex program like PCRE is a complex program that assumes some settings to be present. Just like dyadic iota uses implicit arguments like `IO` and `CT`, a regex engine uses implicit arguments that can be modified.

This new primitive operator is used to modify the behaviour of a function. Here it is used to modify the way the resulting function from `S` will be used. Note that the `¶` character is not in `AV`.

Variant takes a function to the left and an array to the right. The array specifies the options that modify the function, resulting in a new function returned by Variant. For example, in

```
F2← F1 ¶ array
```

F2 is now a modified version of F1.

There are many parameters that can be set via `¶` for `S` and `R`. Here are the main ones (the 1st value is the default):

- IC Ignore Case. Possible values: 0 or 1
- Mode Operating mode. Possible values: L=line mode, D=document mode, M=mixed mode
- DotAll Makes the dot match line delimiters too. Possible values: 0 or 1
- EOL What the line delimiter is. Possible values: CR, LF, CRLF, VT, NEL, FF, LS, PS
- ML Match Limit. Possible values: an integer. 0=no limit, <0=only match |ML
- Greedy Whether matches are the longest or shortest. Possible values: 1 or 0

Examples

1. Change each vowel into XX

```
( '[AEIOU]' ¶R 'XX' ¶ 'IC' 1) 'ABCDE abcde'
XXBCDXX XXbcdXX
```

Explanation: the pattern specifies a set of letters to match, 5 vowels, each to be replaced by 'XX'. Normally only the 'A' and the 'E' would be replaced but because the Variant operator has modified the `R` resulting function to be Case Insensitive (with `¶ 'IC' 1`) then 'a' and 'e' are also modified.

2. Change 'C' followed by any character then 'D' by dash (-) in a multiline text

```
( 'C.D' □R '-' □ ('Mode' 'D') ('DotAll' 1)) 'ABC',CR,'DEF',CR,'CAD'
AB-EF
-
```

Explanation: the dot in the pattern usually means ‘any character but EOL’ but because Variant specifies that the *dot* matches *all* characters and because it also specifies ‘Document mode’ (Mixed mode would do too) then the sequence ‘C,CR,D’ matches and it is replaced by ‘-’. The sequence ‘CAD’ also matches and is replaced accordingly. Note how multiple options are written. Because Variant modifies an existing function it can be applied many times. The following is all equivalent:

```
( 'C.D' □R '-' □ ('Mode' 'M') ('DotAll' 1))...
( 'C.D' □R '-' □ 'Mode' 'M' □ 'DotAll' 1)...
CXD←'C.D' □R '-' ◇ (CXD □ ('Mode' 'M') ('DotAll' 1))...
(CXD □ 'DotAll' 1 □ 'Mode' 'M')...
CXM← CXD □ 'Mode' 'M' ◇ (CXM □ 'DotAll' 1)...
```

3. Change the first 2 letters of each line by ‘x’

```
( '[A-Z]' □R 'x' □ 'ML' 2) 'ABC' 'DEF'
xxC xxF
( '[A-Z]' □R 'x' □ 'ML' -2) 'ABC' 'DEF'
AxC Dx F
( '[A-Z]' □R 'x' □ 'ML' -4 □ 'Mode' 'D') 'ABC' 'DEF'
ABC xEF
```

Explanation : in the first case we ask to change any character in the set A-Z by x. Because we are working in line mode, each vector is processed independently and the first 2 letters of each line is modified. In the second case we ask that only the *second* (-2) match be acted upon, again in each line. In the last case we ask that only the 4th match be processed. Because we now treat the argument as a whole document ('Mode' 'D'), only the 4th letter, ‘D’ is modified.

Using files as input source

It is possible to use a native file as source instead of a array (character vector or VTV).

To do so you supply the tie number of the file to process. If the file is read from the start, and there is a valid Byte Order Mark (BOM) at the start of it, the data encoding is determined by this BOM.

The input document may be processed in any mode. In document mode and mixed mode, the entire input document, line ending characters included, is passed to the search engine; in line mode the document is split on line endings

and passed to the search engine in sections without the line ending characters. The choice of mode affects both memory usage and behaviour.

Solved patterns

Find a number from 0 to 255: `\b(25[0-5]|2[0-4]\d|[01]?\d\d?)\b`

Find an IP address:

`\b((25[0-5]|2[0-4]\d|[01]?\d\d?)\.){3}(25[0-5]|2[0-4]\d|[01]?\d\d?)\b`

Finding a date:

`\b(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|1[12][0-9]|3[01])\b`

Finding APL names or numbers

It may be interesting to know where some names are in some code.

Finding a name with a regex in a conventional language is fairly easy: `[A-Za-z]\w*` will usually do it: that's 1 letter followed by 0 or more word characters (the * means 0 or more times). Not in APL. APL names in Dyalog can have all sorts of characters like `Δ` and `Δ`. A regular expression to find such a name would look like

```
(?x) # ignore spaces and comments
# define what the Leading character of a name and what an APL ID is
(? (DEFINE) (?<L1>[_A-ZΔa-
zΔÁÂÃÇÈÉÊËÌÍÎÏÐÓÔÕÖÙÚÛÜÝÞàìðòöÀĂÄÅÆÉÑÖÜßàáâãäåæçèéêëíîïñóôõöùúü])) )
(?:
    # do not capture
    (?<!\s:|::) # not preceded by space: or ::
    (?<!(?&L1)) # not preceded by any of the characters above
    (?&L1)      # must start with one of them
    (?: (?&L1)|\d)* # followed by 1 of them or a digit 0 or more times
    |α|α|ωω|ω)   # or any of those
```

which is a bit hard to remember and type – and to explain in details here, even with the comments.

Assuming we put the above pattern in variable 'patid' we can find APL names in code like this:

```
patid ⍵S {ω.Offsets,ω.Lengths} '⍵←dsÀÄa←_-.=0 uΔio8 321'
2 5 8 1 14 5
```

Numbers are similar. A number in Dyalog would require a long expression to be used.

There is a project at Dyalog to provide a shortcut for such patterns. For now if you wish to find general APL names or numbers in your code you have to resort to user command `⌈WSLOCATE`. Type `⌈?WSLOCATE` for details.

Technicalities

The engine needs to prepare the search before using it.

This involves parsing the pattern and compiling a structure subsequently used in the process. If a search will be performed repeatedly it would be inefficient to recompile it each time. The interpreter tries to account for this by keeping the last few structures around for a while in case they are to be reused. The interpreter keeps a cache of compiled patterns and flushes the one least recently used when the cache is full and a new pattern is created. This cache is saved in the workspace; and is never garbage collected.

Real life example

This example is for those with a good understanding of regular expression.

In 2011 I needed to look into a log file and extract some data in a log file tied to 'tieno'. I was looking for the patterns

P1: `'(type=\d+, shape=(?:\d+[,])+)'` (here I wanted the whole thing)

Followed by anything and then by

P2: `'\p2 0\B *\d+(\d+)'` (here I only wanted the number in the parentheses)

This is basically the pattern `P1.*?P2`.

By doing `'P1.*?P2.' ⌈S info tieno` I should have been able to extract all the data with `info` a function catenating P1 with P2 results for each match.

Unfortunately the file sometimes contained `P1...P1...P2` which means that the 1st P1 (and not the second) was matched with P2. So I had to use a different pattern.

I used `p1((?!p1|p2).)*p2`

This is "look for p1, advance while you don't have either p1 or p2, then look for p2".

Explanation:

Let's use the following text as an example:

aap1bbp1ccp2dd

The 'p1' at the start of the pattern will cause the text pointer to skip over the "aa" characters and match the first "p1" characters.

The next part in parentheses breaks down to a negative lookahead (the ?! part) and the '.'. The negative lookahead will check the next character(s) (here "bb") and, since there is no match either with p1 or p2 (the 2 alternatives), will succeed and so the '.' will match the first "b". This is repeated with the "bp" characters.

As the next characters are "p1" the lookahead will fail (because of its negative aspect). Because the quantifier that controls this part of the pattern is '*' (0 or more matches) the fact that the lookahead failed is not a problem, it's only the parentheses group that stops matching. The next thing in the pattern is p2 which is compared with the "p1" in the text - which will fail. This global failure causes the regex engine to start backtracking. In this case, there are no alternative ways that they can match so the whole pattern match is rejected. 'p1bbp1' failed to match.

When this occurs, the regex engine will step forward 1 character over the 1st "p" and start the match at the "1bb" character sequence. In this case the regex engine will again start skipping forward until it gets to the second "p1". The process described above will be repeated except that the lookahead will stop the '.' matching when it gets to the "p2" characters. Now, the last part of the pattern (p2) does match which lets the regex engine declare that a complete match has been found.

The sequence 'p1ccp2' matches, info grabs it, extract the sections I want, merges and returns them. I'm one happy camper.

This last example is a real life example which would have been more difficult to program in APL (or any other language for that matter). The ability to use regexes combined with APL allowed me to perform the task in a fraction of the time it would have taken me normally.

Conclusion

Regular expressions are extremely useful when dealing with complicated patterns. Without them you must spend time writing code, sometimes a lot of, to parse and extract, time that could be better spent, well, any other way. Up until now there was no choice, but now there is.

You have to be careful and not overdo it. Using code like `txt←'x' ⍱R 'X' +text` is very inefficient compared to `((text='x')/text) ← 'X'`

Caveat emptor.

Books : Regular Expressions Cookbook (O'Reilly 2009)

References

1. Another one is \ln to Lowercase match of sub pattern 'n'
2. Spaces are important in regular expressions but ignore them here
3. Modes are explained below
4. For example used with iota (ι) it could generate a function to generate series from either 0 or 1. By using $\iota \neq 0$ we could create a function that generates numbers starting at 0 regardless of $\neq IO$.
5. Astute observers will wonder what Classic users do. For them there is an equivalent $\neq OPT$ system operator.

Bayesian financial dynamic linear modelling in APL

Devon McCormick (devon@acm.org)

Editor's note: this article was first published in *Vector* 21.2 Spring 2005. It is being reprinted as an article that contains valuable material that has stood the test of time.

Bayesian statistics is a brand-new idea that's only about 235 years old. The paper that was to immortalize the last name of the Reverend Thomas Bayes, a Nonconformist minister born in 1702, was published in 1763. Unfortunately for any fame he may have hoped to gain from what proved to be his most influential work, Bayes died some 3 years before.

From such inauspicious origins, his contribution to statistics continued inauspiciously. His ideas flourished briefly in the latter part of the 18th century, being taken up by the great mathematician Laplace, before languishing in relative obscurity until this century. Even though revived in the early part of the century, first by Ramsey, then by deFinetti, his ideas only became widespread in recent years.

So, what is this great idea that has come to us through centuries? At first glance, it may appear to be rather trivial. So trivial, in fact, we may briefly derive it here without further exposition.

Bayes examined the problem of contingent probability. We may start, simply enough, by noting that the probability of an event P , denoted here as $\text{Prob}(P)$, contingent on circumstances (or Data) D , is the product of $\text{Prob}(P)$ and $\text{Prob}(D|P)$. This latter formulation may be read *the probability of D given P* . So, since

$$\text{Prob}(PD) = \text{Prob}(P) \times \text{Prob}(D|P)$$

and, the reverse being true and equivalent, that is

$$\text{Prob}(DP) = \text{Prob}(D) \times \text{Prob}(P|D)$$

Taking these two as equal to each other leads us to the common formulation of Bayes's Theorem:

$$\text{Prob}(P|D) = \frac{\text{Prob}(D|P) \times \text{Prob}(P)}{\text{Prob}(D)}$$

There might not seem to be much here, so why should this simple theorem have provoked so much contention and fallen into such conspicuous disfavour? The key here is the interpretation of the prior Data: it includes any information we see fit to include: this makes it subjective, hence often thought to be not fit for proper scientific and mathematical treatment. That is, the powerful effect of Bayes's Theorem is to allow us to incorporate non-traditional, perhaps inexact, data into our probability formulations.

This denigration of subjectivity, in a historical context of pride in exact, hard science, continues to plague Bayesian techniques even up until recent times. In fact, one reference (in Kuhn's *Readings in Game Theory*) to Bayesian techniques appears to justify calling it Bayesian only because of the general use of subjective probability estimations. One much-cited early work about Bayesian techniques is called *Studies in Subjective Probability* (Kyburg, 1964): at the time it was published, this was the defining feature of Bayesian statistics.

However, as we shall see, this perceived subjectivity of the technique is misconstrued as a weakness, when it is rather a strength because it acknowledges and deals with the outside information underlying all statistical inference. This combining of hard and soft data is helpful in the context of the Dynamic Linear Models we will explore later on.

Another strength of Bayesian inference has to do with its approach to probability, an approach often called *the inverse probability problem* because it is backwards in relation to traditional probability. This problem is stated thusly:

Given the number of times an event with unknown likelihood has occurred or failed to occur, what is the chance that the probability of it happening in a single trial lies somewhere between two degrees of probability?

One of the Bayesians' favourite example of this is the urn problem: given an urn containing two colours of balls, say black and white, and a series of tests whereby we draw a ball, note its colour and return it to the urn, what is the likelihood of various combinations of the two colours? For instance, given an urn with five balls and the evidence that three draws with replacement yield only white balls, what is the probability that all five are white?

This is the inverse of a typical problem in probability that would be more along the lines of: given an urn with three white and two black balls, what are the odds of drawing three white balls, one at a time with replacement? Before we delve further into the Bayesian example, notice how more commonly applicable to typical finance problems is this former sort of formulation than is the latter. A classical probability problem would be something like: given the mean returns and volatility of some assets, how are they likely to perform? However, the inverse problem would be something like: given the performance of these assets, what are their likely returns and volatilities? We can see from this that the inverse problem better matches what we often have to work with in terms of real data.

Before we look more at the urn problem, we need to extend Bayes's Theorem to multiple events. Given a series of k events P_i with associated likelihoods $\text{Prob}(P_i)$ and prior observed data D , the contingent probability of a particular event P_i given observations D is stated thusly:

$$\text{Prob}(P_i|D) = \frac{\text{Prob}(P_i) \times \text{Prob}(D|P_i)}{(\text{Prob}(D|P_1) \times \text{Prob}(P_1)) + \dots (\text{Prob}(D|P_k) \times \text{Prob}(P_k))}$$

In APL, this is shown by the function `BayesPP`.

```

▽ R←BayesPP PROBS
[1]  A Calc Bayesian posterior probability given PROBS: 2 row mat:
[2]  A [0;]P(P1), P(P2)...; [1;]P(P1|D), P(P2|D)...; i.e. [0;] isolated
[3]  A event probability, [1;] conditional probability given data D.
[4]  R←R÷+/R←×/PROBS
▽
```

In practice, this works as follows. Suppose we have an urn with 5 black and white balls in unknown proportion. We draw 3, one at a time with replacement; all are white. What is the probability that all 5 are white? Invoking the APL function

```
UrnProbAllWhite 5 3
```

gives us the numerical answer of 0.15625. How do we work this out using the above theorem? We must solve for the probability of all the balls being white given the evidence of 3 white draws ($\text{Prob}(P_i|D)$). The text of this function and a discussion follows.

```

▽ P←UrnProbAllWhite NNW;IO
[1]  A Bayesian urn problem: given NNW[0] (ostensibly black and
white) balls
[2]  A and experimental evidence that NNW[1] white ones picked,
with
```

```

[3]      A replacement each time, what is probability that all are
white?
[4]      ⍵IO←0
[5]      P←((⍵NNW[0]+1)÷NNW[0])*NNW[1]          A 0 to N possible balls
of color
[6]      A picked ... chance of picking NW white ones for each
proportion possible.
[7]      P←+/P×(BinomialCoeff NNW[0])÷2*NNW[0] A 2* because 2 possible
colours;
[8]      A would have to use other than binomial expansion if more
than 2.
[9]      A P is now all probabilities of all combinations. Applying
Bayes's
[10]     A theorem: (Prob(all selections white given all balls in urn
are white ×
[11]     A Prob(all balls in urn are white)) ÷ Sum of Prob(all
combos)
[12]     P←(((÷/2pNNW[0])*3)×÷2*NNW[0])÷P

```

▽

The subfunction BinomialCoeff is defined thusly:

```

▽ R←BinomialCoeff N;⍵IO
[1]     A Give coefficients of Nth binomial expansion.
[2]     ⍵IO←0 ⋄ R←((⍵N)!N),1
▽

```

Working 1st on the divisor of the preceding equation, calculated in line 5 of the APL function, we calculate the probability of the datum D (3 white samples) for each of the possible Pis. These latter range from no white balls (all black) to 5 white balls, or the fractions 0/5, 1/5, 2/5, 3/5, 4/5, and 5/5. These are all the Prob(Pi)s.

Line 7 of the APL function figures the likelihood of 3 white draws given each of the possible combinations of white and black balls. The denominator of each of these is 2*5 or 32 because this is the number of possible combinations of 2 things taken 5 at a time. The numerator is the possible arrangements of each combination, e.g. 1, 5, 10, 10, 5, and 1 corresponding to: 1 way to have all 5 black, 5 ways to have 1 white and 4 black, 10 ways to have 2 white and 3 black, 10 ways to have 3 white and 2 black, 5 ways to have 4 white and 1 black, and 1 way to have all white. Combining all the Prob(Pi)s with the Prob(D|Pi)s, multiplying them together and summing the results gives us (in math):

$$\left[\left(\frac{0}{5} \right)^3 \times \left(\frac{1}{32} \right) \right] + \left[\left(\frac{1}{5} \right)^3 \times \left(\frac{5}{32} \right) \right] + \left[\left(\frac{2}{5} \right)^3 \times \left(\frac{10}{32} \right) \right] + \left[\left(\frac{3}{5} \right)^3 \times \left(\frac{10}{32} \right) \right] + \left[\left(\frac{4}{5} \right)^3 \times \left(\frac{5}{32} \right) \right] + \left[\left(\frac{5}{5} \right)^3 \times \left(\frac{1}{32} \right) \right]$$

or (in APL, origin 0):

```
(( (16)÷5)*3)+.×1 2 10 10 5 1÷32
```

which equals 0.2. The difference between these 2 expressions prompts me to question the reader: which of these 2 expressions looks simpler? Which do you think took about 5 minutes to enter and which took about 5 seconds?

The last line of the APL function calculates the numerator in Bayes's Theorem above, which is the probability of 3 white draws given all 5 balls being white, and divides it by the denominator to give our answer. A more general version of this is the APL function `UrnProblem` that returns all probabilities instead of just the all-white one. An even more interesting function would allow a third input of the number of black balls in a sample instead of just restricting the possible data to observations of all-white draws. This is left as an exercise for the reader. The text of `UrnProblem` follows.

```
▽ P←UrnProblem NNW;⊂IO;NB;NWhite;Pk;Num;Denom
[1]  A Bayesian urn problem: given NB (ostensibly black and white)
balls
[2]  A and experimental evidence that NWhite white ones picked,
with
[3]  A replacement each time, what are probabilities that 0 to NB
are white?
[4]  ⊂IO←0 ∘ NB NWhite←NNW
[5]  Pk←((1NB+1)÷NB)*NWhite           A 0 to N possible
balls of color
[6]  A picked ... chance of picking NW white ones for each proportion
possible.
[7]  Denom←+/Num←Pk×(BinomialCoeff NB)÷2*NB A 2* because 2
possible colors;
[8]  A binomial expansion assumes prior normal distribution for 2
colors.
[9]  A Apply Bayes's theorem:
[10] P←Num÷Denom
▽
```

One point in the above exposition that deserves further mention is our gloss on the use of binomial coefficients (`UrnProblem[7]`). This introduces a primary facet of Bayesian inference: the use of a prior distribution. By using the binomial coefficients to weight the black and white combinations, we are in fact assuming a prior distribution that approximates the normal distribution (for discrete values). The effect of this prior distribution may be seen in the values generated by

```
UrnProblem 6 1
0 0.03125 0.15625 0.3125 0.3125 0.15625 0.03125
```

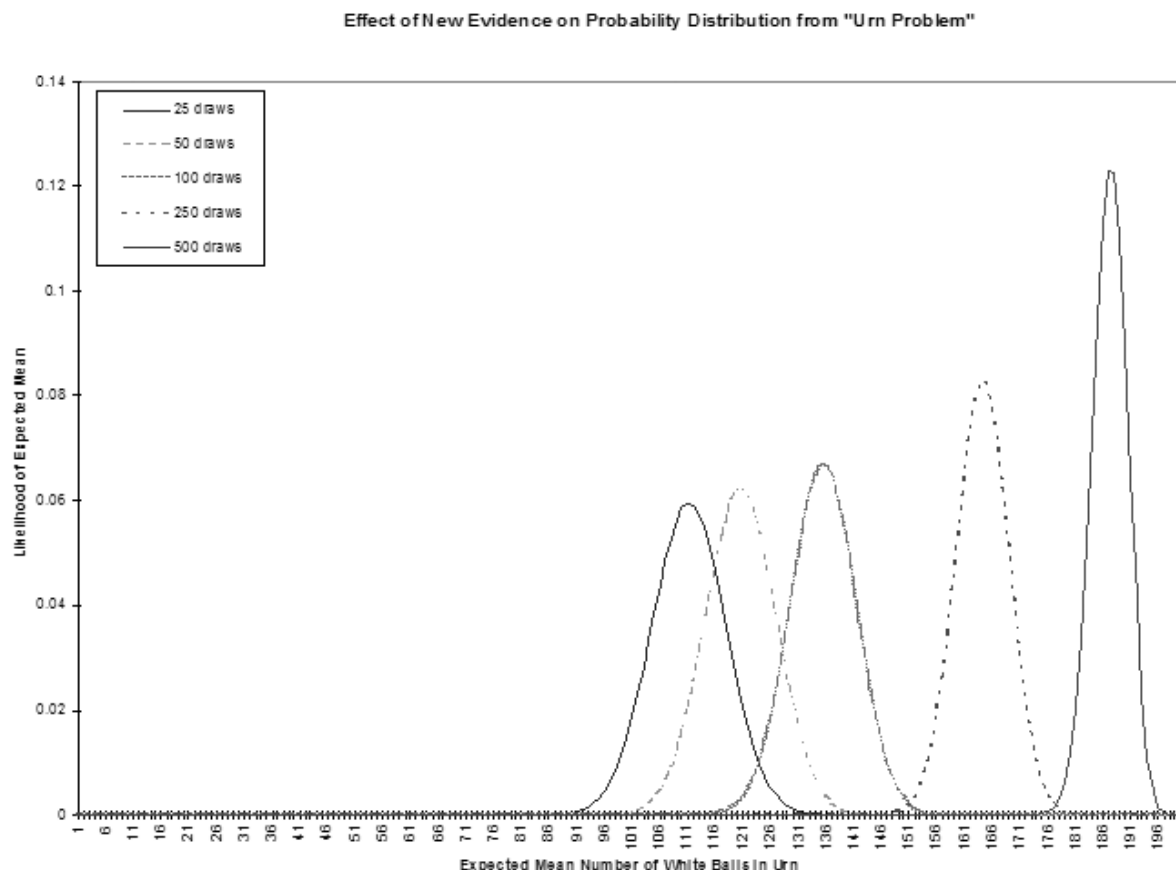
versus the result of


```

UrnProblem 6 6
0 0.0000275999 0.00441599 0.0670678 0.282623 0.431249 0.214617

```

Notice how, in the first case (for 1 draw from an urn containing 6 balls), the peak values are the 2 in the center whereas, in the second case, the peak has moved to the second-to-last value. To better see this, look at the *Evidence Effect* graph: this shows the results of `UrnProblem` run for an urn containing 200 balls and 25, 50, 100, 250 and 500 draws of white balls. Notice how the distribution shifts in the manner hinted at by our small example above. What does this mean?



In the first case, our observation of 25 white draws for 200 balls, combined with our prior (assumed) normal distribution leads us to put the most likely combinations near the centre. This shows the strong influence of the prior distribution with little additional evidence. The subsequent cases, each with more white draws, move the peak toward the all-white end of likely distributions. Of course, evidence like this might lead us to question our choice of prior distribution. In any case, we can see how this Bayesian technique allows us to combine our estimate of a distribution with subsequent evidence about that distribution.

While we're on the subject of prior distributions, it's worth noting that the traditional standpoint of statistics, from which the "subjectivity" of Bayesian

statistics has been attacked, is in fact explainable in Bayesian terms as one with a “flat” prior. That is, if we decide to study the frequency of occurrence of heads and tails when flipping a “fair” coin, we can either assume, as traditionalists do, that we have no information about the likely outcomes (that is, all probabilities are equal, hence the prior distribution is a straight line), or we might acknowledge that this assumption is just one of many possible prior distributions. Under the traditional assumption, after 1 trial of flipping the coin, we would have to say that the likelihood of heads is 100% (or 0%, depending on which turned up).

This influence of the prior distribution suggests a further modification to the `UrnProblem` code: we could supply a prior distribution function as one of its arguments. This change, along with the one suggested earlier that allows for more generality in observations, would give us a version of the function that starts to appear to have more practical application than the small samples of Bayesian inference we’ve seen so far. However, we will leave this pursuit to interested readers. Instead, we will now look at the other important technique that, combined with Bayesian inference, can give us a robust financial forecasting system: Dynamic Linear Models.

Dynamic Linear Models

The following pair of equations characterizes a dynamic linear model:

Type	Equation	Distribution Mean, Variance
Observation	$Y_t = F_t' \theta_t + v_t$	$v_t \sim N[0, V_t]$
System	$\theta_t = G\theta_{t-1} + w_t$	$w_t \sim N[0, W_t]$

The notation in the column labelled *Distribution Mean, Variance* gives the distribution of the error term for each equation; the expression $N[m,v]$ specifies a normal distribution with mean m and variance v .

The variables in these equations have the following meanings:

Variable	Description
Y_t	vector of dependent variables (e.g. asset returns)
F_t'	(transposed) matrix of explanatory factors
Θ_t	regression factors
v_t	observation errors
G	state evolution matrix
Θ_{t-1}	previous time's Θ
W_t	system errors

The ultimate aim of a Dynamic Linear Model (DLM) is to estimate the Y values in terms of a likely mean and variance. We may think of the equation that purports to explain these Y values as a linear equation with variable coefficients applied to time-varying explanatory factors. For instance, a model of bond prices might state the observation equation using factors known to influence bond prices, such as government interest rates and the spreads of corporate rates.

The system equation may be thought of as a system of relations between the observed factors. However, the dynamism of the model derives from assuming that these relations vary.

Bayesian Forecasting with Dynamic Models

The preceding system of equations is updated by successively examining predictive factors with their corresponding returns, predicting the mean and variance of expected returns based on the next set of factors, and modifying these predictions according to the actual returns. We can see how this is analogous to our earlier, simpler problem of estimating the unknown composition of balls in an urn based on the evidence of samples from the urn. In the case of the urn, we saw how the probability distribution moved from the initial assumption of normally distributed with an equal mix of black and white to a higher likelihood of all white balls based on the evidence of repeated samples of white balls. Similarly, our forecasting model starts with assumptions about means and variances of returns then modifies these assumptions based on successive samples of correlations between econometric factors and asset returns.

The West and Harrison book (Bayesian Forecasting and Dynamic Linear Models, Springer-Verlag, 1997) provides a series of increasingly complex sets of data to model in chapter 3. Three of these examples are implemented in APL in the functions `Theorem3p`, `Theorem3p1_1` and `Theorem3p1_2`. The associated data sets are contained in the variables `TBL3P1`, `TBL3P2`, and `TBL3P3`, respectively. These may be of interest for use with the book. However, we will look more at a

multivariate version of the model that more closely parallels, though it is somewhat simplified, the one in use at my company for asset allocation.

The model on which we will concentrate is the one embodied in the function `MultiVDLM3`. Before we look at this, though, a word about the data used in this example. To avoid problems with proprietary or purchased data, we'll be using generated datasets. These are outputs from the `GENFACRETS` function. Hence, a brief look at this function is in order.

```

▽ FR←GENFACRETS NUM;IO;MAX;FACS;RETS
[1]  A Generate a set of factors and returns associated with them.
[2]  IO←0 ♦ FR←(0 4p0)(0p0) ♦ MAX←1000000000
[3]  A FACS←--\1 2 1 2o1 2 3 4÷5 6 7 8
[4]  FACS←(5+?4p100)|1+?4p1000000000
[5]  RETS←--/FACS
[6]
[7]  L_NEXT:FACS←1 2 1 2o0.5 1 1.5 2×FACS+2 3 4 5
[8]  FACS←FACS+0.01×+/-5+?((pFACS),20)p11 A Handful of little
normal noise
[9]  RETS←--/FACS
[10] RETS←RETS+0.01×+/-5+?((pRETS),20)p11 A Handful of little
normal noise
[11] →(NUM>1pp>FR←FR,[0]"FACS RETS)pL_NEXT
▽

```

Notice how the factors are generated using a “hidden” function and the returns are based on these factors. A typical use of this function might look like this:

```
Factors Responses←GENFACRETS 120
```

to assign the 2 vectors `Factors` and `Responses`, each with 120 elements as specified by the function's right argument.

The accompanying illustrations of forecasts compared to actual data use these generated data as inputs to the multivariate dynamic linear model. Looking now at the primary instance of this model, the APL function `MultiVDLM3`, there are a few important points to note. A full explanation of this model is beyond the scope of this paper; however, one of the books by West and Harrison or Pole, West, and Harrison, should provide sufficient detail (see the bibliography). Bearing this in mind, let us examine the few points on which we will concentrate to illuminate the robustness of this type of model. The text of the function follows (next page).

```

▽ DMO←MultiVDLM3
FGVWYMCD;IO;Wt;Vt;MC;SZ;CT;mp;Cp;F;Y;G;Ft;Yt;Rt;ft;Qt;

At;at;et;mt;Ct;DELTA;nt;np;St;Sp;TIT
[1]  A Multivariate Dynamic Linear Model, adapted from West and Harrison.
[2]  A FGVWYMCD: [0] Factors; [1] G (evolution matrix); [2] V (observation
[3]  A variance); [3] W (system variance); [4] responses (Y values to
[4]  A predict); [5] MC: prior mean and variance; [6] (optional) delta

```

```

[5]  A (discount factor). E.G. for 3 factors predicting 1 return:
[6]  A MultiVdLM3 (10 3p130) (3 3p4t1) (30 1p2x130) (1 1p1) (3 3p0.01)
[7]  A (15 9) (0.6)
[8]  A OR DMO+MultiVdLM3 F ((2p-1tpF)p(1+1tpF)t1) (,COVM R) (COVM F)
[9]  A (((pR),1)pR) (( AVG R) ( STDDEV R)) (1)
[10] A QDMO[1;]: [;0] Forecast mean and [;1] variance, [;2] posterior system
[11] A (theta) mean and [;4] variance, [;5] adaptive coefficient, [;6] error,
[12] A [;7] Scaling factor.
[13]
[14] CT←IO←0 ♦ TIT←'ft' 'Qt' 'mt' 'Ct' 'At' 'et' 'St'
[15] F G Vt Wt Y MC DELTA←7tFGVWYMCD,1 ♦ SZ←1ppF ♦ DELTA←DELTA*0.5
[16] DMO←(pTIT)p←0 1p0 ♦ np←Sp+1
[17] mp Cp←(cpWt)pMC A Prior mean and variance: coerce shapes if scalar
[18] at←mp
[19]
[20] L_DO1:Ft←QF[,CT;] ♦ Yt←Y[,CT;] A 1 row mats for conformability
[21] at←G+.xmp
[22] Wt←(DELTA×G+.xCp+.xQG×DELTA)-G+.xCp+.xQG A Discounting
[23] Rt←(G+.xCp+.xQG)+Wt A Prior at time t: theta variance
[24] ft←(QFt)+.xat A 1-step forecast ←means
[25] Qt←Sp+(QFt)+.xRt+.xFt A and ←variance for forecast Y
[26] et←Yt-ft A 1-step error forecast
[27] nt←np+1 A Start on posteriors
[28] At←Rt+.xFt+.xQQt A Adaptive coefficients,
[29] mt←at+At+.xet A Posterior thetas' means
[30] Ct←Rt-(QAt)+.xAt+.xQt A and (scale-free) variances
[31] St←'pSp+(((Qet)+.x(QQt)+.xet)-1)×Sp÷nt A Scaling factor (p. 110 (d))
[32] Ct←(St÷Sp)×Ct
[33] DMO←DMO,[0]''Q''ft Qt mt Ct(QAt)et St
[34] Cp mp Sp np←Ct mt St nt A Current to previous for next loop
[35] →(SZ>CT+CT+1)pL_DO1 ♦ DMO←TIT,[-0.5]DMO

```

▽

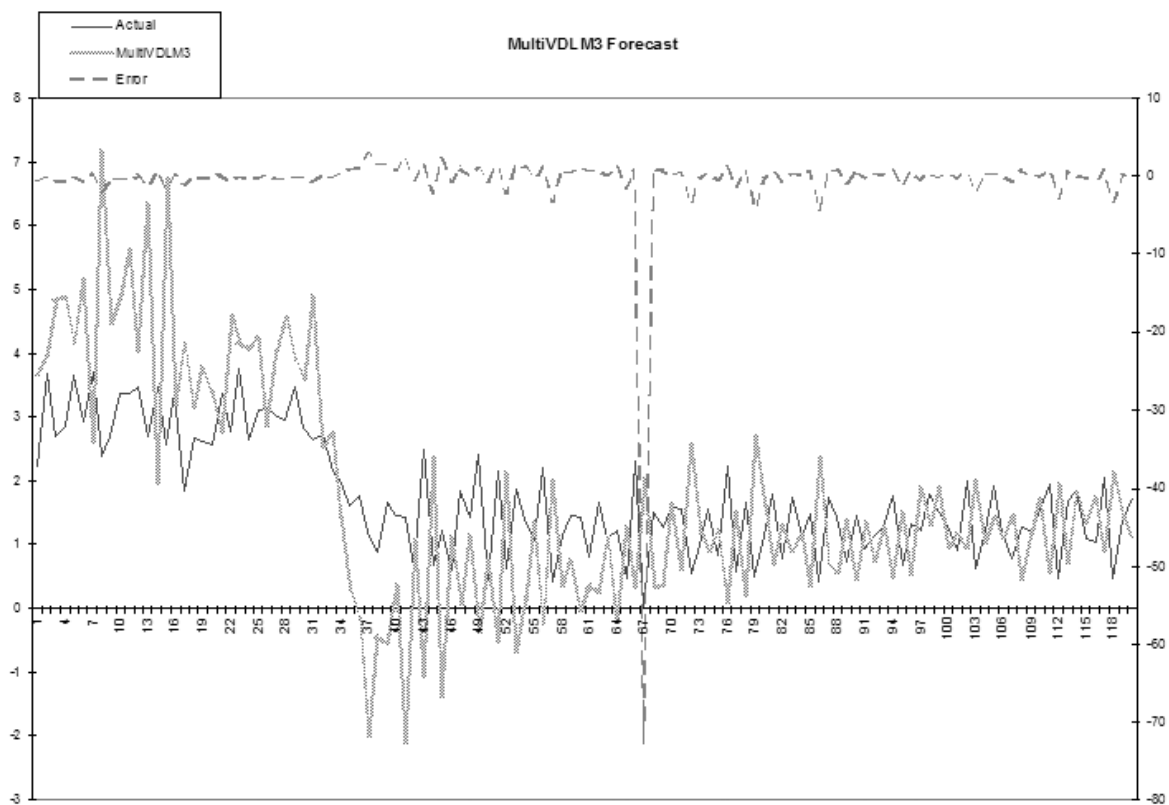
First, let's dispose of variables set to rather trivial values for this example: G and $DELTA$. We set G to be the identity matrix: this corresponds to a random system evolution. We set $DELTA$ to be 1; this might have a value slightly less than one to more greatly discount the effects of data, which is further in the past. In addition to these 2 settings, made uninteresting for simplicity, we also take a simple default on the prior mean and variance setting (`FGVWYMCD[5]` in the function) by using the actual mean and variance of the return series. Disposing of all these leaves us to consider only the factors and returns, which we discussed above, and the 2 variances W and V , for the system and observation series.

How to deal with these 2 inputs remains a subject of some research. We'll look mainly at V , the observation variance, to illustrate the robustness of DLMS and how well the Bayesian framework allows us to incorporate new prior information.

Many of the models (in West and Harrison, 1997) assume that this observation variance is known. In fact, it is tempting to use the same shortcut we used for the prior mean and variance values by essentially looking into the future and basing the value on our present data. However, this detracts from the ability of the model to adapt to new information by reinforcing the values of past data. However, notice that this is essentially what we did in our example just to get

some crude numbers to look at. The important thing to notice is that the system of equations given above specifies v as varying according to time t but our APL implementation treats it as a known constant.

This limitation of our sample implementation detracts from its forecasting power. A better model would attempt to forecast the variance series as it does the return series and progressively correct these forecasts with the receipt of new data. Indeed, the flexibility of this sort of model allows us to do even better than this by incorporating anticipated future data. Since we do not have such an implementation in our simple example, let's see how and why we might do this. First, however, let's look at an example of the function's forecast on some data "made up" by the GENFACRETS function as shown above. These results are shown graphically in the following chart.



Notice how the earlier attempts at forecasting, on the left, diverge widely because the model hasn't "learned" much about the relation between the factors and their associated responses. However, it does track the change in level starting at about points 31 to 43, even though it initially overshoots. Suppose that this change in level were in response to events we could anticipate to some extent: we know that the fundamentals affecting our responses are about to change but we can't quantify them exactly.

Say, for example, that one of the assets whose return and variance we are forecasting is the Hong Kong dollar. Since the value of the Hong Kong dollar has been fixed to that of the U.S. dollar for many years, our model will build a misleading picture of this currency's behaviour based on historical data. This is a very pertinent issue at the time I write this, since that colony has recently reverted to Chinese rule, Asian markets in general have become more volatile of late, and there are indications that the end of this fixed rate regime is close at hand. According to an article in *Barron's* (the column "The striking price", November 17, 1997), the American stock exchange "plans to revise its Hong Kong Option Index so that its value is calculated using a floating rate of exchange for the Hong Kong dollar rather than the fixed value that's now used."

These factors make it very likely that a Hong Kong dollar model will change behaviour drastically in the near future. In addition, this sort of deliberate change will usually be announced in advance, thus aiding the modeller using Bayesian DLMS. A more complete model than the one presented here would use a true time-varying variance and would implement it in such a way as to incorporate an external multiplier. Though one might hazard a guess as to the direction of the Hong Kong dollar when it begins to float, one is not obligated to.

Though a fuller system than the one outlined here might also accommodate external inputs to the calculations of means as well as variances, this would require a view on the market that we might not be prepared to risk. However, it does not require much of a leap of faith to predict that the volatility of the Hong Kong currency will increase relative to the U.S. dollar some time soon.

A Bayesian DLM naturally and easily incorporates such new information, even when it is rather vague and subjective as such advance information often is. We are required to assign some value to our variance multiplier, but, in doing so, we can use whatever other information we see fit (and in this example there is a strong argument for a certain lower limit.) This ability to seamlessly incorporate judgment based on new developments makes these kind of models very attractive, especially since it minimizes or eliminates re-programming and allows vague information to be introduced in a well-controlled manner.

This document is for informational purposes only. Opinions expressed are our present opinions only, and are subject to change. In preparing this presentation, we have obtained information from sources we believe to be reliable, but do not offer any guarantees as to its accuracy or completeness.

The “88 Hats” puzzle

by Roger Hui (RogerHui.Canada@gmail.com)

The 88 Hats puzzle was posed by Andrew Nikitin[1] to the J Forum on 5 Jun 2007.
The following is a slightly modified version.

The puzzle

88 people stand in a circle, each having a hat with a number from 0 to 87 written on it. Everyone can see the numbers on other people's hats but can not see his own number. They simultaneously write a number on a piece of paper and give it to the judge. If at least one of them wrote a number that is on his own hat then everyone wins, otherwise everyone loses. What strategy should they use to guarantee victory?

(Numbers on the hats do not have to be all different. People can not exchange any information during the procedure but can agree on some strategy beforehand.)

The puzzle was solved by John Randall[2] a day later.

A winning strategy

The strategy works for any number of hats m . Beforehand, the people number themselves from 0 to $m-1$. Let h be the list of hat numbers, and $s[n]$ be the sum of the numbers that person n sees. Then $w[n]$, the number that person n writes, is $m | n - s[n]$.

```

m←13
random←{⌊rl←7*5 ♦ ?ω}  A reproducible random numbers

⊢ h←random mpm
1 9 5 6 2 0 8 8 12 4 6 10 0

⊢ s←(∘.≠~ιm)+.×h
70 62 66 65 69 71 63 63 59 67 65 61 71

⊢ w←m|(ιm)-s
8 4 1 3 0 12 8 9 1 7 10 2 6

w ,[-0.5] h
8 4 1 3 0 12 8 9 1 7 10 2 6 1 9 5 6 2 0 8 8 12 4 6 10 0

+ / w=h
1

```


Here's why it works. Let $t \leftarrow m|+ / h$ be the sum of all the hat numbers, modulo m . The following are equal:

t	
$m s[t]+t-s[t]$	A addition and subtraction
$m s[t]+m t-s[t]$	A modulo arithmetic
$m s[t]+w[t]$	A definition of w

Moreover, since $s[t]$ is the sum of all hats excluding $h[t]$, it must be that $t = m|s[t]+h[t]$. Therefore $m|s[t]+w[t]$ and $m|s[t]+h[t]$ are equal and so $w[t]$ and $h[t]$ are equal. That is, the written number $w[t]$ for person t matches his hat number $h[t]$.

$t \leftarrow m + / h$
6
$w[6]$
8
$h[6]$
8

It is instructive to examine what the winning strategy does for 2 hats. The strategy calls for person n to write $m|n-s[n]$. When $m \leftarrow 2$, this reduces to person 0 writing the hat number of person 1 and person 1 writing the negation of the hat number of person 0. The tabulation of all possible hat numberings and corresponding writings shows that in each scenario there is a written number that equals a hat number.

numbering	writing
0 0	0 1
0 1	1 1
1 0	0 0
1 1	1 0

An abstraction

Let G be an abelian group of size m with operation g and whose group members are items of u . Compute $s \leftarrow \{g/(\omega \neq \imath m)/u[h]\}^{\imath m}$; $s[n]$ is the "sum" of hat numbers that person n sees. Then the written number $w[n]$ is $u \imath u[n] g g \imath s[n]$ where $g \imath x$ is the inverse of x in the group. (In Section 1 G is the group of integers under addition modulo m ; the group elements u are $\imath m$.)

Why does it work? Let $t \leftarrow u \downarrow g / u[h]$ be the index of the “sum” of all the hat numbers. The following are equal:

$u[t]$
 $u[t] \ g \ s[t] \ g \ g_i \ s[t]$ A group inverse and identity
 $s[t] \ g \ u[t] \ g \ g_i \ s[t]$ A associative and commutative
 $s[t] \ g \ u[w[t]]$ A definition of w

Moreover, since $s[t]$ is the sum of all hats excluding $u[h[t]]$ and the group is abelian, it follows that $u[t] = s[t] \ g \ u[h[t]]$. Therefore $s[t] \ g \ u[w[t]]$ and $s[t] \ g \ u[h[t]]$ are equal, and so $u[w[t]]$ and $u[h[t]]$ are equal and consequently $w[t]$ and $h[t]$ are equal. That is, the written number $w[t]$ for person t matches his hat number $h[t]$.

88 hats

The original puzzle has $m \leftarrow 88$. Since

$I \leftarrow \{\omega / \downarrow \rho \omega\}$
 $I \ 88 = \{+ / 1 = \omega \vee \downarrow \omega\} \cdot \downarrow 1000$
 89 115 178 184 230 276

There are at least 7 different strategies, derived from addition modulo 88 and multiplication modulo each b of 89 115 178 184 230 276 on the numbers co-prime to b . Thus:

```
win ← {
  A winning strategy based on group table  $\alpha$  for hat
  numbering  $\omega$ 
    assert  $\omega \in \downarrow m \leftarrow \rho \omega$ :
     $\alpha \leftarrow m \mid \circ . + \cdot \downarrow m$       A default is + modulo  $m$ 
    assert  $(\rho \alpha) \equiv m, m$ :
    assert  $\alpha \equiv \Phi \alpha$ :      A must be abelian
     $G \leftarrow \alpha[0;] \downarrow \alpha$       A standardize group table
     $g \leftarrow \{G[\alpha; \omega]\} \cdot$       A group operation
     $g_i \leftarrow \{G[\omega;] \downarrow 0\} \cdot$       A inverses
     $(\downarrow m) \ g \ g_i \ (\circ . \neq \cdot \downarrow m) \ g . \times \omega$ 
  }

assert ← { $\alpha \leftarrow$  'assertion failure'  $\diamond 0 \in \omega : \alpha \sqsubseteq \text{SIGNAL } 8 \diamond \text{shy} \leftarrow$ 
0}
```

The left argument α of `win` is a group table. It is standardized by doing $\alpha[0;]\iota\alpha$, whence the group elements are ιm and the identity element is 0 . These properties permit the computation of the “all but” sums to be simplified from $\{g/(\omega \neq \iota m)/u[h]\}^{\iota m}$ to $(\circ.\neq\iota m)g.\times h$.

`mgrp` ← $\{\omega | \circ.\times\iota I \ 1=\omega \vee \iota\omega\}$ A group table for \times modulo ω

`mgrp 7`

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

`mgrp 9`

```
1 2 4 5 7 8
2 4 8 1 5 7
4 8 7 2 1 5
5 1 2 7 8 4
7 5 1 8 4 2
8 7 5 4 2 1
```

`mgrp ω` computes the group table for the integers under multiplication modulo ω . The group elements are the numbers co-prime to ω .

`m` ← 88

`h` ← random mpm

`I h = win h`

58

`I h = (mgrp 89) win h`

5

`I h = (mgrp 115) win h`

87

`I h = (mgrp 178) win h`

27

`I h = (mgrp 184) win h`

77

`I h = (mgrp 230) win h`

82

```
I h = (mgrp 276) win h
58
```

Even though win h and (mgrp 276) win h result in the same person (58) writing his hat number, the lists of written numbers are different:

```
win h
41 9 72 79 53 39 7 8 32 72 85 26 45 47 2 16 46 80 53 ...
(mgrp 276) win h
70 39 78 9 47 11 47 52 48 20 70 32 26 60 2 76 46 16 ...

(win h)[58]
5
((mgrp 276) win h)[58]
5
h[58]
5
```

Collected definitions

```
assert←{α←'assertion failure' ♦ 0∈ω:α □ SIGNAL 8 ♦ shy←
0}
```

```
I←{ω/ιρω}
```

```
mgrp←{ω|◦.×≍I 1=ω∨ιω} A group table for × modulo ω
```

```
random←{□rl←7*5 ♦ ?ω} A reproducible random numbers
```

```
win←{
A winning strategy based on group table α for hat
numbering ω
```

```
assert ω∈ιm←ρω:
```

```
α←m|◦.+≍ιm A default is + modulo m
```

```
assert(ρα)≡m,m:
```

```
assert α≡ϕα: A must be abelian
```

```
G←α[0;]ια A standardize group table
```

```
g←{G[α;ω]}'' A group operation
```

```
gi←{G[ω;]ι0}'' A inverses
```

```
(ιm) g gi (◦.≠≍ιm)g.×ω
```

```
}
```

References

1. <http://www.jsoftware.com/pipermail/general/2007-June/030272.html>
2. <http://www.jsoftware.com/pipermail/chat/2007-June/000514.html>

Function design

by Kai Jaeger (kai@aplteam.com)

Designing functions is something many APLers don't pay much attention to. They just carry on. This article covers some of the topics associated with function design. In particular it discusses different ways of how to pass parameters, when (and why) to create a direct function (dfn) and when a traditional function (tfn) and why honouring the DRY principle (don't repeat yourself) might be a good idea.

Passing parameters to functions

Passing parameters is something people tend to spend little time on. It's so natural to pass some data as an argument, what's there to ponder about? Well, a lot actually. There are so many different ways to do this that it is worthwhile to figure out what's best for certain circumstances.

Note that the techniques discussed rely on Dyalog because they rely on namespaces. They also assume $\square IO \leftarrow 0$ and $\square ML \leftarrow 3$.

The early days

Passing parameters in the first versions of APL was a serious limitation to the language: there was just a right argument or a left and a right argument, and one could only pass simple arrays. This is sufficient for problems that are somehow "mathematically oriented". For normal programming tasks however we more often than not need more than just 2 parameters.

Nested Arrays

Only with the introduction of nested arrays did APL become a real programming language: nested arrays made it possible to pass as many parameters as needed.

Mandatory and optional parameters

Often we can make a difference between parameters which are mandatory and those which are optional. In the past I often tended to provide mandatory parameters as the right argument and optional ones as the left argument. How to provide the mandatory parameters seemed to be obvious: define a sequence, often called fixed parameters.

Name-value pairs

For optional parameters it is less obvious. Fixed parameters are naturally not an option but what about name-value pairs? This allows us to specify a vector of two-item arrays like this one:

```
parms←''  
parms,←c 'hide' 1  
parms,←c 'workdir' 'C:\App'  
parms,←c 'debug' 0
```

Let's assume that we have a function `Foo` which takes just one mandatory parameter. The name-value pairs we've just defined allow us to specify them as optional parameters:

```
parms Foo someArray
```

Alternatively, we can provide no optional parameters at all:

```
Foo someArray
```

That's all well and good but there are a couple of obstacles we have to deal with. First of all, if we specify just one optional parameter:

```
(c'hide' 1) Foo someArray
```

we must enclose the single pair to ensure conformability. You might think that this can be avoided by investigating the left argument within `Foo` and deal with it appropriately depending on its depth but you might find this surprisingly difficult and error prone depending on the parameters you expect: if the second item of a name-value pair can be nested itself trouble is looming.

Two scalars also pose a problem:

```
(c'a' 1) Foo someArray
```

It is also not particularly easy to check the parameters for being valid. What about case sensitivity? Shall "hide" and "Hide" both be treated as a valid name for a certain parameter? Last but not least you have to assign the values to variables inside `Foo`.

Nothing of this is particularly laborious but you need to care about these problems.

Namespaces to the rescue

Now there is another approach which is ridding us of all these problems effortlessly. Look at this:

```
parms←[]NS' '
parms.hide←1
parms.workdir←'C:\App'
parms.debug←0
```

In the first line we create a new unnamed namespace and assign it to parms. To rephrase it: parms is now a reference pointing to an empty namespace. We then create variables inside this namespace with the appropriate values.

Now we can pass this namespace as left argument:

```
parms Foo someArray
```

Looking at this from inside Foo there are some differences:

We don't need to bother about the format of the left argument.

We don't need to establish variables – we already have them.

As a bonus the parameters are separated from other local variables in the function.

All this makes this solution significantly more attractive than name-value pairs, but there is even more. Look at this function:

```
[0] parms←CreateParmsForFoo
[1] :Access Public Shared
[2] A Creates a namespace with default values for all
[3] A optional parameters of method Foo
[4] parms←[]NS' '
[5] parms.hide←1
[6] parms.workdir←'C:\App'
[7] parms.debug←0
```

Note that the first line makes this a public shared method. Now let us assume that CreateParmsForFoo and Foo are methods of a class "Sample". Let's also assume that Foo has a line :Access Public Shared . When you consider calling the method Foo with a certain parameter different from its default you can now do this:

```
myParms←#.Sample.CreateParmsForFoo
myParms.hide←0
myParms Foo ...
```

From a user's point of view this is not different except that a) she cannot forget to enclose a single pair, b) she can stop worrying when passing two scalars. Most importantly, if she is interested in the default values processed by Foo she doesn't need to look into Foo anymore, or combing through documentation.

Inspecting the contents of the namespace gives the answer. In short: life is easier now from a caller's point of view.

Other examples are overloaded constructors of classes. Depending on the number of parameters as well as the data types of the parameters somehow automatically the correct constructor is executed. That sounds nice but it has a clear drawback: it makes reading and understanding a statement like the following one harder than necessary not only because of the positional parameters provided but also because there is no easy way to find out the defaults for the parameters not specified:

```
NEW Sample ('hello' 1 (3 4) 'universe')
```

Obviously these statements:

```
myParms←#.Sample.CreateParmsForFoo
myParms.hide←0
myParms Foo ...
NEW Sample (, <myParms)
```

are more readable but also provide an easy way to inspect the defaults.

Adding a “List” method

We can make the caller's life even more comfortable by adding one more line to CreateParmsForFoo:

```
...
[7] parms.debug←0
[8] parms.⌊FX'r←List' 'r←{ω,τ⊕ω}~ο'' ''↓⌋nl 2'
```

Now after assigning the result of the function to a variable my one can say:

```
my.List
debug      0
hide       1
workdir    C:\App
```

That is certainly a nice way to investigate the contents of the parameter space.

Checking optional parameters

For a programmer it is also more convenient to deal with namespaces rather than name-value pairs. Think of how to make sure that such a namespace contains just the variables it's supposed to contain. Our method could achieve that quite simply:

```
[0] {optional}Foo array;f;b;msg;l
[1] :Access Public Shared
[2] :If 0=⌊NC'optional'
```

```
[3] optional←CreateParmsForFoo
[4] :ElseIf 0<1>optional.␣NL 2
[5]   l←~o' '""↓optional.␣NL 2
[6] :AndIf 1∈b←~l←CreateParmsForFoo.List[;0]
[7]   msg←'Invalid optional parameter',((1<+/b)/'s'),' :
[8]   11 ␣SIGNAL~msg,↑{α,',',w}/b/~o' '""↓optional.␣NL 2
[9] :EndIf
[10] A ...
```

Line 3 creates optional with default settings in case no left argument was provided.

Line 4 checks whether optional is empty. If it is not...

Line 6 checks whether we have a problem. If we have one than b (for Boolean) can be used as a mask.

Line 7 compiles a proper message and...

Line 8 adds the name(s) of the problem case(s), performs some formatting gymnastics and throws an error.

If we now do this:

```
parms←CreateParmsForFoo
parms.whatIsThat←'?'
parms.Hide←0
parms Foo 1
```

we get

```
Invalid optional parameters: Hide,whatIsThat
```

because the parameter “Hide” was misspelled.

Obviously this way of specifying parameters has advantages for both, the user of a function (or method) as well as the implementer of it.

References as parameters

So far we have restricted the stuff a parameter space can contain to variables. There is a good reason to lift that restriction. Think of a parameter `refToUtils`. Obviously his parameter is expected to be a reference to a namespace that holds utilities. Presumably the default is just `#`. In order to deal with references we need to change the `List` function so that it also reports references:

```
parms.$FX'r←List' 'r←{ω,[0.5]±ω}' ' ' '~'~'~↓nl 2 9'
```

Constants

Sometimes you might want to add a “parameter” which can’t actually be changed because its value depends on the environment and can be worked out by the CreateParmsFor function itself. Why would you want to add this? To give the programmer an easy way to actually look at the information. That is not only convenient; it also makes clear that this value is taken into account by the program the parameter space is going to be fed to. But the user must not change it, so a variable is not appropriate. APL has no concept of what is called a Constant in most other programming languages. Niladic functions to the rescue:

```
∇r←IS_DEVELOPMENT
[1]  r←'Development'≡3>'#'\WG'APLVersion'
∇
```

Strictly speaking this is not a constant, but a niladic function poses convincingly as a constant. In most other programming languages names for constants use uppercase letters; that makes them easy to recognize. This seemed to be a good idea so I adopted this here.

Enhancing “List”

To include our pseudo-constants we need to make sure that List is taking functions into account but without List itself, therefore we localize List:

```
parms.⊞FX'r←List;List' 'r←{ω,[0.5]⊥ω}' ' '~''~↓⊞nl 2 3 9'
```

Finally the information what name class a certain parameter actually belongs to is sometimes valuable, so we add it to the result returned by List:

```
parms.⊞FX 'r←List;List' 'r←{(ω,[0.5]⊞NC ω),⊥ω}' ' '~''~↓⊞NL 2 3 9'
```

Our function would now look like this:

```
r←CreateParmsForFoo
:Access Public Shared
A Creates a namespace with default values for all
A optional parameters of method Foo
r←⊞NS''
r.hide←1
r.workdir←'C:\App'
r.debug←0
r.⊞FX'r←IS_DEVELOPMENT' 'r←'D''≡3 1>'#'\WG'APLVersion''
r.refToUtils←#
fns←'r←List;List' 'r←{ω,⊥ω}~ο' ' '~''~↓⊞nl 2 3 9'
r.⊞FX fns
```

Let’s check:

```

    parms←CreateParmsForFoo
    parms.List
IS_DEVELOPMENT  3.1      1
debug            2.1      0
hide             2.1      1
refToUtils       9.2      #
workdir          2.1    C:\App

```

That's exactly what we are looking for; job done.

Direct functions or traditional functions?

When direct functions were introduced by Dyalog my first thought was something along the lines of “well, nice, but there are more important things we need right now.” That was a long time ago; we called them dynamic functions back then. Boy has my opinion changed since then! Today about 90% of the functions I write are direct functions. Why is that?

Name scope

First of all it's about name scope: when a variable is created in the direct function (also called dfns or curlies because of the curly brackets {}) all the variables created inside that function are local by default. In order to create a true global you have to say:

```
⌈THIS.MyGlobal←'something'
```

Even better: “local” really means local: in traditional functions every local variable created in a function can be seen by functions called within that function like a global which is the reason why they are sometimes called semi-globals. There is no such problem in dfns: local really means local.

You think that's not that important? Allow me to tell you a story emphasizing the fact that it is important: In the early eighties I worked for a client who ran VSAPL on a mainframe. My task was to maintain and enhance a large program written by somebody who had moved on. One day I inserted a new comment into the main function, the one executed by ⌈LX . An hour or so later I got feedback from users claiming that the results were rubbish. It took me a while to make the connection: was it me who was to blame for the problem because I added the comment line? A short investigation showed that this was unlikely: the program did not do any branching with → , so how could the new comment line make a difference?

I went for the pragmatic approach anyway and removed the comment line. Then I restarted the program and asked the users. They reported that now the program's results were back on track.

But why was that? As it turned out the function had a couple of labels defined, despite not using branching in that function. That was not unusual in these days: the original author used labels not only for branching but also for documentation purposes. Now in a traditional APL function a label is simply an integer variable and its value is the line number, and they are also semi-globals.

As it turned out deep in the calling hierarchy there was a function which used branching, and it tried to jump to a label with a name also used in the main function. Unfortunately the programmer forgot to specify the label. Rather than causing a VALUE ERROR APL managed to find a variable with that very name on the stack. Finally the value of that variable (=the line number) was good enough to let the function with the missing label do its job.

When I introduced my comment line, the label got a new value; in the function with the missing label that had a consequence: one line that was executed in the past was now simply ignored. Unfortunately this did not make the program fail, it rather created wrong results. This is a perfect example why local should really mean local.

Drawbacks of dfns

Unfortunately direct functions come with drawbacks, some of which could be removed easily by Dyalog:

- There are no stop vectors. Despite the editor pretending otherwise dfns do not honour stop vectors. There are rumours that this will be fixed in version 13.2.
- If in a direct function a value is assigned twice to a variable "foo" then the second assignment effectively creates a new variable "foo" shadowing the first one. "So what?!" you might ask, but this has a nasty side effect when you try to watch changes made to a variable by opening an edit window on that variable: that works brilliantly with tfns but not at all with dfns: the new value is not shown because a new variable is created, and the editor does not care about this new variable.
- Sometimes people complain that it is a disadvantage that you cannot have named arguments with dfns, therefore reducing readability; right, good point, but nothing stops you from saying `(parm1 parms anotherParm)←ω`

in the first line of your dfns which has the same effect. Like in tfns these three variables are local by definition, so everything is okay.

- Sometimes a `:For` loop or a `:Select` statement has its merits, in particular when it comes to debugging, but in those rare cases one can still write a tfn.

DRY and functional programming

DRY[1,2] stands for “don’t repeat yourself”. It means that any piece of information should only have just one representation in an application. Easy to understand examples are:

- The name of an application which is repeatedly shown in window captions.
- The main key of all Windows Registry entries used in an application.
- Rather than repeating these pieces of information over and over again it should come from just one source, be it a variable or the result of a function call. In these cases the advantages are obvious: in case of a change one needs to change just one single line in the application and the job is done.

However, there are less obvious cases: when a certain piece of code gets used in two or more different places then this is already good enough a reason to put it into a separate function. Let’s discuss this by looking at a real life example.

Calculating index positions

With the help of direct functions calculating index positions can be done by the expression $\{\omega/\iota\rho\omega\}$. Is assigning this expression to a function name a good idea or not? According to the DRY principle the answer must be yes. Reality proved that this is true.

There was a longstanding bug in Dyalog: prior to version 13.0 the expression $\iota\theta$ returned $\square\text{IO}$ when it should have returned $\epsilon\theta$. You may think “so what?!” but this can have quite a dramatic impact: when the expression $\{\omega/\iota\rho\omega\}$ gets a scalar as argument, the result in 12.1 is very different from that in 13.0. With $\square\text{IO} \leftarrow 0$ it is:

```
12.1:  0 ↔ {ω/ιρω} 1
13.0:  εθ ↔ {ω/ιρω} 1
```

In most applications we have to make sure that the expression continues to return `⊥IO` rather than an enclosed empty vector. Now when there is a function defined like this one:

$$\text{Where} \leftarrow \{\omega / \iota p \omega\}$$

then obviously you change that function to

$$\text{Where} \leftarrow \{\omega / \iota p, \omega\}$$

and your application is ready for 13.0 in this respect. Instead you might perform a search in a big application and find plenty of places where the expression is used. All of them need to be changed.

By following the DRY principle you are going to write code that is better to maintain, but there is a second advantage: your functions tend to get smaller, and that's a good thing.

Where is the DRY principle coming from?

This is what the Wikipedia has to say:

In software engineering, Don't Repeat Yourself (DRY) is a principle of software development aimed at reducing repetition of information of all kinds, especially useful in multi-tier architectures. The DRY principle is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

The principle has been formulated by Andy Hunt and Dave Thomas in their book *The Pragmatic Programmer*. [...] When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically-unrelated elements. Additionally, elements that are logically related all change predictably and uniformly, and are thus kept in sync.

Is the DRY principle globally valid?

Well, some big shots claim that it is not: a given test case should create its environment, execute the test cases and then tidy up. Actually it should tidy up (because there might be leftovers from a former test case that failed), create its environment, perform the test and finally tidy up again.

This allows one to read the test case from top to bottom and understand what it's supposed to do and how it will try to do it. At the same time it's likely to violate the DRY principle.

Of course one has to make exceptions: if many or all test cases need a particular environment and creating that is costly than this should be created upfront and

deleted after having executed the last test case. A typical example is creating a (potentially big) data base.

Function size

Functions should be small, and most people would agree with this. But what exactly should go into a function? Following the DRY principle will quite often introduce plenty of small functions into an application by separating certain parts of the code, but that alone is not enough.

Often programmers populate a function with statements that are supposed to be executed together in terms of time. In all honesty, that is not good enough a reason to put them together! A good advice I once got from an old-hand is that when you have difficulties to find a proper name for a function then that function is a candidate for splitting it into several functions.

Statements should be combined into a function when they serve a certain task. That also makes it easier to find good names.

Names

Many people have strong views on this, but it's probably fair to say that there is some common ground between most people.

Avoid abbreviations

Names should clearly advertise what a function is doing. With autocomplete at your fingertips there is no good reason to avoid clear self-explaining names. Actually, there are not even bad reasons. Abbreviations are fine when they are well-established: programmers would know what `RC` stands for, so that is fine. But `cntrs` rather than `countries` does not save much and certainly makes a program harder to read. It can also become an obstacle when somebody scans code for "country".

The trouble is that within a program you are just about to write you might find such abbreviations handy, and reading as well as understanding them is not a problem at all. Right, but when another programmer will take over one day she might be in more trouble than necessary. Even you might run into this: I myself more than once stopped looking at some code for a while, and when I came back 2 years later I found the names hard to understand.

Meaningful names

There are programmers who always name the first local variable needed in a program with `a` and the next one will be `b` and so on. Bad move. It saves you the

time to find a good name which quite often is a difficult thing to do, but it makes it harder to read.

Reasonable exceptions

If a function has only a few lines however then nothing is wrong with a statement like this:

```
lc←GetAllCountryNames 0      A List of Countries
```

and then using the name `lc` later in that function. It's easy to read and understand, and because the function is short the assignment line will always be visible, so there is nothing wrong with this technique.

Side effects

One of the fundamental ideas of functional programming is to avoid side effects. That's why functional programming languages are back in the mainstream: side effects are deadly in multi-threaded (or multi-cored) programs.

APL is not only a functional programming language: it was the very first functional programming language ever. Worth to remember because these days I keep reading claims that Lisp was the first one.

Although the general design of dfns makes it almost impossible to implement functions that have unwanted side effects it is of course still possible to implement dfns that actually have side effects. However, it is certainly a good idea to try hard to avoid this or at least to make the fact obvious.

For example, it is certainly a good idea to emphasize the fact that in a dfn a global variable is updated. Although the following statement would work in a dfn if there is a global variable `Buffer` (otherwise it's a VALUE ERROR):

```
Buffer,←c
```

It might be better to say:

```
⌈THIS.Buffer,←c
```

which makes it obvious that a global is involved here.

Conclusion

There is certainly more to say about the design of functions in general and how we pass parameters to them. Discussing the issue with my fellow colleagues turned out to be surprisingly difficult. It seems that APLers have particular

difficulties to agree on something, or shall I say anything? That's a bit strange and certainly not helpful. It might be one of the reasons why APL is not as successful as it deserves to be because it stops us from doing things that other communities have so easily achieved, for example common libraries of utilities, all developed following certain style guidelines accepted by that community. That is certainly true for the Ruby and the Python community, and it is likely to be true in other communities as well. I would love to see a discussion regarding this in the APL world, a discussion that would ideally result in a paper "APL Style Guidelines" we can hand over to newcomers one day.

References

1. http://en.wikipedia.org/wiki/Don't_repeat_yourself
2. Interview with Andy Hunt and Dave Thomas:
<http://www.artima.com/intv/fixit.html>

J

Backgammon tools in J

3: Two-sided bearoff probabilities

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

J programs are presented as analytical tools for expert backgammon. Part 3 here develops a program to compute probability distributions for two-sided bearoff (without contact) and uses it to determine the probability of winning.

The inner board is represented as a list of six integers. For example, one piece on the 5-point and one piece on the 6-point:

```
board =: 0 0 0 0 1 1
```

Utility programs:

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

Master program PD produces a list of probabilities for bearing off in increasing numbers of rolls. Defined much like `Rolls` in [1], it calls `PDBearoff` or else returns 1 when all pieces are off the board:

```
PD =: PDBearoff ELSE 1: WHEN Alloff
    Alloff =: All@(Board = 0:)
    Board =: ]
    All=: And/
    And =: *.
PDBearoff =: 0: , Average@AllPDs
    Average =: +/ % #
    AllPDs =: PD"1@BestMoves

BestMoves =: PDDoubles , 2: # Better@PDSingles

doubles =: > 1 1 ; 2 2 ; 3 3 ; 4 4 ; 5 5 ; 6 6
singles =: > 1 2 ; 1 3 ; 1 4 ; 1 5 ; 1 6 ; 2 3 ; 2 4 ; 2 5 ; 2 6 ;
3 4 ; 3 5 ; 3 6 ; 4 5 ; 4 6 ; 5 6
singles =: singles ,: |."1 singles

PDDoubles =: doubles Best@AllMoves"1 Board
PDSingles =: singles Best@AllMoves"1 Board
Better =: Best@,: "1/
    Best =: {~ MinIn@:(N"1)
    MinIn =: i. <./
```

Note that program N (from [1]) is used in Best to evaluate boards.

AllMoves and its subprograms are the same as in [1]:

```
AllMoves =: (Die2 Moves Die1 Moves Board) ELSE
              (Die1 Moves ^:4 Board) WHEN (Die1=Die2)
  Die1 =: First =: {.@Dice
  Die2 =: Last =: {:@Dice
  Dice =: [
  EACH =: &.>
Moves =: ~.@;@.(LegalMoves EACH <"1)
  LegalMoves =: (Possibles Move"1 Board) ELSE Board WHEN AllOff
    Possibles =: FromTo Where FromTo OK"1 Board
      FromTo =: On ,. On-Die1
      Where =: #~
        On =: Points Where Board > 0:
      OK =: Off Or Inside Or Highest
        Or =: +.
        Inside =: Last > 0:
        Off =: Last = 0:
        Highest =: First = Max@On
        Max =: >./
      Move =: Board + To - From
        From =: Points e. First
        To =: Points e. Last
        Points =: 1: + i.@6:
```

For example, probabilities of bearing off the board above in 0, 1, 2, 3, and 4 rolls:

```
PD 0 0 0 0 1 1
0 0.166667 0.707562 0.124829 0.000943073
```

Such a probability distribution is related to expected number of rolls (N):

```
+ / pd * i.#pd =. PD 0 0 0 0 1 1
1.96005
N 0 0 0 0 1 1
1.96005
```

Further, PD can be used to determine the probability of the first player winning:

```
Pwin =: X/@,:&PD
X =: + / . * +/\.
```

Pwin computes the PD of both player's inner boards, then sums products of player 1's probabilities times reverse cumulative sums of player 2's probabilities – that is, the sum of probabilities of 1 bearing off times the respective probabilities of 2 not bearing off.

The result is the probability of player 1 winning the bearoff.

Example:

```
0 0 0 0 1 1 Pwin 0 0 0 0 2 0
0.766415
```

The first player is about 77% likely to win.

The programs above are extremely inefficient, so PD should be re-defined to utilize a pre-established database of probability distributions for fast look up (as done for N in Part 1). Upon request, the author will provide a script with efficient programs.

References

1. Peelle, Howard A. "Backgammon Tools in J (Part 1) Bearoff Expected Rolls", Vector, Vol. 24, No. 2&3.
2. Peelle, Howard A. "Backgammon Tools in J (Part 2) Wastage", Vector, Vol. 24, No. 4.

Savitzky-Golay Interpolation for Smoothing Values and Derivatives

David Porter (dporter@cissoid.net)

Cliff Reiter (reiterc@lafayette.edu)

1 Introduction

In a previous note on image processing filters [4] Cliff observed that the Savitzky-Golay filter could be used to smooth data, thereby removing some noise, but also sharpening the edges somewhat. That is a classic image processing application, [3, 8, 10]. David has noted that filters based on the Savitzky-Golay kernels are really much more general than the case considered in that reference. Besides smoothing values, it can provide smoothed derivatives of many orders. In fact, a classic application is using Savitzky-Golay estimates for the first derivative to pin-point spectral peaks [2, 9]. The usefulness of the higher order derivatives is enhanced by the overlapping local-polynomial smoothing inherent in these filters.

The central idea of Savitzky-Golay filters is to use best least squares polynomial fits to approximate data; then use those polynomials to estimate data or derivatives. The beauty is that weights may be computed ahead of time so that the approximations can be computed very rapidly. In this note we explain how that works, implement it in J [1], and show applications using both value and derivative approximations to image processing.

2 Computing Weights

We begin by assuming that we want to estimate a value or derivative at the point $u_0=0$ and that we know the data values at lr (left radius) uniformly spaced points to the left of u_0 and rr (right radius) uniformly spaced points to the right of u_0 . Thus, for purposes of our derivation and applications we assume the points at which the data is known are $lr=u_{-lr}, \dots, u_0=0, \dots, u_{rr}=rr$. Suppose we want to use a polynomial of degree N to approximate the data. We need $lr+rr \geq N$ for the least squares polynomial to be well defined. Suppose the polynomial has the form $p(u)=a_0+a_1 u+a_2 u^2+\dots +a_N u^N$ and the corresponding values are given by v_i . That is, we desire the least squares solution to the system of equations $p(u_i)=v_i$. In matrix-vector form this can be written as $U A = V$ where A is the list of the polynomial coefficients and V is the list of the data values. The matrix U has

entries $U_{i-1ptj} = u_{i-1}^j$. Thus, the solution is given by $A = U^* V$ where $U^* = (U^T U)^{-1}$ is the least-square pseudo-inverse. The pseudo-inverse is a primitive in J and APL.

In order to determine $a_0 = p(0)$ we matrix multiply (a dot product here) the first row of U^* times the data values V . Thus, the weights we use as a filter to estimate the values are given by the first row of U^* . Likewise, the row with index k estimates a_k . A standard fact about the coefficients for McLaurin polynomials is that $a_k = p^{(k)}(0)/k!$. Thus, to estimate the k^{th} derivative, the weights we want are given by the row of U^* with index k times $k!$. We implement this in J as `sgw` shown below. Its left arguments are the order of the derivative and degree of the polynomial. The right argument is the left and right radii to use. The expression to the right of `%` computes the matrix U . The pseudo-inverse is computed. The expression `(!@[*{)` selects the row and multiplies by the factorial. We see some examples of weights below. The last one of those gives the weights that will approximate the first derivative using a degree three polynomial by utilizing two points to the left and right of the data point of interest.

```
sgw=: 4 : 0
({.x)(!@[*{)% (({.y)~i.1+({.+{:})y) ^/ i. 1+{:x
)

0 1 sgw 2 2
0.2 0.2 0.2 0.2 0.2

0 3 sgw 2 2
_0.0857143 0.342857 0.485714 0.342857 _0.0857143

1 1 sgw 1 1
_0.5 0 0.5

1 3 sgw 2 2
0.0833333 _0.666667 4.48235e_17 0.666667 _0.0833333
```

Next we define the matrix product and check that when we multiply actual polynomial data times these weights we get the appropriate derivatives. We use the fourth degree polynomial with coefficients 5 4 3 2 1 for these examples.

```
mp=: +/ . *

p=: 5 4 3 2 1&p.

((0 4 sgw 3 3)mp p i:3); p 0
+---+
|5|5|
+---+

((1 4 sgw 3 3)mp p i:3); p d.(1) 0
```



```

+--+--+
|4|4|
+--+--+
  ((2 4 sgw 3 3)mp p i:3); p d.(2) 0
+--+--+
|6|6|
+--+--+

  ((3 4 sgw 3 3)mp p i:3); p d.(3) 0
+---+---+
|12|12|
+---+---+

  ((3 4 sgw 2 4)mp p i:3); p d. (3) _1
+----+----+
|_12|_12|
+----+----+

```

The last of those illustrates what happens when the left and right radii are different. With two points on the left, a point of interest, and four points on the right and independent variable based on `i:3`, we see `_1` is the point of interest in the polynomial's coordinates (which is not the same as the analysis coordinates).

3 Applying Filters to Oscillations

Before discussing the Savitzky-Golay filter on image data, we will take a look at its behaviour on illustrations that have been modified with some standard normal noise. In this section we consider oscillations of varying frequency. We generate the standard normal values using `randsn` from *povkit2.ijs* from the *fvj3* add-on. Figure 1 shows the oscillations and a noisy variant.

```

require '~addons/graphics/fvj3/povkit2.ijs'

require 'plot'

y=: sin *: 0.01 *i.1000

plot y,:~z=(0.1* randsn 1000)+y

```

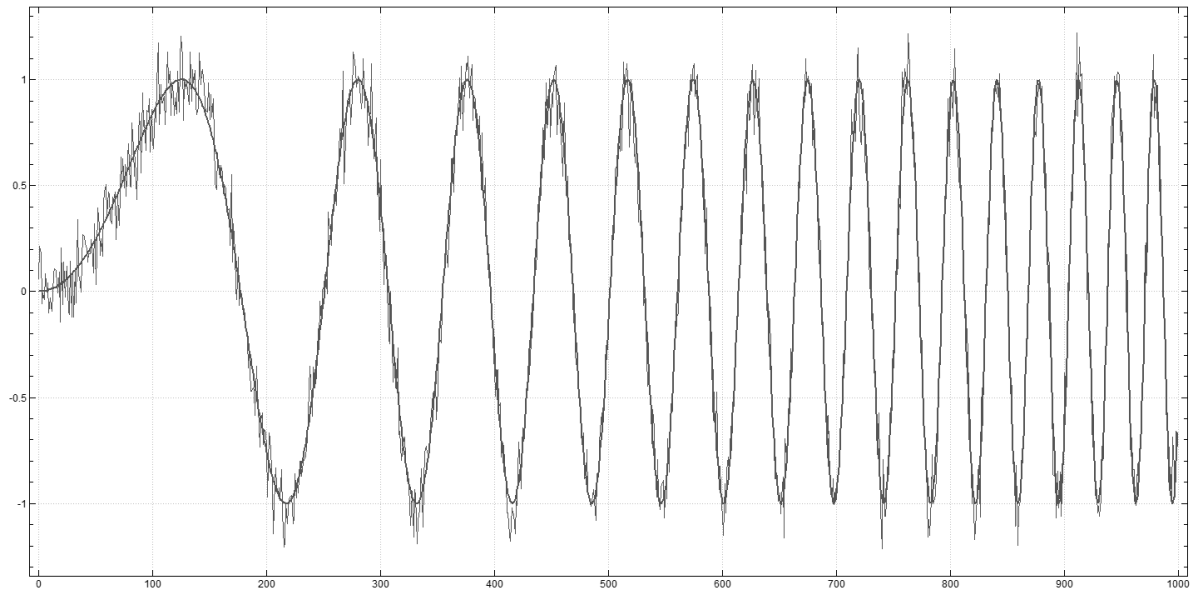


Figure 1: Oscillations and added noise.

Our goal is to, as best possible, recover the original (red) curve from the noisy data (blue).

In order to compare the data to the result of a Savitzky-Golay filter on the data it is convenient to extend the data and we will extend the data in a constant manner using context, repeating the first and last values an appropriate number of times. The conjunction SG takes the derivative order and polynomial degree as its left argument and the right argument is the radius. The conjunction extends the data in a constant manner and applies the filter `(m sgw n)&mp` to all the windows of appropriate size. Below we compute Savitzky-Golay weights with left and right radii 15 so that we will be using windows of width 31. Fifth degree polynomials are being used. The result is shown in Figure 2; this Savitzky-Golay filter (red) smooths the data substantially, but still has some noise and edge artefacts.

```
conext=:1&$: : ((#,:@:{.}),],(#,:@:{:}))

3 conext i.5
0 0 0 0 1 2 3 4 4 4 4

SG=:2 : '(1+2*n)&((m sgw n)&mp\ )@:(n&conext)'

f1=: 0 5 SG 15

plot z,:~f1 z
```

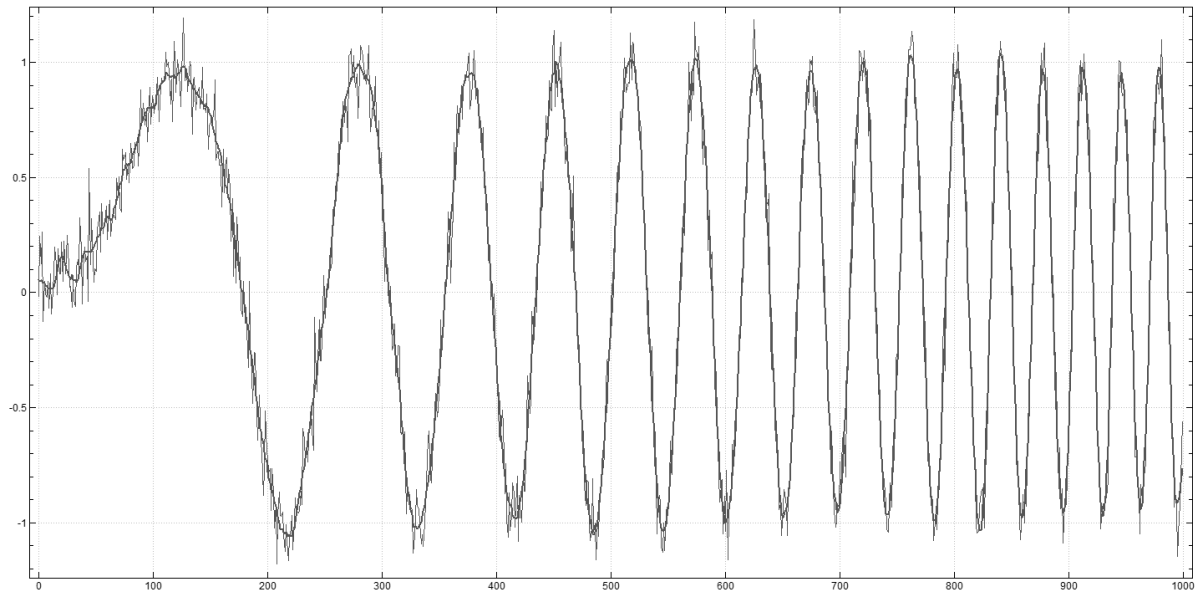


Figure 2: Savitzky-Golay filter preserves amplitude.

Note that that Savitzky-Golay filter does a fine job of recovering the proper amplitudes. That is in contrast to what happens when a simple moving average is used instead, as seen below and in Figure 3. Namely, amplitude information is lost. Other filters may damage frequency or phase information. Note that the Savitzky-Golay weights are uniform when linear functions approximate values, so we can obtain a radius 15 moving average with f2 below.

```
0 1 sgw 2 2
0.2 0.2 0.2 0.2 0.2
```

```
f2=: 0 1 SG 15
plot z,:~f2 z
```

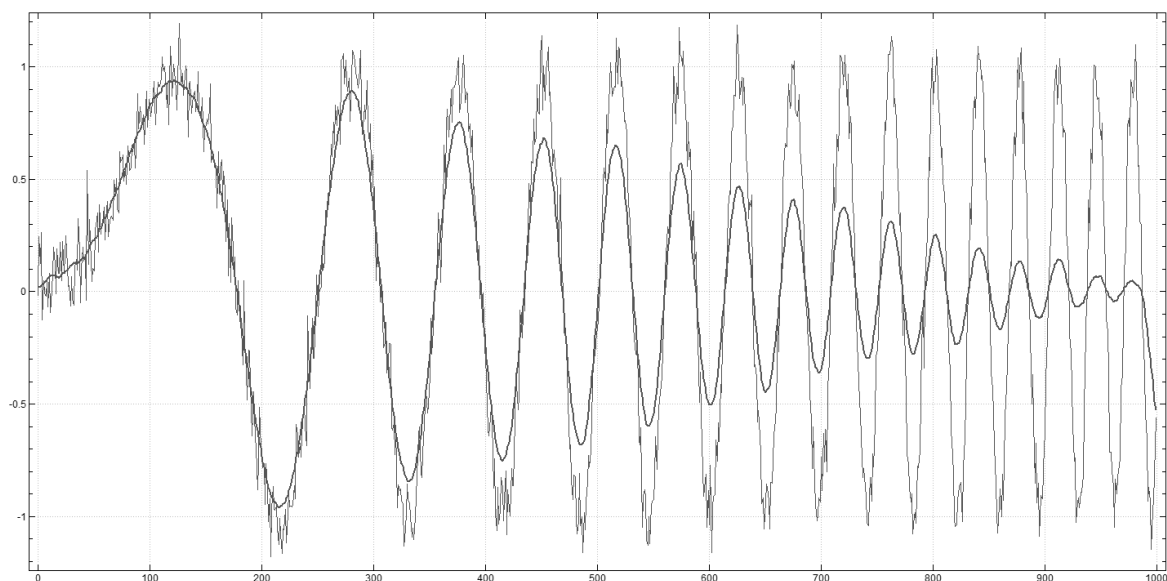


Figure 3: Ordinary moving averages lose amplitude information on narrow peaks.

4 Applying Filters to Edges

There are three parameters to the Savitzky-Golay algorithm that must be chosen. The following examples of computer generated edges (step functions) help illustrate the behaviours these parameters control. Figure 4 shows two steps, one pure and the other has noise added to it.

```
st=: 100<:i.200
```

```
plot rst=: st,st+0.1*randsn 200
```

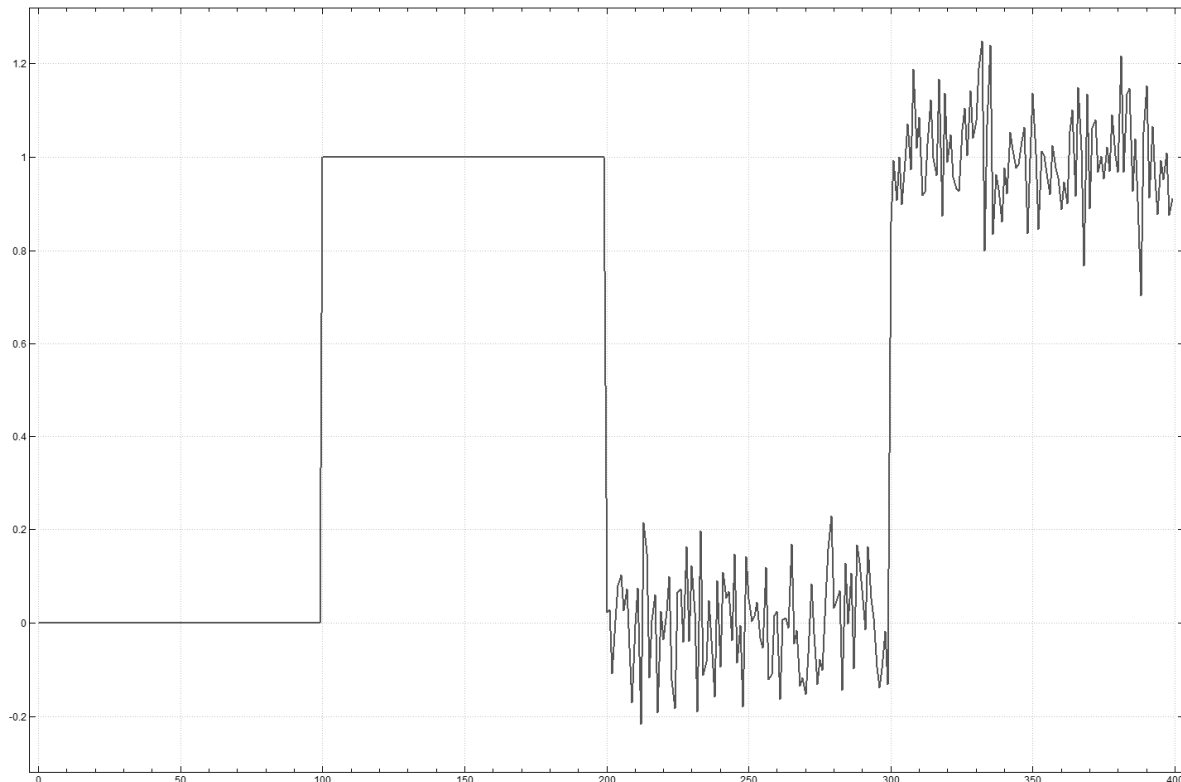


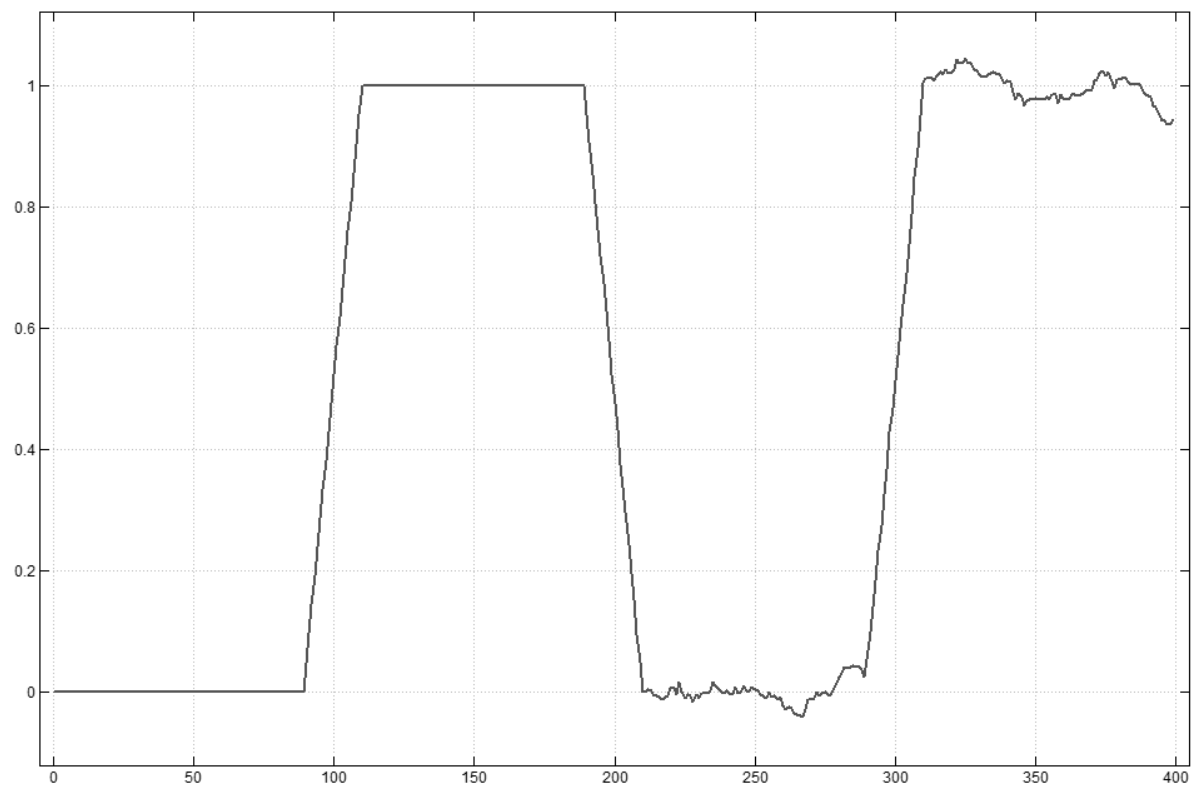
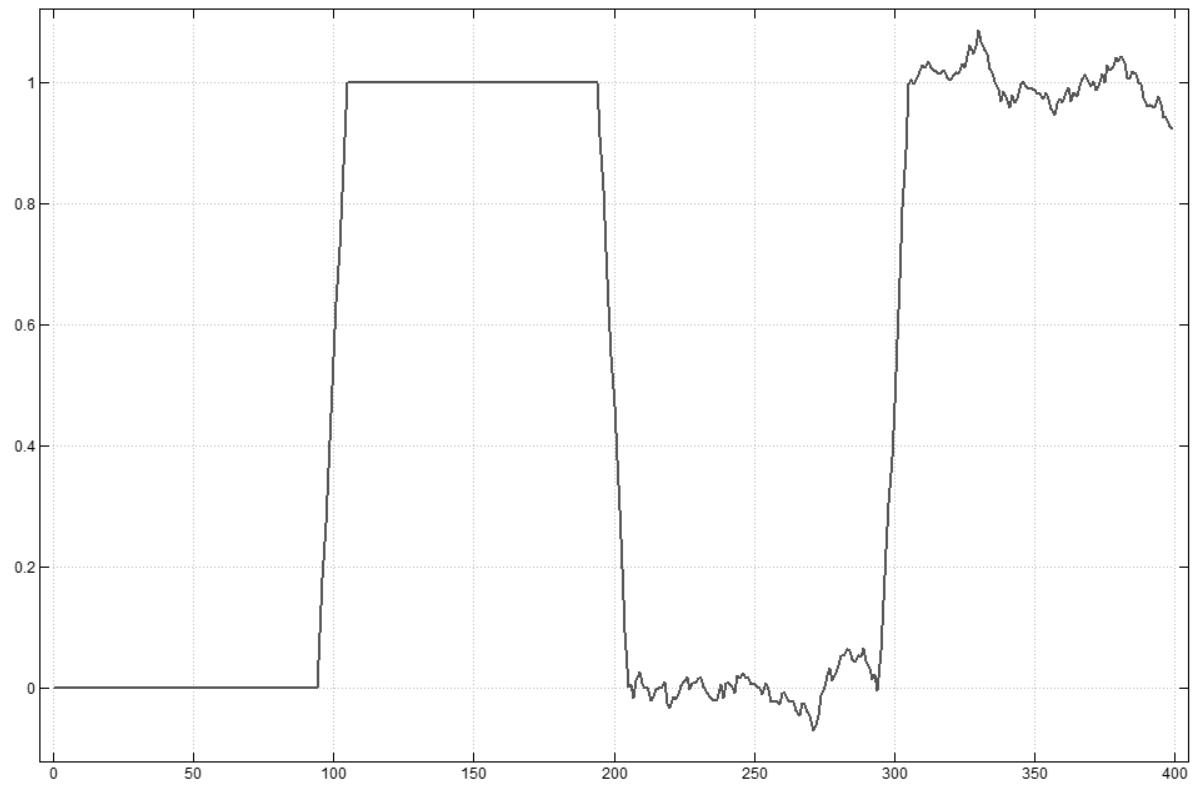
Figure 4: Step functions: pure and noisy.

Figure 5 show the effects of changing the smoothing window radius when using a Savitzky-Golay filter using the zeroth derivative, a zero degree polynomial, and a smoothing window radii of 5, 10, and 15 pixels. This is equivalent to a running average with 11, 21, and 31 sample points. As can be seen, the noise is reduced with larger radii and the edges become broadened and less steep.

```
plot 0 0 SG 5 rst
```

```
plot 0 0 SG 10 rst
```

```
plot 0 0 SG 15 rst
```



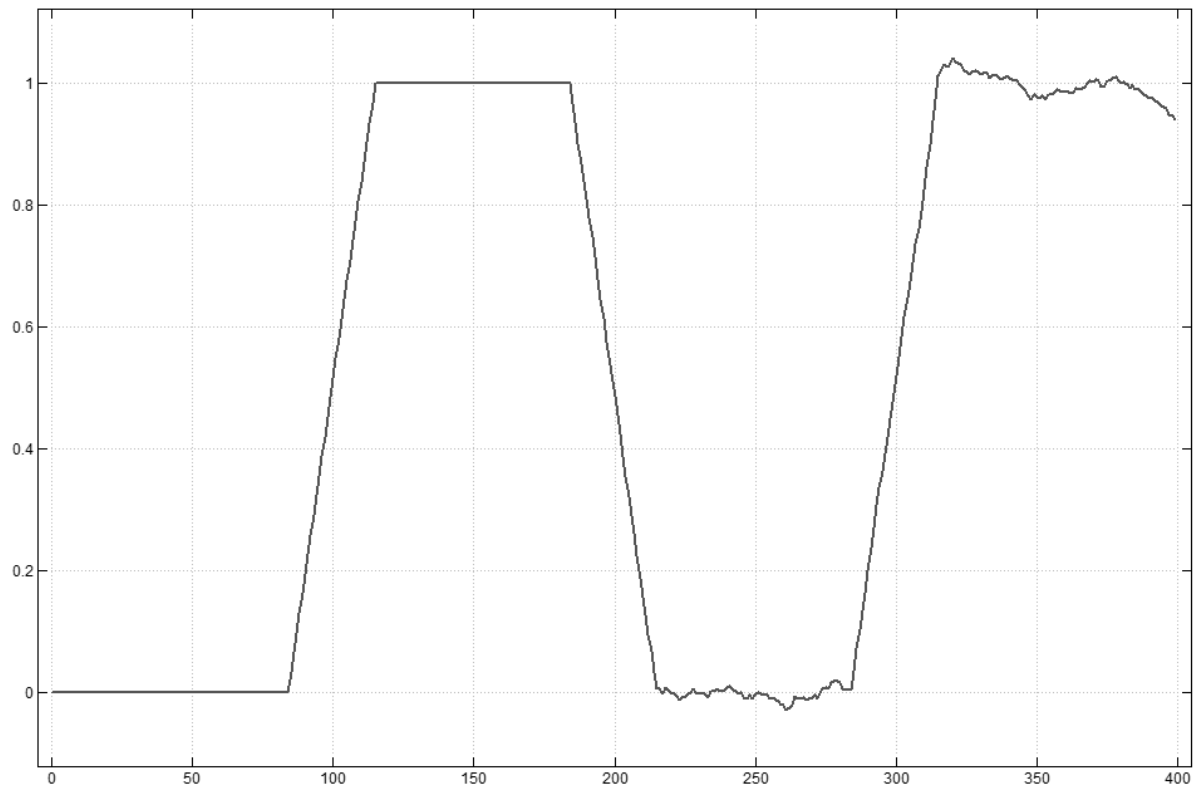


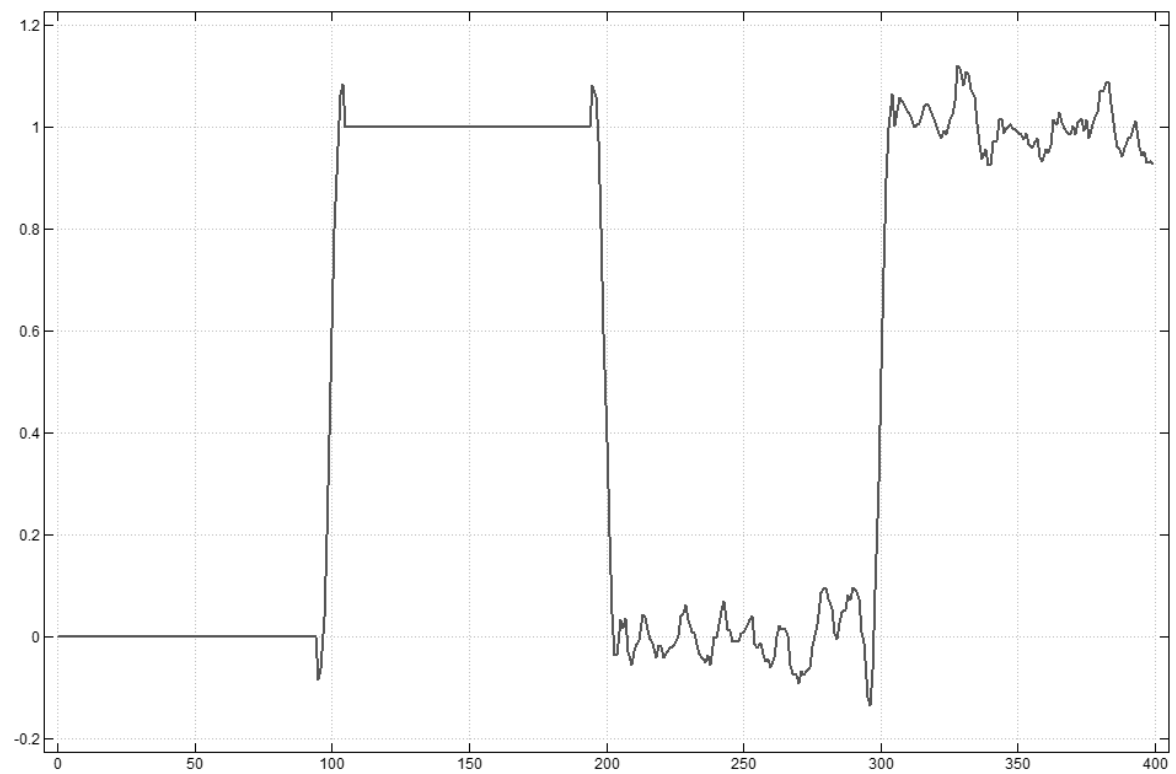
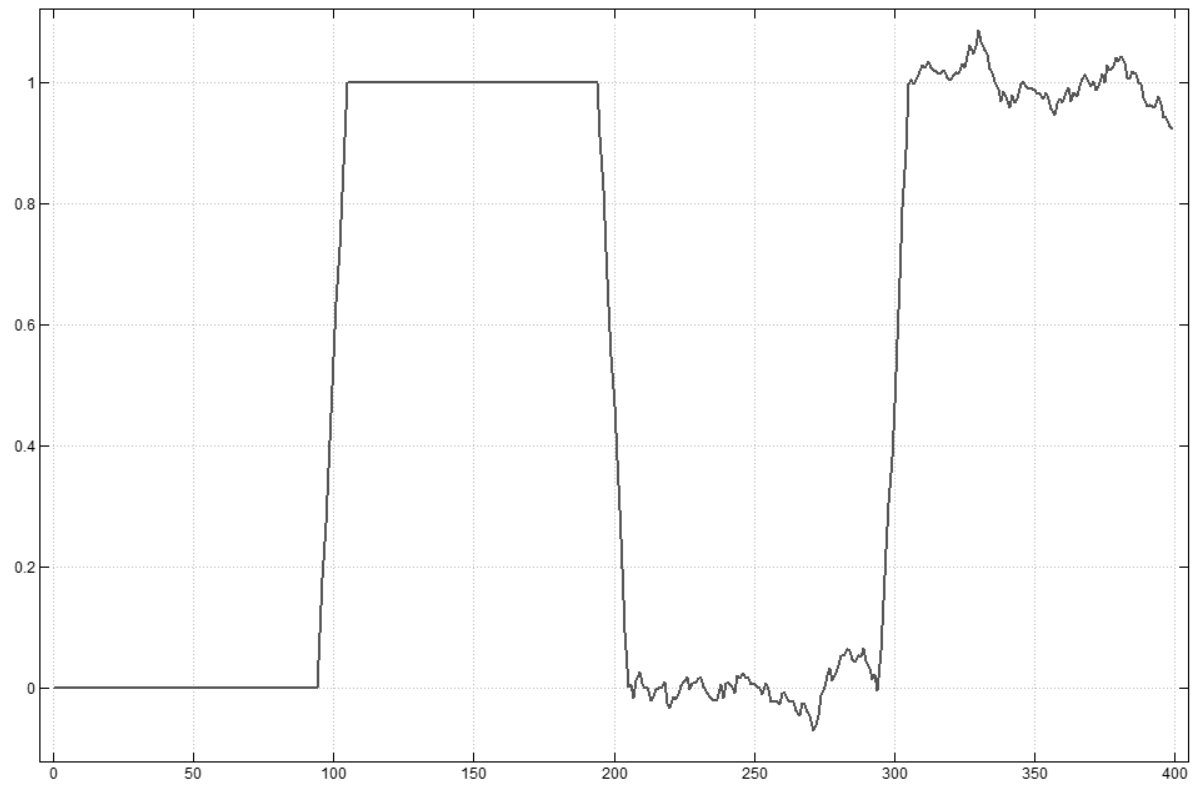
Figure 5: Smoothed with radius 5, 10, and 15 filters.

In Figure 6 the Savitzky-Golay parameters are set for zeroth derivative, a 5 point smoothing window radius, and the polynomial order 0, 2 and 4. The noise at higher polynomial orders contains more high frequency components and overshoot is apparent at the pure step.

```
plot 0 0 SG 5 rst
```

```
plot 0 2 SG 5 rst
```

```
plot 0 4 SG 5 rst
```



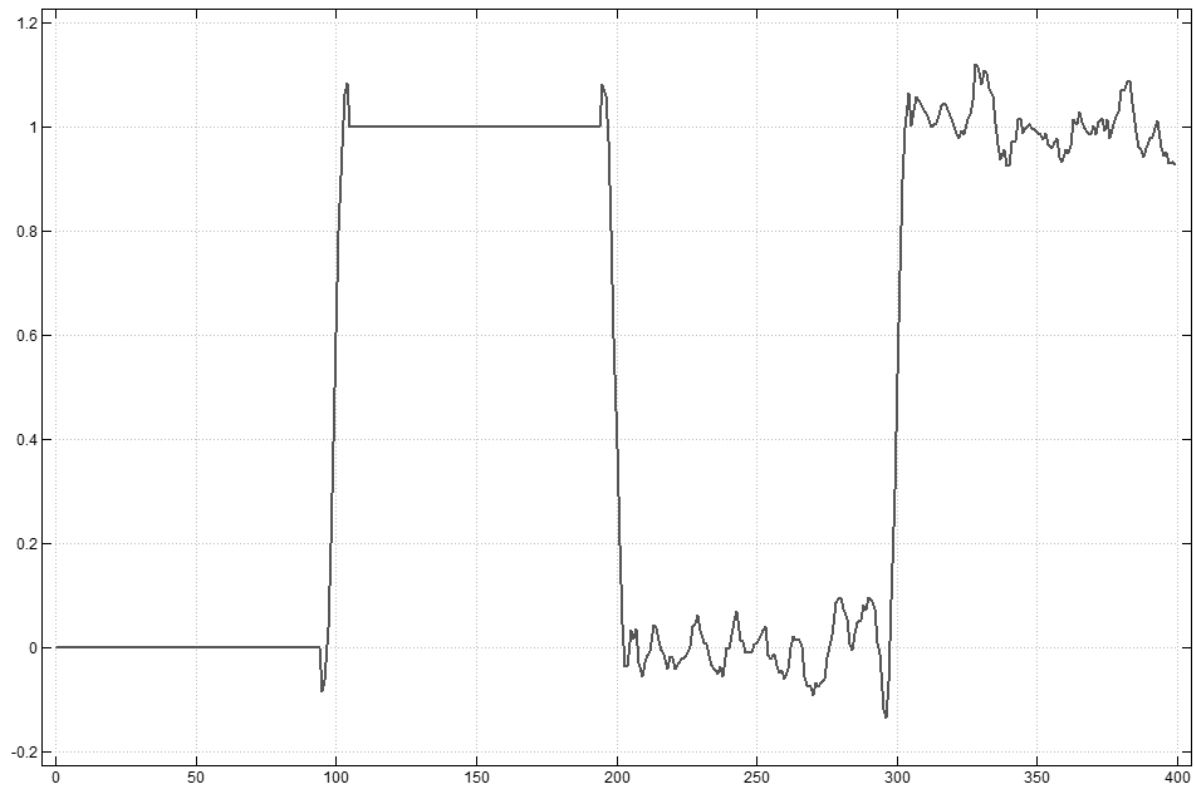
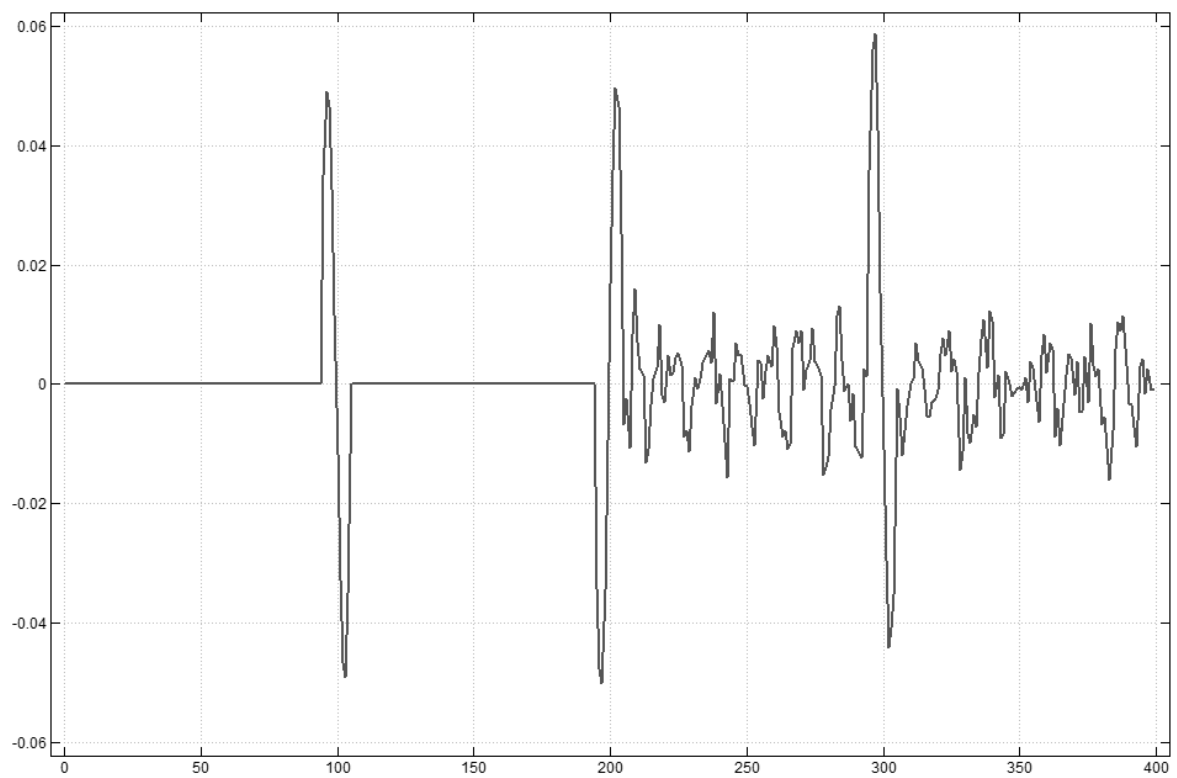
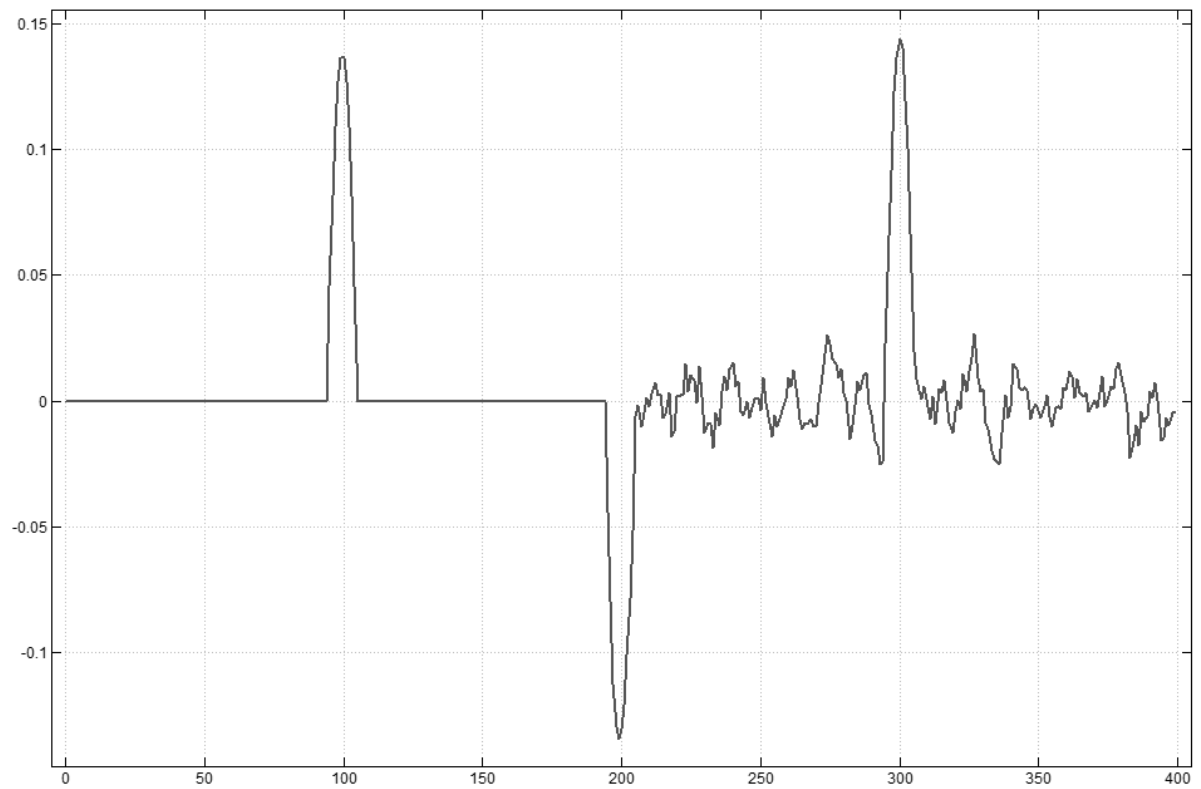


Figure 6: Smoothed with radius 5 and polynomials of degree 0, 2 and 4.

In Figure 7 the first four derivatives are calculated. Each one is calculated with a polynomial one order larger than the derivative and the smoothing window radius is set at 5 pixels. The filter computed first derivative of a step function is a Gaussian. The peak marks the location of the edge very accurately. The zero-crossings of even order derivatives mark the location of the edge. The peaks of odd order derivatives mark the location of the edge. The overshooting skirts of the odd order peaks are a part of the high-order derivative and can be used to help identify them in high noise. The location is preserved to a high degree of accuracy.

```
plot 1 2 SG 5 rst
plot 2 3 SG 5 rst
plot 3 4 SG 5 rst
plot 4 5 SG 5 rst
```

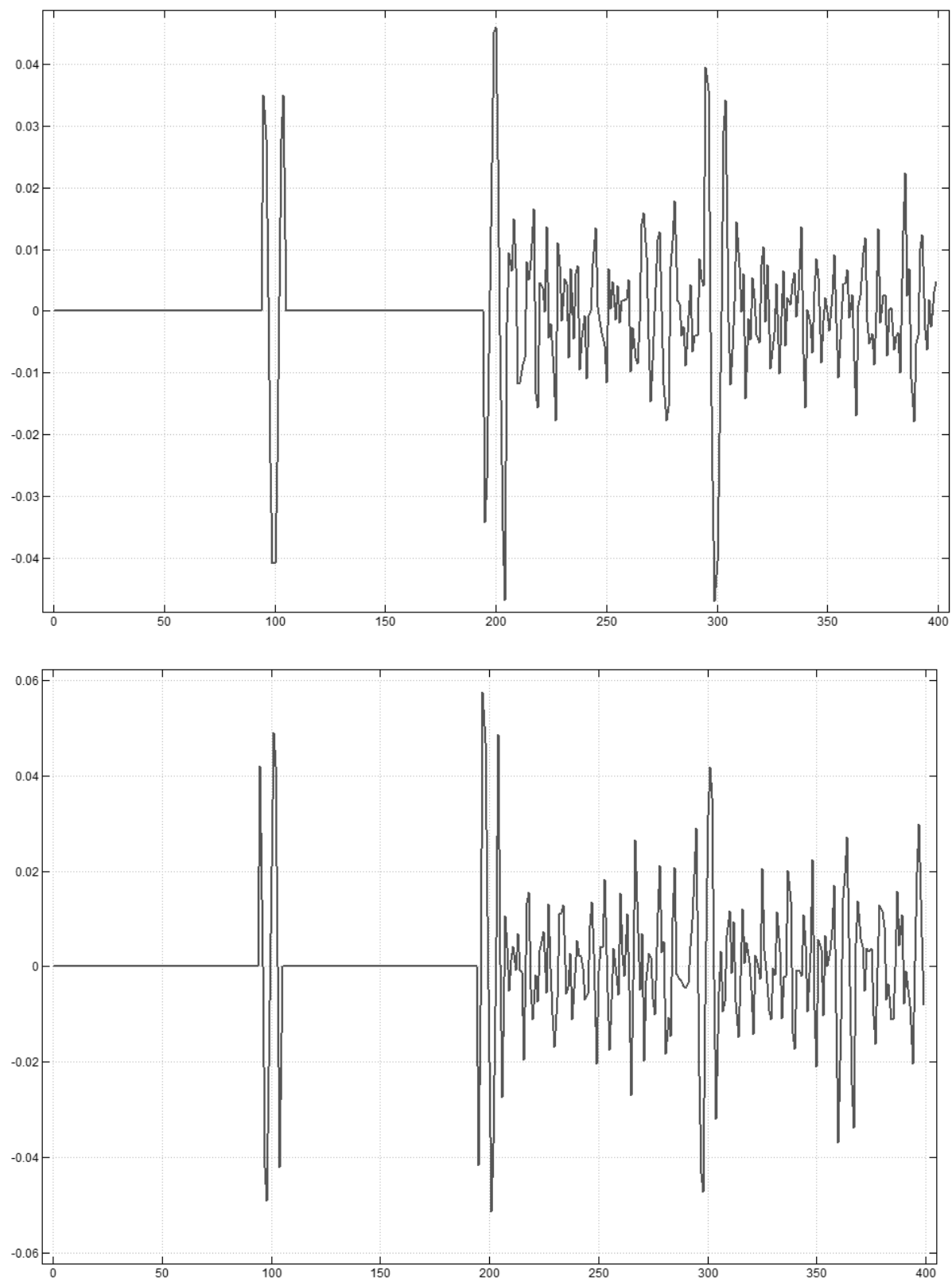


Figure 7: Derivatives of order 1, 2, 3 and 4 and polynomials one degree higher.

The higher derivatives shown in Figure 7 are almost lost in the noise. Readers are encouraged to retry these experiments using radii 10 and 15 to observe the higher derivatives more clearly while the precise location of the lower derivatives becomes less apparent.

We have seen that often using a polynomial of degree one higher than the order of the derivative is advantageous because ringing is minimized. However, we also see that in some circumstances, such as the previous section, one may need to use higher degree polynomials in order to retain high frequency information.

5 Filtering a Beaver Photo

We will consider the application of Savitzky-Golay filters involving values and first and second derivatives. We begin with values. The idea is very similar to the above data filtering except that we apply the filter in two dimensions of a raster image. For simplicity we will use a grayscale image. The *image3* addon contains *filter1.ijs* which has facilities for for converting between color models. In particular, the "Y" component of "YUV" space gives a good grayscale [5]. The image used below may be found at [6] although readers may want to experiment with their own images.

```
require '~addons/media/image3/color_space.ijs'

require '~addons/media/image3/view_m.ijs'

$b=:read_image 'C:/temp/dscf3950.jpg' NB. modify path/image
names
2736 3648 3

BW256 view_data g=:0{"1 RGB_to_YUV b
3648 2736
```

Figure 8 shows the resulting greyscale image. We use *view_data* with a greyscale palette since it automatically rescales data not in the to 255 range. The photo was taken in moderately adverse conditions: the evening light was dim, the beaver was moving, and there were high contrast reflections of sky and clouds.



Figure 8: A swimming beaver.

The first of the following filters produces a Savitzky-Golay filter using radius 7 (which gives 15 by 15 blocks of pixels) and degree 7 polynomials. The result is the array `sgg` which can be viewed. The other examples arise from cubic polynomials with radii 7 and 11 respectively. It is possible to see the improvement in the full scale image, but the differences are somewhat subtle. Instead, we pixel peek at two small portions of the image.

```
sgv=: 1 : 0
f=.(0,{.m) SG ({:m)
f f"_1 y
)

BW256 view_data sgg=:7 7 sgv g
3648 2736

zm1=:3 : '200 200{.600 500}.y'

spix=: [##"_1

BW256 view_data 5 spix g ,.&zm1 sgg
2000 1000
```

```
BW256 view_data 5 spix (3 7 sgv g) ,.&zm1 3 11 sgv g
2000 1000
```

First we peek at a reflection of the sky and clouds using `zm1` defined above to zoom into a small portion of the image. The function `spix` replicates pixels in blocks so they become apparent. Figure 9 compares the degree 7 polynomial with radii 7 filtered version with the original. The noise is noticeably reduced but the boundary between regions remains distinct, unmoved and slightly clarified. Figure 10 shows the same zoom but with radii 7 and 11 and degree 3 polynomials. Notice these give much more smoothing.

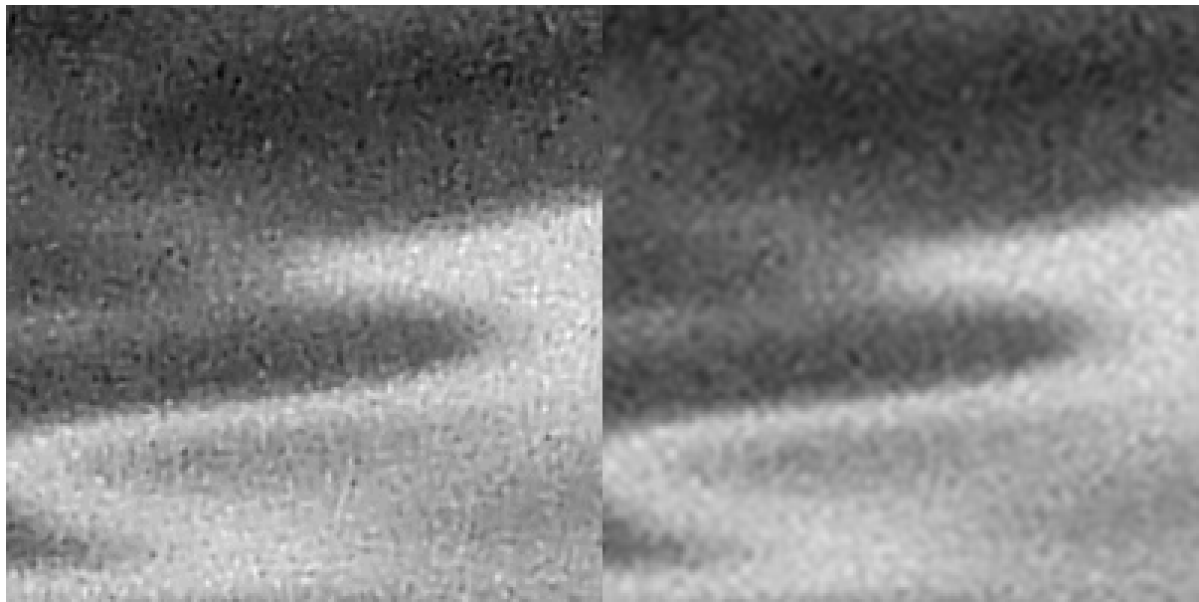


Figure 9: Cloud boundary before and after filtering.

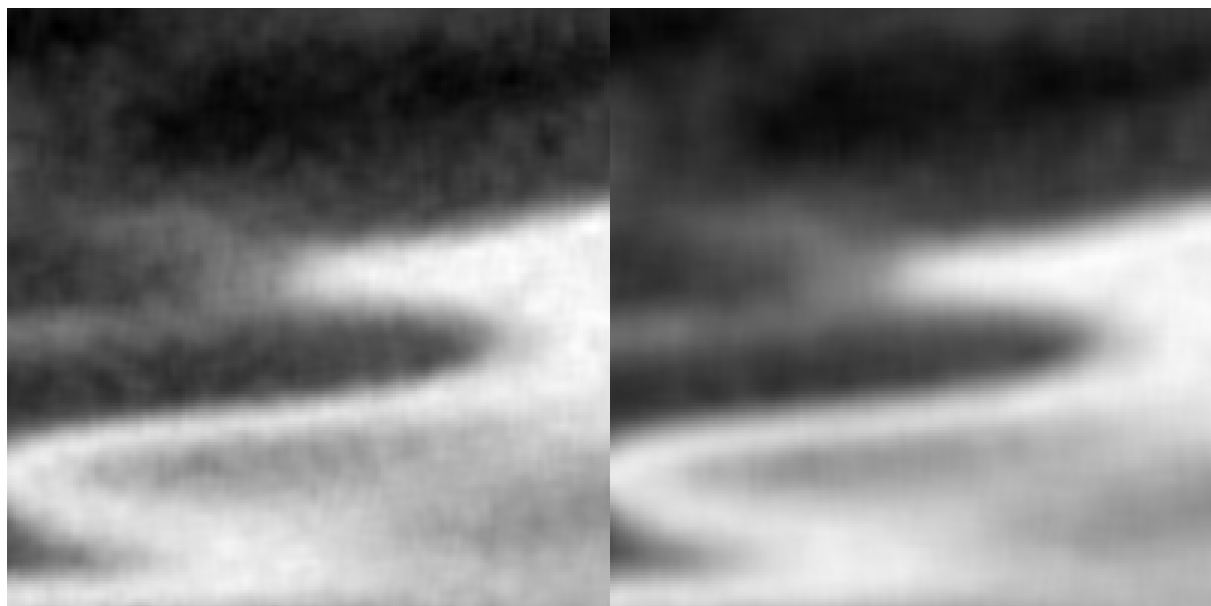


Figure 10: Cloud boundary after heavier filtering.

Similar zooms are used zoom into the edge of the beaver with water where whiskers are visible. In Figure 11 we observe that despite significant smoothing, the whiskers not only remain, but are slightly enhanced. There the degree 7 polynomial with radius 7 filtered version is compared with the original. Figure 12 shows the filters with radii 7 and 11 with degree 3 where damage to the whiskers is observed.

```
zm2=:3 : '200 200{.1050 1400}.y'

BW256 view_data 5 spix g ,.&zm2 sgg
2000 1000

BW256 view_data 5 spix (3 7 sgv g) ,.&zm2 3 11 sgv g
2000 1000
```

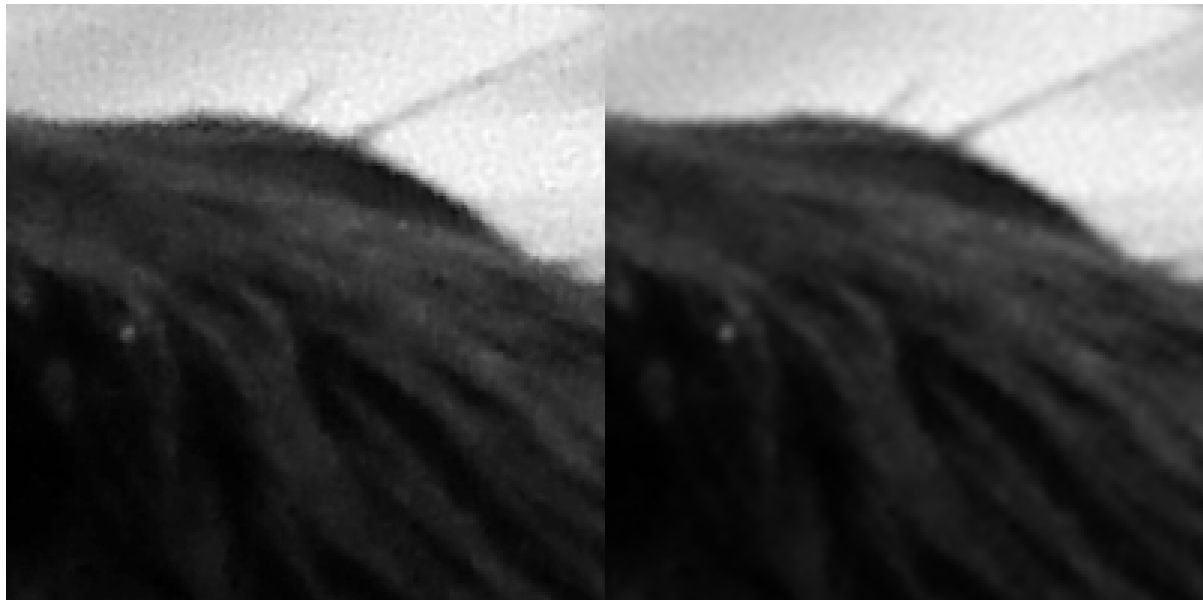


Figure 11: Beaver and water. Whiskers remain.



Figure 12: Beaver and water. Whiskers damaged by heavy filtering.

Thus we see that the desire to enhance different features may affect our choice of filter and no one filter is ideal for all purposes.

6 Sobel and Laplacian Filters based upon Savitzky-Golay

A classic method for edge detection is to use length of the gradient vector; namely, the square root of the sum of squares of the first partial derivatives as follows.

$$\sqrt{\left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}$$

Usually those partial derivatives are computed using divided differences of nearby pixels. The method is effective, but sensitive to noise in those pixels. We compute the derivatives using 7 by 7 Savitzky-Golay approximations and contrast that with the divided differences from [6]. In Figure 13 we see that the edges are distinctly highlighted. The divided difference version shows many noise artefacts.

```

lxy=:+&.*:
3 lxy 4
5
sob=:1 : 0
d=.(1,{.m) SG ({:m)

```

```

(d y)lxy d"_1 y
)

BW256 view_data 5 spix zm2 3 7 sob g
1000 1000

BW256 view_data 5 spix zm2 3 7 sob g
1000 1000

dx=: 1 2 1 */ 1 0 _1

]dy=: |:dx
1 2 1
0 0 0
_1 _2 _1

filt2=: *&dx +&. : *&(+/@,) *&dy

sobel=: 3 3&(filt2;._3)

BW256 view_data 5 spix zm2 sobel g
1000 1000

```

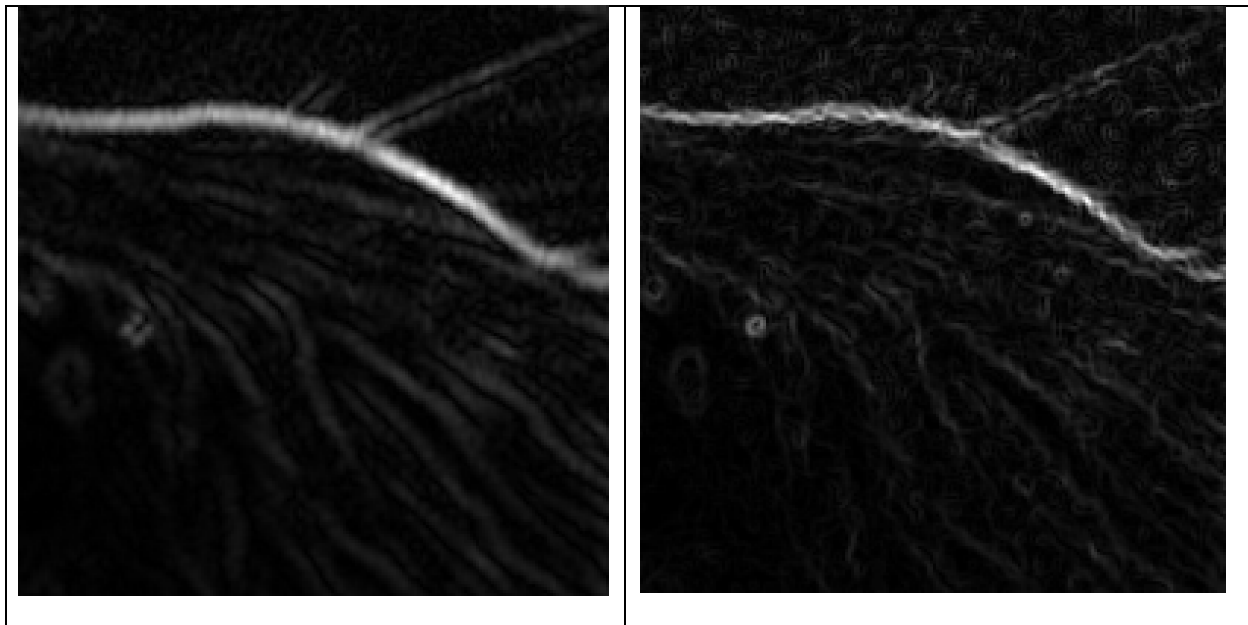


Figure 13: Sobel edges using Savitzky-Golay derivatives (left) and using classic divided differences (right).

The Laplacian is the sum of the second order derivatives with respect to the independent variables: $\partial^2 v / \partial x^2 + \partial^2 v / \partial y^2$. Again, these are traditionally estimated using nearby pixels and are susceptible to the noise in those pixels. We use Savitzky-Golay estimates to reduce the noise in the second order derivatives. In Figure 14 we see that it detects edges with an up-down band.

```

lap=:1 : 0
d=. (2,{.m) SG ({:m)
(d y)+d"_1 y

```


)

```

      BW256 view_data 5 spix  zm2 3 5 lap g
1000 1000

```

```

      BW256 view_data 5 spix  g ,.&zm2 g+2 2 lap g
2000 1000

```

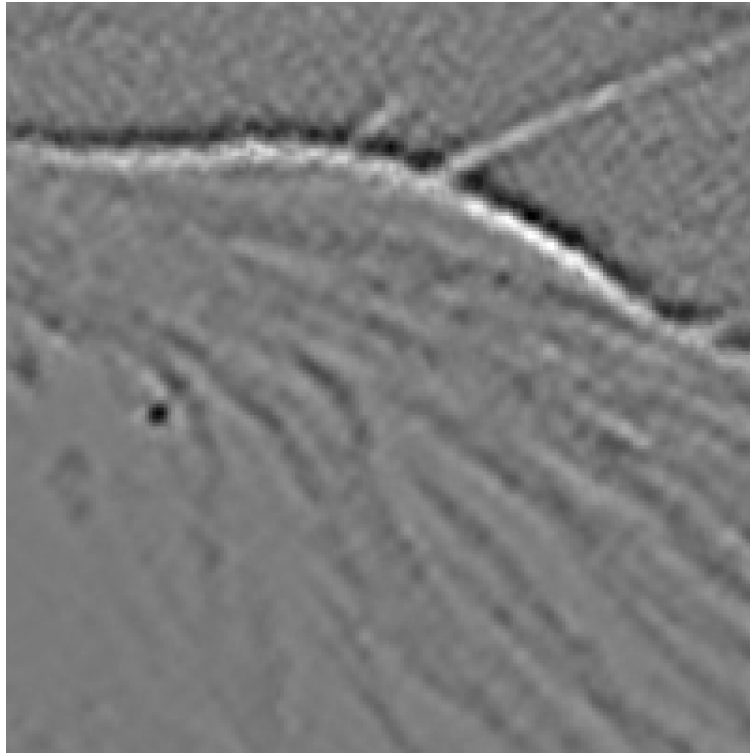


Figure 14: Laplacian shows edges with an up-down band.

One application of the Laplacian is that when it is added to the original image in some cases the edges can be clarified and noise reduced. Figure 15 shows the original data and the data added to a Laplacian filter with radius and degree 2. Notice that a remarkable amount of noise has been removed.

```

      BW256 view_data 5 spix  g ,.&zm2 g+2 2 lap g
2000 1000

```

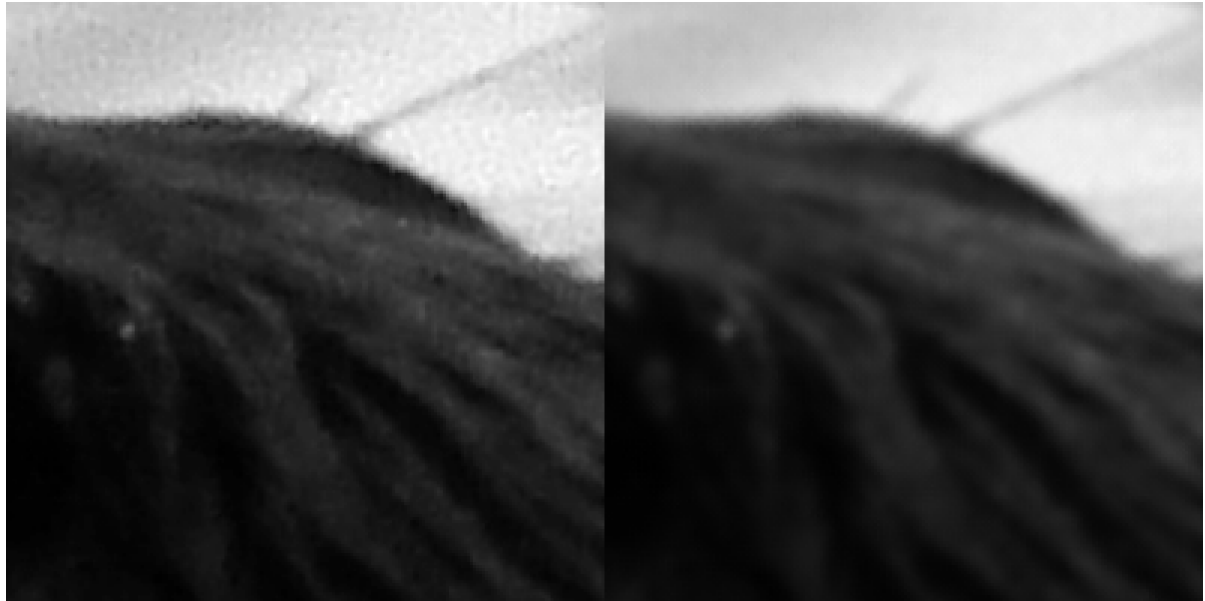


Figure 15: Image and image added to its Laplacian.

7 Image Processing a Disk

We finish by offering a few lines that interested readers could use to investigate these Savitzky-Golay image processing tools on an abstract image of a disk. Readers might enjoy trying to predict the result based upon the discussion earlier in this note.

```

    BW256 view_data dot=:1>2*!xy/~(i:%])500
1001 1001
    BW256 view_data 1 10 sgv dot
1001 1001
    BW256 view_data 1 30 sgv dot
1001 1001
    BW256 view_data 3 30 sgv dot
1001 1001

    BW256 view_data 2 15 sob dot
1001 1001
    BW256 view_data 5 15 sob dot
1001 1001
    BW256 view_data 2 2 sob dot
1001 1001

    zm3=:3 : '200 200{.500 600}.y'
    BW256 view_data 3 5 lap dot
1001 1001
    BW256 view_data 5 spix dot ,.&zm3 dot+3 2 lap dot
2000 1000

```

References

1. Jsoftware, J6.01c, with Image3 and FVJ3 addons, <http://www.jsoftware.com>, 2007.
2. Anthony J. Owen, Uses of Derivate Spectroscopy, <http://www.chem.agilent.com/Library/applications/59633940.pdf>.
3. William H. Press et al, Numerical recipes: the art of scientific computing, 3rd ed., Cambridge University Press, 2007.
4. Cliff Reiter, With J: Image Processing 1: Smoothing Filters, *APL Quote Quad* , 34 2 (2004) 9-15.
5. Cliff Reiter, With J: Image Processing 2: Color Spaces, *APL Quote Quad* , 34 3 (2004) 3-12
6. Cliff Reiter, Fractals, Visualization and J, 3rd ed., Lulu.com, 2007.
7. Cliff Reiter, Beaver image, <http://webbox.lafayette.edu/~reiterc/j/vector/index.html>.
8. John C. Russ, The image Processing Handbook, 5th edition, CRC Press, 2006.
9. A. Savitzky and M.J.E. Golay, M.J.E., Smoothing and Differentiation of Data by Simplified Least Squares Procedures, , *Analytical Chemistry* 36 8 (1964) 1627-1639.
10. Wikipedia, Savitzky-Golay smoothing filter http://en.wikipedia.org/wiki/Savitzky-Golay_smoothing_filter

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	___
Non-voting UK member (student/OAP/unemployed)	£10	___

Payment methods (*Sterling only*)

1. A Sterling cheque, payable to *British APL Association*, drawn on a UK bank.
2. By American Express, MasterCard or Visa:

I authorize you to debit my American Express/MasterCard/Visa account

Number: _____ Expires: ____/____

for the membership category indicated above.

Signature: _____ Date: _____

3. By electronic transfer.

Our account details are: Barclay's Bank; Cambridge, Chesterton Branch; Sort code: 20-17-35; Account number: 63955591; Account name: British APL Association; SWIFTBIC: BARCGB22; IBAN: GB86 BARC 2017 3563 9555 91.

4. Use PayPal to credit account treasurer@vector.org.uk (no account needed – ask for details).

If you pay by cheque or credit card, please send the completed form to:

BAA, c/o Nicholas Small, 12 Cambridge Road, Waterbeach, Cambridge CB25 9NJ

Privacy Policy
Your personal information
will be stored on computer
but not disclosed to third
parties. Card data will not be
stored on computer.