APL-Journal

A Programming Language

1-2/2024

APL-Germany e.V. Doppelnummer Nr. 1-2 2024 Jahrgang 43 ISSN - 1438-4531 RHOMBOS-VERLAG

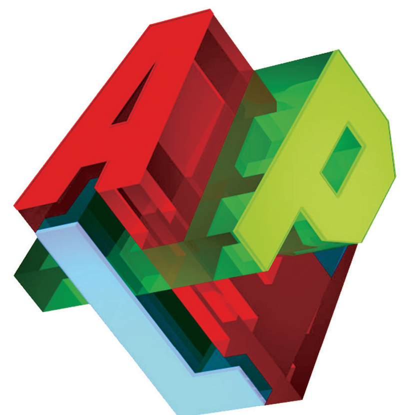
Jörg Hudelmaier
Euklids Elemente interaktiv

Bernd Ulmann
Analog- und Hybridrechnen im 21. Jahrhundert

Jürgen Sauermann
Neue Funktionen in GNU APL

Markos Mitsos
Migration from APL+Win to Dyalog

https://apl-germany.de/



Liebe Mitglieder von APL Germany,
liebe APL-Freunde,

heute erhalten Sie die neue Ausgabe des `APL-Journals` mit Beiträgen, die auf Vorträgen unserer beiden Tagungen des vergangenen Jahres in Mainz und Solingen basieren. Zu den Tagungen hatten sich wieder Teilnehmer aus dem In- und Ausland online zugeschaltet. Dank des Engagements von DPC und HBA war die Tagung in Solingen überdurchschnittlich besucht und das Programm breit aufgestellt und gut gefüllt!

Markos Mitsos berichtet über Probleme und Erfolge der Migration seiner Arbeitsbereiche von APL+Win zu Dyalog APL. Als leidenschaftlicher Anhänger von Analogrechnern beschreibt unser Gastredner Bernd Ulmann, Hochschule für Ökonomie und Management in Frankfurt, seine Ideen zum Einsatz von Analog- und Hybridrechnern im 21. Jahrhundert. Kai Jäger geht auf seine Arbeiten mit modernen Entwicklungswerkzeugen innerhalb GitHub ein. Jürgen Sauermann führt in die Besonderheiten seines GNU APLs ein und erklärt neue Funktionen. Und schließlich zeigt uns Jörg Hudelmaier einen interaktiven Zugang zu Euklids Elementen.

Unsere letzte Mitgliederversammlung fand am 30. April 2024 statt. Zu dem verschickten Protokoll gab es keine Einwände, es ist damit beschlossen. Die nächste Tagung mit Mitgliederversammlung und Vorstandswahlen findet am 27./28. März in Berlin-Adlershof statt. Das Protokoll der Mitgliederversammlung erhalten Sie wieder per E-Mail.

Ich wünsche Ihnen weiterhin viel Erfolg und gute Gesundheit und freue mich auf die Teilnahme vieler Mitglieder an den Tagungen.

Dieter Kilsch

Kontakt:

Prof. Dr. Dieter Kilsch
E-mail: d.kilsch@th-bingen.de

INHALT

Impressum

APL-Journal

43. Jg. 2024 ISSN 1438-4531

Herausgeber: Prof. Dr. Dieter Kilsch, APL-Germany e.V., Mainz, Homepage: https://apl-germany.de/, E-Mail: d.kilsch@th-bingen.de

Redaktion: Dipl.-Volksw. Martin Barghoorn (verantw.), E-Mail: Martin@Barghoorn.com

Verlag: RHOMBOS-VERLAG, Bernhard Reiser, Berlin, Postfach 67 02 17, D-10207 Berlin, Tel. (030) 261 9461, eMail: verlag@rhombos.de, Internet: https://rhombos.de/

Erscheinungsweise: halbjährlich

Erscheinungsort: Berlin

Druck: dbusiness.de GmbH, Berlin

Copyright: APL Germany e.V. (für alle Beiträge, die als Erstveröffentlichung erscheinen)

Fotonachweis: Martin Barghoorn (Umschlagseite 1 und 4)

APL-Germany e.V.

1. Vorstandsvorsitzender: Prof. Dr. Dieter Kilsch, Dumontstraße 12, 55313 Mainz, Tel. 06131 6982200, E-Mail: d.kilsch@th-bingen.de.

2. Vorstandsvorsitzender: Martin Barghoorn, Lückhoffstr. 8, 14129 Berlin, Tel. 030 80403192, E-Mail: Martin@Barghoorn.com

Schatzmeister: Jürgen Beckmann, Im Freudenheimer Grün 10, 68259 Mannheim, Tel. 0621 7 98 08 40, E-Mail: JBecki@onlinehome.de

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutzgesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen. Eine Haftung für die Richtigkeit der veröffentlichten Informationen kann trotz sorgfältiger Prüfung von Herausgeber und Verlag nicht übernommen werden.
Mit Namen gekennzeichnete Artikel geben nicht unbedingt die Meinung des Herausgebers oder der Redaktion wieder. Für unverlangte Einsendungen wird keine Haftung übernommen. Nachdruck ist nur mit Zustimmung des Herausgebers sowie mit Quellenangabe und Einsendung eines Beleges gestattet. Überarbeitungen eingesandter Manuskripte liegen im Ermessen der Redaktion.

Editorial 1

Analog- und Hybridrechnen im 21. Jahrhunderthdt 3

Euklids Elemente interaktiv 12

Migration from APL+Win to Dyalog 18

Neue Funktionen in GNU APL 38

Development in Dyalog APL with Modern Tool 42

Impressum 2

Analog- und Hybridrechnen im 21. Jahrhundert

Bernd Ulmann*

ulmann@anabrid.com

1 Einleitung

So beeindruckend die Rechenleistung moderner Digitalrechner ist, wird dennoch zunehmend deutlich, dass klassische (speicherprogrammierte) Digitalrechner an physikalische Grenzen stoßen, die nur durch Verwendung anders gearteter Berechnungsparadigmata überwunden werden können, um die unaufhaltsam steigenden Anforderungen an die verfügbare Rechenleistung nicht nur in naher, sondern auch ferner Zukunft erfüllen zu können. Hauptprobleme moderner Digitalrechner sind vor allem der hohe Energiebedarf (gerade im Bereich künstlicher Intelligenz ist dieser Aspekt bereits jetzt ein limitierender Faktor, vor allem im Zusammenhang mit dem Training künstlicher neuronaler Netzwerke), die auf wenige GHz beschränkten Taktfrequenzen (diese sind im Wesentlichen einerseits durch den superlinear mit der Taktfrequenz steigenden Energiebedarf, andererseits durch physikalische Eigenschaften der verwendeten Halbleitermaterialien beschränkt und bewegen sich seit nunmehr fast 20 Jahren im Bereich um 3 GHz herum) sowie die beschränkte Integrationsdichte (mit gegenwärtig 2 nm technology node stößt man hier bereits langsam an physikalische Grenzen). Weitere Herausforderungen im Bereich des klassischen Digitalrechnens sind die Schwierigkeiten, hochgradigen Parallelismus bei Berechnungen zu erzielen (Amdahls Law) sowie die Tatsache, dass letztlich nur ein (sehr) kleiner Teil der Transistorfunktionen eines solchen Computers für wirkliche Rechenoperationen (ALU) genutzt wird, während der Großteil, selbst von Speichern bzw. Caches abgesehen, für eine überbordend komplexe Ablaufsteuerung, Interfaces, Datentransfers etc. erforderlich ist.

Entsprechend rücken andere Ansätze zur Lösung bestimmter Problemklassen in den Fokus. An erster Stelle ist hier sicherlich das Quantencomputing zu nennen, das – zumindest perspektivisch – grundlegende Vorteile gegenüber klassischen algorithmischen Lösungsverfahren verspricht. Eine weitere, bislang oftmals übersehene Rechentechnik ist das Analog- und eng damit verbunden das Hybridrechnen. Im Unterschied zu Quantencomputern, bei denen es sich im Wesentlichen noch immer um Laborsysteme handelt, die nicht nur vergleichsweise weit von einer einfachen, kostengünstigen und flächendeckenden Anwendung entfernt sind, sondern immer noch grundlegende Probleme aufweisen, für die Lösungen noch ausstehend sind (in diesem Zusammenhang ist vor allem das Problem der Fehlerkorrektur zu nennen), erfordern Analogrechner keine grundlegenden wissenschaftlichen Durchbrüche, um Serienreife zu erlangen, da die grundlegenden Technologien gut verstanden und technisch leicht beherrschbar sind. Dazu kommt, dass Lösungsverfahren für Analogrechner deutlich einfacher als für Quantencomputer zu entwickeln und zu implementieren sind. Interessanterweise zeigte sich in den letzten Jahren, dass bestimmte Problemklassen, die gemeinhin eher (adiabatischen) Quantencomputern zugeordnet werden, auch mit

*Hochschule für Oekonomie und Management, Frankfurt/Main sowie anabrid GmbH
ulmann@anabrid.com

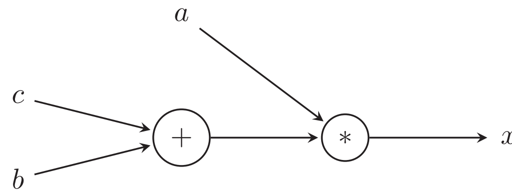


Abbildung 1: Berechnung von $(a+b)c$ auf einem Analogrechner

Hilfe deutlich einfacherer Analogrechner (oszillator basierte ISING-Maschinen, siehe unten) behandelt werden können.

2 Analog- und Hybridrechnen

Der Begriff des *Analogrechnens* ist etwas unglücklich, da „analog" heutzutage oftmals mit „veraltet" assoziiert wird, was im Zusammenhang mit dem Analogrechnen nicht weiter von der Realität entfernt sein könnte. Entsprechend wichtig ist eine saubere Definition dessen, was einen Analogrechner auszeichnet:

Ein speicherprogrammierter Digitalrechner besitzt eine feste interne Struktur und löst Probleme durch Abarbeiten eines Algorithmus, d. h. einer Sequenz einfacher Instruktionen, die aus einem Speicher gelesen werden. Die Lösungszeit für Probleme hängt vom Algorithmus und der Problemgröße ab und wird in der Regel durch LANDAU-Symbole („Big-O"-Notation) beschrieben. Letztlich ist ein solcher Rechner in der Lage, Komplexität gegen Zeit zu „tauschen", d. h. komplexere Probleme/größere Datenvolumina resultieren in längeren Lösungszeiten.

Im Unterschied hierzu besitzt ein Analogrechner nicht nur keinen Speicher, sondern arbeitet auch grundlegend nicht-algorithmisch. Er besteht aus einer Vielzahl einzelner *Rechenelemente*, wie beispielsweise Summierer, Integrierer, Multiplizierer, Koeffizientenglieder etc., die miteinander verschaltet werden, um somit ein – in der Regel elektronisches – *Modell*, ein *Analogon*, für ein zu lösendes Problem zu implementieren. Das bedeutet insbesondere, dass ein solches System mindestens so viele Rechenelemente umfassen muss, wie für den Aufbau eines solchen Modells benötigt werden, d. h. die Größe eines Analogrechners wächst mit $\mathcal{O}(N)$, wobei N die Anzahl der Operationen in den einem Problem zugrundeliegenden Differentialgleichungen bezeichnet. Dies ist durchaus ein Nachteil gegenüber einem Digitalrechner mit seiner festen Struktur, bringt aber einen grundlegenden und massiven Vorteil hinsichtlich der Lösungszeit mit sich, die im Wesentlichen konstant, d. h. $\mathcal{O}(1)$ ist.

Soll beispielsweise der Ausdruck $(a+b)c$ ausgewertet werden, erfolgt dies auf einem klassischen Digitalrechner in sechs Instruktionen (drei zum Laden der Operanden a, b und c, eine Addition, eine Multiplikation sowie eine Store-Instruktion), die nicht parallel (sieht man von – zum Teil – überlappender Ausführung einmal ab) ausgeführt werden können. Ein entsprechendes Setup eines Analogrechners ist in Abbildung 1 dargestellt. Variablen werden in einem Analogrechner in der Regel durch Spannungen oder Ströme repräsentiert, so dass eine Verbindung zwischen zwei Rechenelementen lediglich eine Leitung erfordert. Da keinerlei Speicher vorhanden ist, arbeiten die Rechenelemente in perfektem Parallelismus (ohne an dieser Stelle auf Effekte, wie sie aus der endlichen Bandbreite der verwendeten Rechenelemente resultieren, einzugehen).

Ein Analogrechner wird also programmiert, indem ein zu lösendes Problem, das in der Regel in Form gekoppelter Differentialgleichungen (DGLs) vorliegt, in eine äquivalente *Rechenschaltung* umgewandelt wird. Bei klassischen Analogrechnern, die allerdings ihren Platz zu Recht in Museen gefunden haben, werden die Rechenelemente mit Hilfe von Kabeln manuell über ein – meist zentrales – Patchfeld

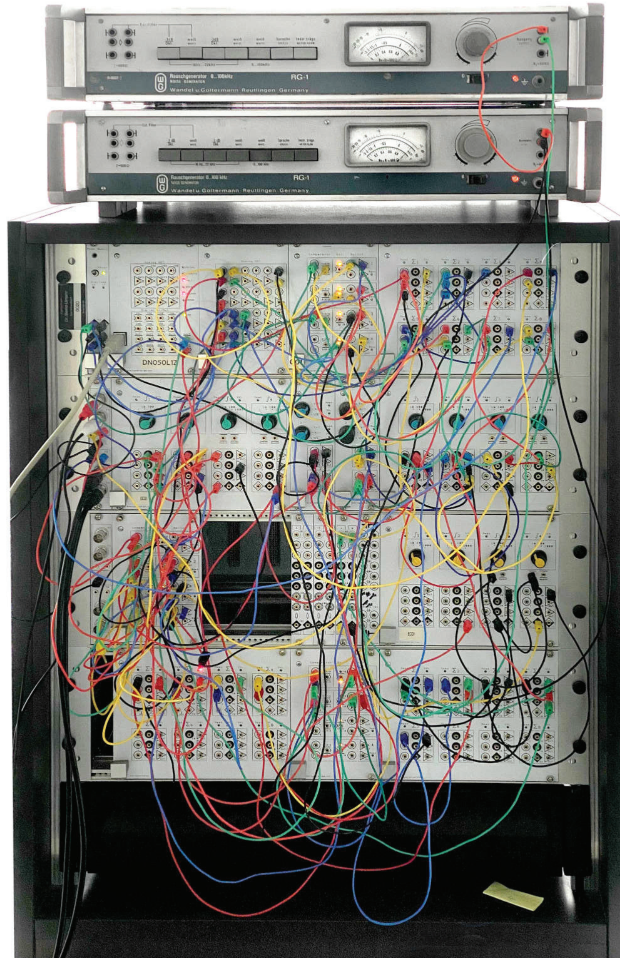


Abbildung 2: Klassisch programmiertes Analogrechnersystem

miteinander verbunden. Moderne Analogrechner verfügen anstelle dieses Verbindungsfeldes über Kreuzschienenverteiler, die, gesteuert durch einen mit dem Analogrechner gekoppelten Digitalrechner, die erforderlichen Verbindungen herstellen, so dass die Programmierung eines Analogrechners aus der digitalen Domäne heraus erfolgen kann und keinen Medienbruch mit sich bringt.

Abbildung 2 zeigt exemplarisch die Programmierung eines klassisch aufgebauten Analogrechnersystems. Die einzelnen Rechenelemente werden hierbei mit Patchkabeln verbunden und bilden so das elektronische Modell des zu lösenden Problems. Diese Art der Programmierung skaliert offensichtlich nicht gut und ist zeitaufwändig, was einen großen Nachteil klassischer Analogrechner darstellt.

Ein etwas komplexeres Beispiel, die VAN DER POL-Gleichung, mag das Grundprinzip der Programmierung eines Analogrechners verdeutlichen.[1] Zu lösen ist

$$\ddot{y} + \mu \left(y^2 - 1\right) \dot{y} + y = 0,$$

wobei y, $\dot{y}$ und $\ddot{y}$ die gesuchte, von der Zeit abhängige Lösungsfunktion, sowie deren erste und zweite Ableitung darstellen. μ ist ein freier Parameter. Die übliche Vorgehensweise, solche Probleme zu lösen, geht auf Lord KELVIN zurück, der in den 1870er Jahren ein heute als KELVIN*sches Rückkopplungsverfahren* bekanntes Vorgehen entwickelt hat. Hierzu wird die zu lösende Gleichung zunächst nach der

[1] Eine umfassende Einführung in die Programmierung von Analog- und Hybridrechnern findet sich in [ULMANN 2023].

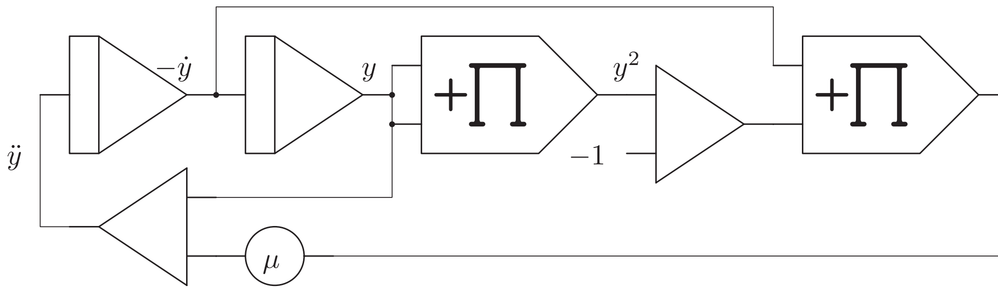


Abbildung 3: Rechenschaltung zu Gleichung (1)

höchsten Ableitung der gesuchten Funktion aufgelöst:

$$\ddot{y} = -y - \mu \left(y^2 - 1\right) \dot{y}. \tag{1}$$

Nun werden, ausgehend von der (noch) unbekannten Funktion $\ddot{y}$ alle niedrigeren Ableitungen von y durch fortgesetzte (Zeit-)Integration bestimmt. Hierzu werden im vorliegenden Fall zwei Integrierer (grundlegende Rechenelemente eines jeden Analogrechners) benötigt, an deren Ausgängen dann $-\dot{y}$ bzw. y zur Verfügung stehen.[2] Mit Hilfe dieser Funktionen $\dot{y}$ und y wird im vorliegenden Fall nun der Ausdruck auf der rechten Seite von Gleichung (1) gebildet, der ja gerade gleich $\ddot{y}$ ist. Dieser Ausdruck bildet also das Eingangssignal für den ersten Integrierer dieser Kette aus zwei Integrierern, so dass sich die in Abbildung 3 dargestellte Rechenschaltung ergibt.

Bei den beiden Rechenelementen links oben handelt es sich um Integrierer, an deren Ausgängen jeweils $-\dot{y}$ und y zur Verfügung stehen. Mit Hilfe eines Multiplizierers wird y^2 gebildet, das dann mit Hilfe eines Summierers (das dreieckige Symbol) in den Ausdruck $1 + y^2$ (Vorzeichenwechsel) eingeht. Mit Hilfe eines weiteren Multiplizierers wird hieraus $\dot{y}(y^2 - 1)$ erzeugt, das, multipliziert mit der Konstanten μ (ein Kreis kennzeichnet ein sogenanntes *Koeffizientenglied*), mit Hilfe eines weiteren Summierers den Ausdruck $-y-\mu(y^2-1)\dot{y}$ ergibt. Dieser Ausdruck ist gemäß der zugrunde liegenden Gleichung gleich $\ddot{y}$ und wird an den Eingang des ersten Integriers angeschlossen, womit die Rechenschaltung vollständig ist.

Zugegebenermaßen ist diese Rechenschaltung noch nicht ganz vollständig, so wurden beispielsweise die erforderlichen Anfangswerte der Integrierer ebenso wenig berücksichtigt, wie die in der Regel unerlässliche *Skalierung* der Gleichung(en). Eine solche Skalierung ist erforderlich, da Variablen als Spannungen (seltener als Ströme) dargestellt werden, die, technisch bedingt, stets in einem, durch die jeweilige Implementation des Analogrechners vorgegebenen Intervall, liegen, das gemeinhin als das Intervall $[-1, 1]$ betrachtet wird, unabhängig von den tatsächlich verwendeten Spannungen, um die Skalierung zu vereinfachen.[3]

Im Unterschied zu den heute dominierenden Digitalrechnern handelt es sich bei Analogrechnern um Spezialmaschinen, die vor allem für die Lösung von Problemen, die durch gekoppelte Differentialgleichungen oder durch partielle Differentialgleichungen beschrieben werden können, geeignet sind. Für Probleme dieser Klasse weisen Analogrechner Rechengeschwindigkeiten und nicht zuletzt eine Energieeffizienz auf, die z. T. mehrere Zehnerpotenzen über denen moderner Digitalrechner liegen.

[2] Aus Gründen der Implementation nehmen Integrierer und Summierer in Analogrechnern in der Regel eine implizite Vorzeichenumkehr vor.

[3] In der Frühzeit elektronischer Analogrechner waren meist Rechenspannungen im Bereich von ± 100 V zulässig. Spätere transistorisierte Systeme nutzten Spannungen von ± 10 V, während moderne hochintegrierte Analogrechner in CMOS-Technologie meist mit weniger als ± 1 V arbeiten.

Idealerweise betrachtet man einen modernen Analogrechner als spezialisierten Co-Prozessor für einen Digitalrechner, ähnlich dem Einsatz von GPUs, wobei der Digitalrechner bestimmte Probleme auf den analogen Co-Prozessor auslagert, indem er ihn parametrisiert, konfiguriert und das Ergebnis nach kurzer Zeit abholt.[4] Eine solche enge Kopplung eines Digitalrechners mit einem als Co-Prozessor agierenden Analogrechners wird als *Hybridrechner* bezeichnet.

Die zentralen Vorteil von Analogrechnern sind neben dem vollkommenen Parallelismus der Rechenelemente, die extrem hohe Energieeffizienz sowie die sehr hohe Rechengeschwindigkeit, die deutlich über jenen moderner Digitalrechner liegen. Diese Vorteile kommen jedoch nicht ohne gewisse Nachteile mit sich zu bringen, die jedoch in der Praxis in der Regel weit weniger schwerwiegend sind, als es auf den ersten Blick der Fall zu sein scheint.

Zunächst ist die prinzipiell auf ca. drei bis vier Dezimalstellen beschränkte Rechengenauigkeit zu nennen, was aber für die Mehrzahl der Probleme (vor allem auch im Bereich künstlicher neuronaler Netze) keine grundlegendes Problem darstellt. Auch die erforderliche Problemskalierung ist bei der Programmierung zu berücksichtigen, was allerdings mit Hilfe hybrider Rechenansätze auch automatisiert werden kann. Weiterhin steht bezüglich der Integration nur die Zeit als unabhängige Variable zur Verfügung, was besondere Verfahren zur Lösung partieller Differentialgleichungen erforderlich macht.

Nun handelt es sich bei Analogrechnern an sich um kein neuartiges Konzept – Analogrechner blicken auf eine sehr viel längere Vergangenheit zurück als Digitalrechner, und hatten einen wesentlichen Einfluss auf die technologische Entwicklung im 20. Jahrhundert, nicht zuletzt im Bereich der Luft- und Raumfahrttechnik, des Automobilbaus, der Physik, Chemie, Medizin uvm.[5] Dennoch sind sie in den vergangenen Jahrzehnten nahezu in Vergessenheit geraten, während Digitalrechner in den Vordergrund rückten. Dieses Verschwinden hatte eine Reihe von Gründen, die allerdings mit modernen Technologien überwunden werden können, was zu einer Renaissance des Analogrechnens führen wird.

Hauptprobleme klassischer Analogrechner waren zunächst die manuelle Programmierung, die sehr zeitaufwändig und nicht zuletzt auch fehlerträchtig war, aber auch die Unmöglichkeit, solche Systeme in einem Time-Sharing-Betrieb einer Reihe von Nutzern quasi-simultan zur Verfügung zu stellen. Ein für ein bestimmtes Problem konfigurierter Analogrechner musste (zeit-)aufwändig umkonfiguriert werden, um ein anderes Problem zu lösen, was zu Programmwechselzeiten im Bereich von Minuten oder gar Stunden führte. Erschwerend kam hinzu, dass Analogrechner den rapiden Preisverfall von Digitalrechnern, der bereits in den 1970er Jahren einsetzte, seinerzeit nicht mitmachen konnten, da sie auf Präzisionsbauelemente, eine komplexe Verdrahtung des Patchfeldes etc. angewiesen waren. Obwohl klassische Analogrechner ihren digitalen Verwandten bis weit in die 1980er hinein hinsichtlich der reinen Rechenleistung deutlich überlegen waren, überwogen diese Nachteile dennoch in so großem Maße, dass sie bis ca. Mitte der 1980er Jahre fast vollständig durch Digitalrechner ersetzt wurden.

Mit moderner CMOS-Technologie können diese Probleme jedoch als überwunden betrachtet werden: Das manuelle Patchfeld ist hochintegrierten Kreuzschienenverteilern gewichen, die in Mikrosekunden umkonfiguriert werden können. An die Stelle einer Vielzahl manuell einzustellender Zehngangpotentiometer sind multiplizierende Digital-Analog-Wandler getreten. Die Notwendigkeit, hochpräzise Bauelemente – vor allem Widerstände (Summierer und Integrierer) und Kondensatoren

[4] Die Lösungszeiten von Analogrechnern sind mitunter so hoch, dass allein die Interruptlatenz des Digitalrechners, d.h. die Zeitspanne, die erforderlich ist, um auf eine externe Unterbrechungsanforderung wie die des Analogrechners, dass ein Ergebnis zur Verfügung steht, zu reagieren, zu einem Engpass wird.

[5] Siehe [Ulmann 2023].

(Integrierer) – zu verwenden, kann durch eine automatische Kalibration, gesteuert durch einen Digitalrechner, umgangen werden.

Verfügten mittlere bis große klassische Analogrechner in der Regel über einige Dutzend bis einige hundert Rechenelemente, sind mit moderner CMOS-Technologie Systeme, bestehend aus tausenden, zehntausenden oder hunderttausenden von Rechenelementen, kein grundlegendes Problem mehr.[6]

Zentrale Anforderungen an moderne Analog- und Hybridrechner sind die Folgenden:

- Die Programmierung muss in Form einer *domain specific language* (*DSL*) erfolgen, die möglichst nahtlos in aktuelle Programmiersprachen und -Werkzeuge eingebunden werden kann. Aus Programmierersicht muss der Analogrechner quasi transparent sein, d. h. seine Programmierung darf kein Wissen über seine konkrete Implementation voraussetzen, um ihn als allgemein einsetzbaren Co-Prozessor einzusetzen.
- Es werden Bibliotheken zur Einbindung eines Analogrechners in ein Hybridrechnersetup benötigt, wobei der Kommunikationsoverhead zwischen Digital- und Analogrechner möglichst gering gehalten werden muss. Interruptlatenzzeiten in der Größenordnung einiger Mikrosekunden liegen häufig bereits in der gleichen Größenordnung wie die eigentlichen Lösungszeiten des Analogrechners.

3 Moderne Anwendungsgebiete

Eine zentrale Frage ist die nach möglichen und vor allem wirtschaftlich relevanten Anwendungsgebieten moderner Analog- und Hybridrechner, die im Folgenden kurz dargestellt werden.

3.1 High Performance Computing

An erster Stelle steht hier sicherlich das High Performance Computing. Die Mehrzahl der Probleme in diesem Bereich werden durch gekoppelte bzw. partielle Differentialgleichungen beschrieben und sind damit hervorragend für Analogrechner geeignet. In Bereichen wie der Strömungsdynamik versprechen Analogrechneransätze sogar Lösungsmöglichkeiten für Probleme, die algorithmisch gegenwärtig ausserhalb der Reichweite selbst der leistungsfähigsten Supercomputer liegen.

3.2 KI

Vor allem der Bereich der *künstlichen Intelligenz*, um den heute ja kein Weg herum zu führen scheint, kann stark von analogen Rechentechniken profitieren. Besonders deutlich wird dies bei künstlichen neuronalen Netzen (*ANN*s), deren natürliche Vorbilder, ähnlich Analogrechnern, ebenfalls aus einer Vielzahl geeignet verschalteter Rechenelemente bestehen, die zu 100% parallel arbeiten. Im einfachsten Fall können solche ANNs mit ihren synaptischen Gewichten beispielsweise als Matrizen veränderlicher Widerstände implementiert werden, die synaptische Gewichte darstellen, wobei die Summe über solchermaßen gewichtete Eingangssignale über Stromsummation, d. h. die *Kirchhoff* sche Knotenregel erfolgt.[7]

[6] Erste Ansätze im akademischen Bereich gehen im Wesentlichen auf [Cowan(2005)] und [Guo et al.(2016)] zurück.

[7] Solche Ansätze werden meist als *in memory computing* bezeichnet und beispielsweise von Unternehmen wie MythicAI (`https://mythic.ai`, Aspinity (`https://www.aspinity.com`) oder auch IBM (`https://research.ibm.com/blog/the-hardware-behind-analog-ai`) verfolgt.

Als programmierbare Widerstände werden in diesem Zusammenhang gegenwärtig sowohl Flash-Speicherzellen als auch *Phase Change Memory*-Speicher (*PCM*) bzw. *Memristoren* untersucht. Mit Hilfe analoger Rechentechniken und ihrer hohen Energieeffizienz werden auch ANNs on-the-edge bzw. in always-on-devices möglich.

3.3 Medizinische Anwendungen

Medizinische Anwendungen wie beispielsweise Herz- oder Hirnschrittmacher, monitoring devices etc. profitieren ebenfalls vor allem von der hohen Energieeffizienz analoger Rechenschaltungen. Denkbar sind in naher Zukunft Implantate, die ihren Energiebedarf mit Hilfe von energy harvesting direkt aus dem Körper des Trägers decken, und keine externen Energiequellen, umständliche Ladeverfahren etc. erfordern.

3.4 Industrielle Systeme

Im Bereich industrieller Systeme stehen ebenfalls die Energieeffizienz sowie die Robustheit und hohe Rechengeschwindigkeit von Analogrechnern im Vordergrund. Wurden in den Jahrzehnten nach 1970 die damals vorherrschenden analogen Mess-, Steuer- und Regelungssysteme vollständig durch Digitalrechner abgelöst, ist in Zukunft eine Umkehrung dieses Trends zu erwarten. Hierbei sind besonders die „Unhackbarkeit“ von Analogrechnern im klassischen Sinn (kein Speicher, kein Algorithmus, d. h. übliche Angriffsvektoren verfangen hier nicht) sowie die hohe Energieeffizienz von Vorteil, wobei letztere einen Einsatz direkt in Sensoren und Aktuatoren ermöglicht (edge computing).

3.5 Signalvor- und -nachverarbeitung

Gerade in Mobilgeräten wie beispielsweise Smartphones ist ein nicht unerhebliches Maß an Signalvor- und -nachverarbeitung erforderlich, das mit Hilfe analoger Rechentechniken deutlich energiesparender umgesetzt werden kann als mit algorithmischen Ansätzen. Ein konfigurierbares Bandpassfilter ist beispielsweise sehr viel einfacher und vor allem energieeffizienter analog als digital zu implementieren.[8] In diesen Bereich fällt auch die Erkennung von Triggerwörtern bei always-on-devices, die Erkennung von Glasbruchgeräuschen etc., die ebenfalls rein analog durchgeführt werden können.[9]

3.6 Monte-Carlo Simulationen

Eine Vielzahl von Problemen lässt sich mit Hilfe stochastischer Ansätze, sogenannten *Monte-Carlo Verfahren* lösen. Auch hier sind Analogrechner prädestiniert, da in Form physikalischer Rauschquellen hervorragende Entropiequellen zur Verfügung stehen, mit deren Hilfe Simulationen durchgeführt werden können, wobei die Auswertung der Resultate auf dem Digitalteil eines Hybridrechners erfolgt.[10]

3.7 Optimierungsprobleme etc.

Auch Optimierungsprobleme können mit Hilfe von Analogrechnern gelöst werden, wobei eine der vermutlich interessantesten Entwicklungen der letzten Jahre auf

[8] Siehe beispielsweise [Anadigm(2006)].

[9] Aspinity ist hier einer der zentralen Anbieter.

[10] Ein einfaches Beispiel anhand einer stochastischen Differentialgleichung findet sich hier: `https://analogparadigm.com/downloads/alpaca_50.pdf`.

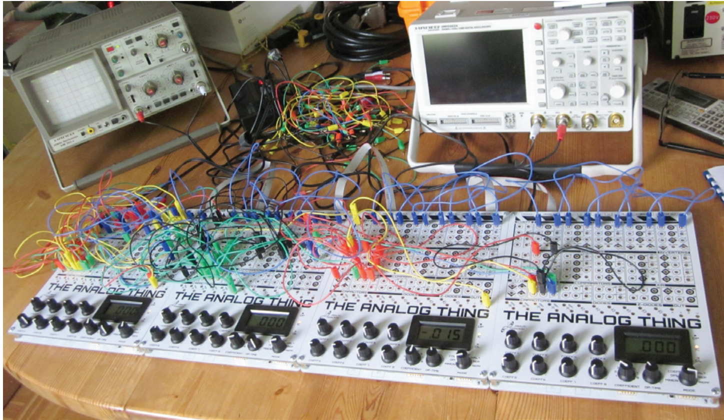


Abbildung 4: Einsatz von Analogrechnern in der Lehre

oszillatorbasierten Ising-Maschinen beruht, mit deren Hilfe beispielsweise das NP-vollständige Max-Cut-Problem gelöst werden kann.[11] Auf solchen Maschinen lässt sich eine Vielzahl von Problemen abbilden, siehe beispielsweise [Lucas(2014)]. Aufgrund ihrer Nähe zu adiabatischen Quantencomputern wird diese Systemklasse oftmals als *quantum inspired* bezeichnet.

3.8 Ausbildung und Lehre

Analogrechner sind strukturell deutlich näher an dem zu lösenden Problem als Digitalrechner, da Problemgleichungen nach wenig Übung direkt in Rechenschaltungen umgesetzt werden können, an denen sich beispielsweise die Auswirkungen von Parameteränderungen durch manuelles Verstellen von Potentiometern direkt an einem angeschlossenen Oszilloskop beobachten lassen. Studenten profitieren stark von diesem anschaulichen und im wahrsten Wortsinne begreifbaren Ansatz.

Abbildung 4 zeigt exemplarisch den Einsatz von vier gekoppelten kleinen Analogrechnern des Typs *THE ANALOG THING*[12] in einer Übung im Bereich der medizinischen Systembiologie. Dieses Beispiel demonstriert überdies deutlich die Eigenschaft, dass die Anzahl der Rechenelemente dem zu lösenden Problem angemessen sein muss. Die Darstellung von Variablen als Spannungen gestaltet das Zusammenschalten mehrerer Analogrechner, wie man sieht, ausgesprochen einfach – neben einer Masseverbindung sind lediglich wenige Verbindungen für die auszutauschenden Variablen erforderlich.

Erste Erfahrungen in der Programmierung und Anwendung moderner Analogrechner, deren Rechenelemente mit Hilfe von Kreuzschienenverteilerstrukturen anstelle des klassischen Patchfeldes miteinander verbunden werden, lassen sich bereits mit dem in Abbildung 5 gezeigten System sammeln.[13]

4 Ausblick

Das Analogrechnen erlebt gegenwärtig eine ausgesprochene Renaissance, wobei gerade die Verwendung moderner CMOS-Technologien einen Durchbruch darstellen und es perspektivisch ermöglichen, eine Vielzahl von Rechenelementen als hoch energieeffizienten Co-Prozessor eng mit einem Digitalrechner zu koppeln. Dies wird

[11] Siehe z. B. [Ulmann et al. 2024].
[12] Siehe `https://the-analog-thing.org`.
[13] Siehe `https://anabrid.com/lucidac`.

Abbildung 5: Drei rekonfigurierbare Analogrechner

neben den oben genannten Anwendungsgebieten sicherlich zu vollkommen neuartigen Applikationen führen, die grundlegend neue Märkte erschließen werden.

Zusammen mit speicherprogrammierten Digitalrechnern sowie dem gerade in aller Munde geführten Quantencomputing stellt das Analogrechnen die dritte Säule des maschinellen Rechnens dar und wird zu einem festen Bestandteil moderner (Hochleistungs-)Rechnersysteme werden.

Literatur

[Anadigm(2006)] Anadigm, *AN13x/AN23x series, AnadigmApex dpASP Family User Manual*

[Cowan(2005)] Glenn Edward Russell Cowan, *A VLSI Analog Computer / Math Co-processor for a Digital Computer*, Columbia University, 2005

[Lucas(2014)] Andrew Lucas, “Ising formulations of many NP problems”, `https://arxiv.org/abs/1302.5843`, abgerufen am 05.06.2024

[Guo et al.(2016)] N. Guo, Y. Huang, T. Mai, S. Patil, C. Cao, M. Seok, S. Sethumadhavan, and Y. Tsividis, “Energy-Efficient Hybrid Analog/Digital Approximate Computation in Continuous Time”, in *IEEE Journal of Solid-State Circuits*, vol. 51, no. 7, pp. 1514-1524, July 2016

[Ulmann 2023] Bernd Ulmann, *Analog Computing*, DeGruyter, 2nd edition, 2023

[Ulmann 2023] Bernd Ulmann, *Analog and Hybrid Computer Programming*, DeGruyter, 2nd edition, 2023

[Ulmann et al. 2024] Bernd Ulmann, Shrish Roy, „Building a simple oscillator based Ising machine for research and education“, in *IEEE Xplore*, `https://ieeexplore.ieee.org/document/10739029`

EUKLIDS ELEMENTE INTERAKTIV

Jörg Hudelmaier

joerghudelmaier@web.de

We present an interactive version of Euclid's *Elements* based on the Dyalog APL HTMLRenderer object. Dots, lines, and circles are constructed as SVG elements on a web control using HTML-Renderer's **ExecuteJavaScript** function.

Euklids *Elemente* sind zweifellos das einflussreichste jemals geschriebene Lehrbuch der Mathematik. Beispielhaft hierfür ein Zeugnis über die Konversion des Philosophen Thomas Hobbes (1588–1679):

He was forty years old before he looked on geometry; which happened accidentally. Being in a gentleman's library Euclid's Elements lay open, and 'twas the forty-seventh proposition in the first book. [NB: der Satz des Pythagoras] *He read the proposition. "By God," said he, "this is impossible!" So he reads the demonstration of it, which referred him back to such a proof; which referred him back to another, which he also read. ... at last he was demonstratively convinced of that truth. This made him in love with geometry.* (John Aubrey: *A Brief Life Of Thomas Hobbes*)

Die berühmteste Ausgabe der *Elemente* ist jene von Oliver Byrne aus dem Jahr 1847 – „das vielleicht schönste und zugleich skurrilste Mathebuch aller Zeiten" (FAZ), wieder aufgelegt im Jahr 2013, mittlerweile auch in einer Online-Ausgabe (https://www.c82.net/euclid) sowie als TeX-Code resp. Metapost-Code (https://github.com/jemmybutton/byrne-euclid – einschliesslich der Initialen!) verfügbar. Diese Ausgabe umfasst die ersten sechs Bücher über die Ebene Geometrie. Ihr ausführlicher Titel lautet: *THE FIRST SIX BOOKS OF THE ELEMENTS OF EUCLID in which coloured diagrams and symbols are used instead of letters for the greater ease of learners.*

Bekanntlich handelt die Euklidische Geometrie von Geraden und Kreisen und deren Schnittpunkten. Wir wollen diese im folgenden in der komplexen Zahlenebene abbilden. Geraden definieren wir dabei natürlich wie gewöhnlich durch zwei Punkte, Kreise dagegen definieren wir entweder durch Mittelpunkt und Radius oder aber durch drei nicht kollineare Punkte. Überdies benötigen wir die bekannten Formeln für die Schnittpunkte von Geraden und Geraden, Geraden und Kreisen sowie Kreisen und Kreisen.

In der Byrne'schen Ausgabe lautet die Proposition I des ersten Buches wie folgt: „On a given finite straight line to describe an equilateral triangle." Der Beweis besteht i.W. aus einer Graphik und einigen Anweisungen, diese zu lesen und kommt – wie alle Beweise in dieser Ausgabe – gemäß dem vollständigen Titel ohne die üblichen Buchstaben-Bezeichnungen aus. Dies wird uns die Abbildung beträchtlich erleichtern.

Mit dem HTMLRenderer-Objekt, das mit Dyalog-APL/Version 16 ausgeliefert wurde, haben wir jetzt ein Werkzeug, solche anspruchsvolle interaktive Graphik mit CSS und SVG in APL umzusetzen.

Die Grobstruktur eines Programms zum Beweis der Proposition 1 mag dann etwa so aussehen:

```
Run;A;B;L1;O1;O2;C;L2;L3;msg
createSVGCanvas

msg←'Bitte zweimal tippen, um eine Strecke zu zeichnen'
msgToTheUser msg
A←getPoint
'color:yellow'drawDot A
B←getPoint
'color:red'drawDot B
L1←'color:blue'drawLine A B

msg←'Bitte tippen, um Kreise um die beiden Endpunkte zu zeichnen'
msgToTheUser msg
getPoint
O1←'color:yellow'drawCircle A B
O2←'color:red'drawCircle B A

msg←'Bitte tippen, um einen Schnittpunkt der beiden Kreise zu bestimmen'
msgToTheUser msg
getPoint
C←O1 intersection O2
'color:blue'drawDot C

msg←'Bitte tippen, um den roten und den gelben Punkt '
msg,←'mit dem blauen Punkt zu verbinden'
msgToTheUser msg
getPoint
L2←'color:red'drawLine A B
L2 rotateTo A C
L3←'color:yellow'drawLine B A
L3 rotateTo B C

msgToTheUser'QED!'
```

In den Funktionen **createSVGCanvas** und **createHTMLSkeleton** wird eine HTML-Datei erstellt und die Kommunikation zwischen Javascript und APL etabliert:

```
createSVGCanvas

 :If 0=⎕NC'HR'
    'HR'⎕WC'HTMLRenderer'
    HR.Coord←'Pixel'
    HR.Posn←10 990
    HR.Size←1510 2000
    HR.HTML←createHTMLSkeleton
*   HR.onWebSocketReceive←'getClick'
*   HR.InterceptedURLs←1 2⍴'ws://dyalog_root/' 1
    ⎕DL 1
:EndIf
N←0
```

```
html←createHTMLSkeleton;style;fn;meta

html←''
html,←⊂'<!DOCTYPE html>'
html,←⊂'<html>'
html,←⊂'<head>'
meta←'name="viewport" '
meta,←'content="width=device-width, initial-scale=1.0, user-scalable=no">'
html,←⊂'  <meta ',meta
html,←⊂'  <meta charset="utf-8">'
html,←⊂'</head>'
html,←⊂'<body>'
html,←⊂'  <svg id="canvas"'
html,←⊂'       xmlns="http://www.w3.org/2000/svg"'
html,←⊂'       xmlns:xlink="http://www.w3.org/1999/xlink";'
html,←⊂'       height="700px" width="970px" viewBox="0 0 970 700"'
html,←⊂'       version="1.2"'
style←'"position:absolute; top:10px; background-color:lightgrey; '
style,←'transform:scaleY(-1)"'
html,←⊂'       style=',style
html,←⊂'  >'
html,←⊂'  </svg>'
html,←⊂'  <div id="instruction"'
style←'"position:absolute; top:520px; left:100px; '
style,←'font-size:3.2rem; color:white; max-width:800px"'
html,←⊂'       style=',style
html,←⊂'  >'
html,←⊂'  </div>'
html,←⊂'  <script>'
*html,←⊂'    ws = new WebSocket("ws://dyalog_root/");'
fn←'function(e)ws.send([e.offsetX,e.offsetY])'
*html,←⊂'    document.getElementById("canvas").onclick=',fn
html,←⊂'  </script>'
html,←⊂'</body>'
html,←⊂'</html>'
html←∊html,¨⎕TC[2]
```

Für die Kommunikation von Javascript nach APL sorgen die hier mit * gekennzeichneten Befehle: Zunächst wird ein Websocket zum URL dyalog_root eingerichtet, dann wird eine Callback-Funktion definiert, die beim Klicken auf die Weboberfläche die Koordinaten über den Websocket an diesen URL schickt. Damit APL über den Klick informiert wird, muss der Websocket-URL hierfür im HTMLRenderer als Intercepted URL registriert werden. Damit löst ein Aufruf des URLs im Javascript dann ein Event in APL aus. Diesem Event wird sodann im APL die Callback-Funktion **getClick** zugeordnet.

In der Funktion **msgToTheUser** wird die Kommunikation in umgekehrter Richtung verwendet:

```
msgToTheUser text;js

js←'document.getElementById("instruction").innerHTML="',text,'"'
HR.ExecuteJavaScript js
```

Hier wird lediglich der jeweilige Text per Javascript-Aufruf aus APL ins DOM eingefügt. Interessant sind noch die drei Funktionen

```
{P}←getPoint;z;P

z←waitForClick
P←⎕NS''
P.(T Z N)'D'z N
N+←1
```

und

```
z←waitForClick;x;y;z

'#.F'⎕WC'form'('size'1 1)('posn'0 0)
⎕DQ'#.F'
x y←X Y
z←code x y
```

und

```
getClick msg;js

js←'document.getElementById("instruction").innerHTML = "";'
HR.ExecuteJavaScript js
X Y←⍎3⊃msg
⎕EX'#.F'
```

Der Ablauf ist folgender: Beim Aufruf von **getPoint** wird via **waitForClick** eine unsichtbare APL-Form angelegt. Diese wartet mit **⎕DQ** auf Ereignisse, welche sie betreffen. Damit bleibt das Programm zunächst stehen. Wenn der Benutzer dann auf die Oberfläche klickt, löst dies zunächst ein Javascript-Clickevent aus. Dieses löst via Websocket wiederum ein APL-Event aus, wodurch die Callback-Funktion **getClick** aufgerufen wird. Diese löscht die unsichtbare APL-Form und schreibt die Klick-Koordinaten in zwei globale Variabeln. Durch das Löschen der Form endet zugleich **⎕DQ**. Damit läuft nun die Funktion **waitForClick** weiter und setzt die Klick-Koordinaten **X** und **Y** zu einer komplexen Zahl $z = \mathtt{X} + i\,\mathtt{Y}$ zusammen. Diese komplexe Zahl gibt sie an die Funktion **getPoint** zurück, und hier wird dann ein anonymer Namespace angelegt als Repräsentant des geklickten Punktes im APL.

Auf diesen Namespace kann dann die aufrufende Funktion zugreifen, um z.B. an der betreffenden Stelle einen Punkt zu malen. Hier kommt dann etwa die Funktion **drawDot** zur Anwendung:

```
{style}drawDot obj;js;color;Z;N;Y;X;C;cs;estyles;estyle

color←'black'
:If 0<⎕NC'style'
  evalStyle style
:EndIf
Z N←obj.(Z N)
X Y←decode Z
C←color
js←''
js,←'p = document.createElementNS("http://www.w3.org/2000/svg","circle");'
js,←'p.setAttribute("cx","',(⍕X),'");'
js,←'p.setAttribute("cy","',(⍕Y),'");'
js,←'p.setAttribute("r","8");'
js,←'p.setAttribute("fill","',C,'");'
js,←'p.setAttribute("id","',(⍕N),'");'
js,←'document.getElementById("canvas").appendChild(p);'
HR.ExecuteJavaScript js
obj.C←C
```

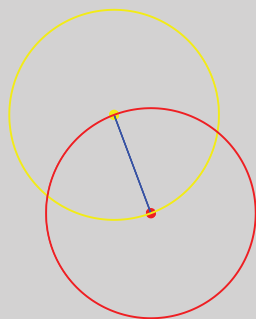
Die übrigen **draw**-Funktionen werden analog definiert, indem man per JavaScript die entsprechenden SVG-Elemente anlegt. Dies kann ggf. auch animiert werden, indem man z.B. statt des gesamten Kreises successive mit passender Verzögerung eine Anzahl Kreissegmente zeichnen lässt.

Es sei allerdings noch ausdrücklich darauf hingewiesen, dass das beschriebene Programm sich tatsächlich nur zur Illustration der allerersten Proposition eignet. Alle folgenden Propositionen benützen ja bereits zuvor bewiesene Lemmata. Hierfür wird dann ein Mechanismus benötigt, mit dem ohne weitere Benutzereingaben auf vorhandene Illustrationen zugegriffen werden kann. Das ist aber eine Erweiterung, die vollständig im APL ablaufen kann und keine Erweiterung der Interaktion zwischen Javascript und APL erfordert. Deshalb lassen wir es hiermit bewenden und merken lediglich noch an, was Sie, lieber Leser, bereits bemerkt haben werden, daß nämlich die Existenz der Schnittpunkte unserer beiden Kreise stillschweigend vorausgesetzt wurde, aber letztlich nicht bewiesen. Das ist in der Tat eine bekannte Lücke, die sich auch unter den Euklidischen Annahmen nicht schliessen lässt.

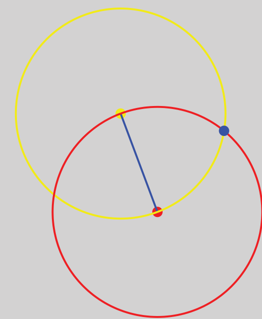
Abschliessend seien hier noch einige mittels der Funktion **PrintToPDF** erstellte Abbildungen zu den einzelnen Konstruktionsschritten angefügt:

Bitte zweimal tippen, um eine Strecke zu zeichnen

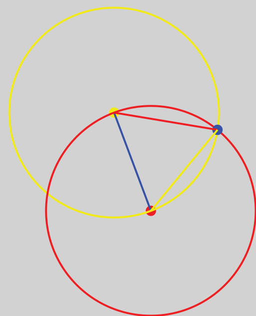
Bitte tippen, um Kreise um die beiden Endpunkte zu zeichnen



Bitte tippen, um einen Schnittpunkt der beiden Kreise zu bestimmen



Bitte tippen, um den roten und den gelben Punkt mit dem blauen Punkt zu verbinden



QED!

Migration from APL+Win to Dyalog

Dr. Markos Mitsos — ERGO Group AG

Markos.Mitsos@ergo.de

Abstract

The task at hand is the migration of a sheaf of APL+Win workspaces to Dyalog. For various reasons, among others lack or time, this turns out to be a long-term project. Underlying the migration is a workspace management framework. It uses **Link** for coding, TortoiseSVN for versioning and Array Notation as well DB2 for debugging/testing.

"Migration" is understood to encompass not only copying code and making absolutely necessary changes, but creating a "true" Dyalog workspace. This means for example

- a workspace superstructure fitting into the management framework and an internal namespace structure
- a schematic object structure using multiline headers, Migration Level 1, clean-up end and no semi-globals
- use of new methods like local namespaces, dfns, trains, new primitives and different/new system functions
- some utilisation of Classes, replacement of infrastructure for GUI and COM

In other words the targets are better workspaces than those to begin with!

The listed changes could in principle form part of a post migration project. However they would not have been possible without interruption of operation because of very low developer capacities.

In fact the migration is used, overloading its meaning, as an opportunity to make further changes, which would similarly have been very difficult with the additional difficulty of classic workspaces and no object versioning. Those may be described in another article. Some could in principle have been done in APL+Win (like better error reaction/handling), some not (like automated tests based on **Link** and Array Notation).

Contents

1 Targets and requirements . . . **19**

1.1 Target . . . 19

1.2 Necessary framework . . . 20

1.3 Meaning of "migration" . . . 21

1.4 Further changes . . . 22

2 Structure of workspaces and objects **22**
2.1 Workspace structure . . . 22
2.1.1 Workspace interaction . . . 22
2.1.2 Workspace code . . . 22
2.2 Object structure . . . 24
2.2.1 Header and functionality . . . 24
2.2.2 Passing arguments . . . 25
2.2.3 Schematic internal structure . . . 27

3 Dyalog methods and Classes **27**
3.1 New/Different Dyalog methods and capabilities . . . 27
3.1.1 Namespaces . . . 27
3.1.2 Defined Objects and Tacit Code . . . 28
3.1.3 Primitives . . . 29
3.1.4 System functions . . . 31
3.2 Object Oriented Programming with APL syntax . . . 34
3.2.1 Classes . . . 34
3.2.2 Schematic GUIs . . . 36
3.2.3 COM-interfaces . . . 37

List of Tables

1 Object name examples in old versus new workspace . . . 24

List of Figures

1 Usual work setup . . . 20
2 Exchange of data between Dyalog and APL+Win through cf . . . 21
3 Schematic workspace superstructure . . . 23
4 Exchange of code between workspaces during deployment . . . 23
5 Schematic example of a new workspace structure . . . 24
6 Example of schematic main GUI . . . 36
7 Example of schematic parameter GUI . . . 37

1 Targets and requirements

1.1 Target

In DKV/ERGO APL has been used for many years for applications referencing the balance sheet and premium recalculation. Other programs implement controls on databases or relay technical specifications to the DB2 administrators. Still other applications provide the means to define premium capping schemes and are used for making decisions.

A highly interconnected sheaf of APL+Win workspaces, that may be described in another article, has developed over the years. This must be migrated to Dyalog. An obvious requirement is that the operation of all applications must be warranted at all times.

For various reasons the migration turns out to be a long-term project. Among other reasons, there is a lack of developers. The problem is compounded by the fact, that the few developers have many other assignments and can only use a small part of their time for the migration.

A resulting constraint is, that old APL+Win workspaces must be able to use at least data from new, migrated Dyalog ones. This also means that the migration progresses from low level utilities to intermediate functionalities, leaving complex simulations as the last to be migrated.

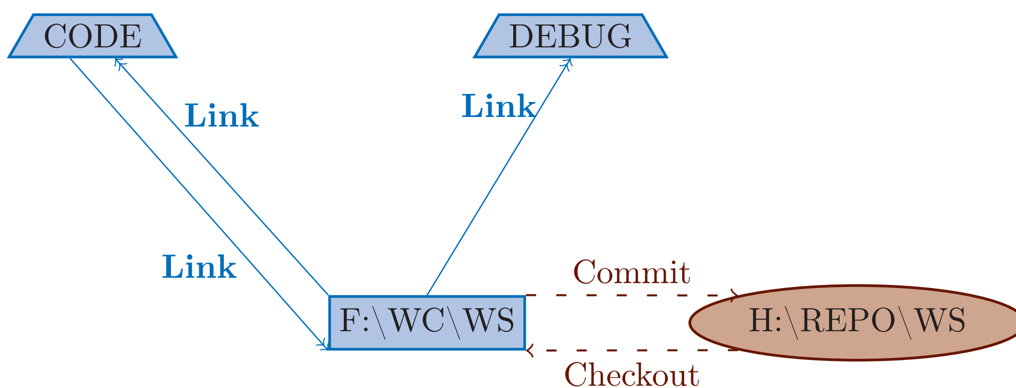
Another consequence is, that the migration is seen as a good opportunity to also enhance, improve or correct: if the code has to be worked on, then one may directly do it the right way. Put another way the resulting workspaces should be better than the sources!

1.2 Necessary framework

Underlying the migration is a workspace management framework. This has been described in the APL-Journal of APL-Germany under the title “Management of Dyalog APL workspaces” (see https://apl-germany.de/wp-content/uploads/2023/03/APL_Journal_2022.pdf). A brief summary follows here.

The framework uses **Link** for managing code in Unicode text files. This means that the migration is accompanied by a fundamental change in the way development, debugging and deployment is done. For example `)SAVE` is not used any more. The framework provides an convenient way to build temporary workspaces for various uses. Figure 1 for example outlines the usual work setup, two workspaces in tandem.

Figure 1: Usual work setup



The code is versioned using TortoieSVN. This is just a choice. Any versioning system or non at all could be used. However the technical implementation of the framework in APL at the moment uses only SVN commands on the DOS level.

Debugging or more generally testing for stability under changes in the code itself, other workspaces, the APL-installation or the IT-environment is build on Array Notation as well as DB2. The process implemented in the framework documents not only the target results of the used Unit tests, but also the outcome of comparisons to actual results.

An important process is deployment of the application as workspace. The framework first executes all provided Unit tests and then saves the workspace in the background using ⎕SAVE. In this way the code management system fulfils requirements on deployment without creating a huge overhead.

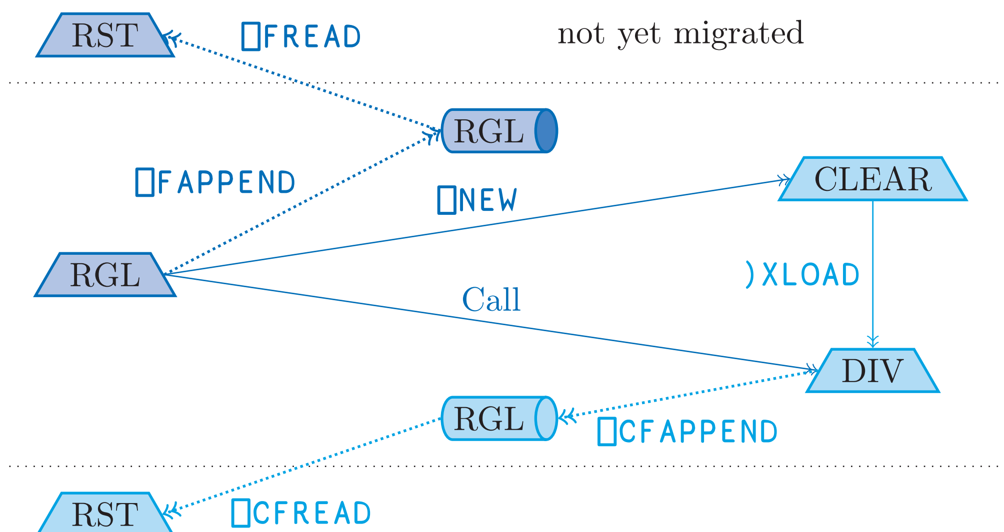
1.3 Meaning of “migration”

Because of the aforementioned constraints it was decided to interpret “migration” in a broad sense. It is understood to encompass more than copying code and making changes absolutely necessary for functionality. This narrow, but usual, use of “migration” would mean keeping workspace structures, down to object names, as they are in APL+Win. It would also probably mean using Migration Level 3 to minimise changes.

Instead it is intended that the resulting workspaces are “true” Dyalog workspaces. Those should be namespace based and not only use the Dyalog standard of Migration Level 1, but also features that differentiate Dyalog from APL+Win.

Additionally the long-term scope of the migration means that some interfaces between workspaces must be streamlined in a way that APL+Win component files may be used as bridges between migrated and not-yet-migrated ones, as shown in Figure 2. Only this way workspaces can be migrated one at a time.

Figure 2: Exchange of data between Dyalog and APL+Win through cf



Part of the migration described in this article is the structure connecting workspaces as well as those of workspaces and objects. Another part consists in the use of new/different methods offered by Dyalog, like namespaces, new

primitives or different system functions. These decisions are aimed at the target of improved resulting workspaces.

1.4 Further changes

In fact the migration is used, overloading its meaning, as an opportunity to make further changes, which are not described in this article. Those modifications would have been extremely difficult, even practically impossible, within the above constraints on operation, under APL+Win with the additional burden of classic workspaces and no object versioning. Some could in principle have been done in APL+Win and some not.

An example of the first sort is better error reaction/handling, of the second automated tests based on **Link** and Array Notation. Many other changes boil down to enhanced modularisation, also known as problems accumulated "historically" over the years...

2 Structure of workspaces and objects

2.1 Workspace structure

2.1.1 Workspace interaction

At the highest level all workspaces must conform to the superstructure required by the corresponding management framework. The requirement has multiple purposes.

Central among them is the functionality of the framework itself. It requires certain root level namespaces, namely a main, a test and a description/control one, in order to build an deploy workspaces, as shown in Figure 3. The burden on migration is negligible. Each workspace must be associated with a main name, for example **div** for DIVERSES.

In fact the old APL+Win flat workspaces already used conventions about prefixes of object names to facilitate name distinction in the case of workspaces exchanging code. This leads to the second reason, easy and comprehensive separation and cooperation of workspaces, as demonstrated in Figure 4.

A third target is to reserve distinct root level namespaces for global or temporary objects. For example the framework imports itself into #**.temp**, builds the required workspaces and then deletes itself.

2.1.2 Workspace code

At the level of workspaces the code proper of each one should be compartmentalised into namespaces lying directly under the main one. This is not strictly necessary for building workspaces, but leads to an easily readable overall structure.

For the number of objects in each relevant workspace, one level of namespaces provides sufficient lexicographical order and does not create too large collections.

Figure 3: Schematic workspace superstructure

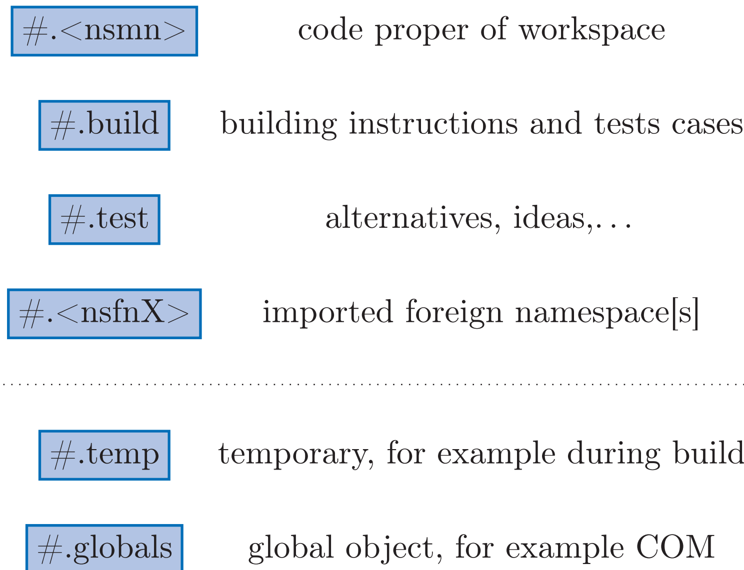
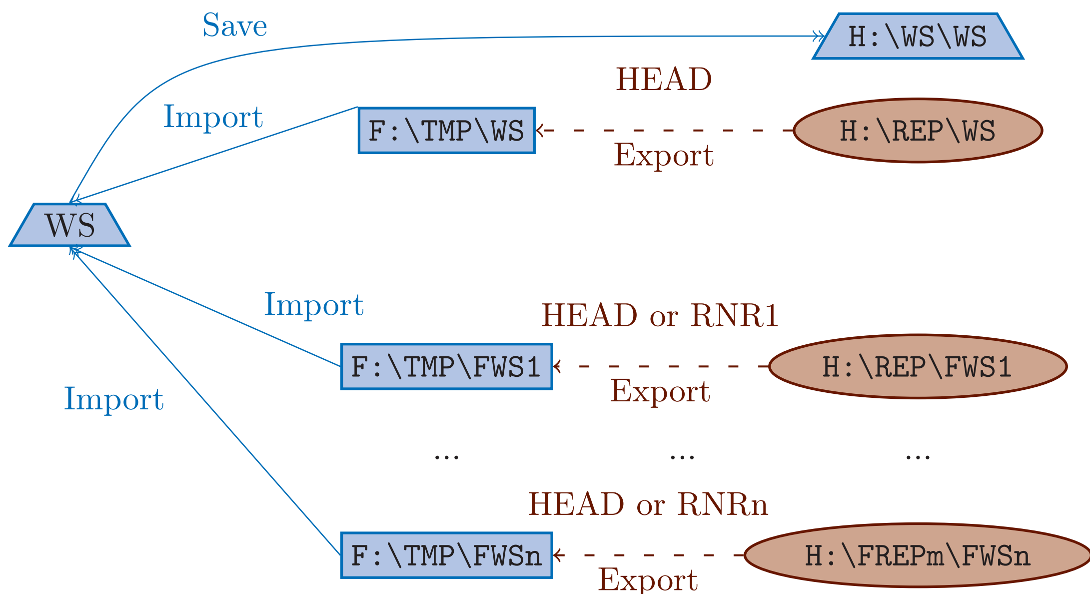


Figure 4: Exchange of code between workspaces during deployment



For that to be true good modularisation and cooperation among workspaces is needed.

To use automated tests however, the namespace structure is more than convenient. The framework handles test cases at the level of triples of workspace, namespace and object and expects one function describing them for each namespace of the current workspace. Of course it is possible to trick the framework into testing different structures — but cumbersome.

For the migration this means some conceptual work. Usually the objects in a workspace will form collections with similar purpose and/or function that can be grouped into an namespace. In fact thinking about the grouping may clarify

details about the workspace function! Figure 5 outlines the (coarse) structure of a Dyalog workspace, whereas Table 1 compares the old names of some of the contained objects with the new ones, hinting also at some modularisation improvements.

Figure 5: Schematic example of a new workspace structure

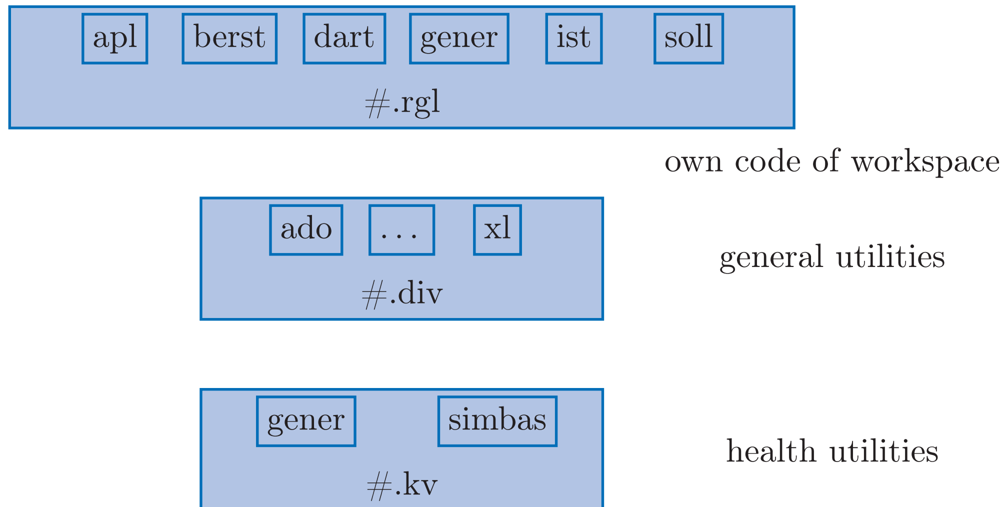


Table 1: Object name examples in old versus new workspace

APL+Win	Dyalog
actuarial data	
RGL_BEREITSTELLEN	#.rgl.gener.BEREITSTELLEN #.rgl.berst.BEREITSTELLEN
RGL_KALK_BEREITSTELLEN	#.rgl.soll.BEREITSTELLEN
RGL_KALK_DART_BEREITSTELLEN	#.rgl.dart.BEREITSTELLEN
RGL_KALK_DART_APL_BEREITSTELLEN	#.rgl.apl.BEREITSTELLEN
RGL_TM_BEREITSTELLEN	#.rgl.ist.BEREITSTELLEN
RGL_TM_BER_ST_KVTB*	#.rgl.ist.BER_ST_KVTB*
RGL_TM_HGB_BEREITSTELLEN	#.rgl.ist.HGB_BEREITSTELLEN
RGL_TM_HGB_BER_ST_KVTB*	#.rgl.ist.HGB_BER_ST_KVTB*
general utilities	
ADO_CONNECT	#.div.ado.CONNECT
ADO_SELECT	#.div.ado.SELECT
health utilities	
KV_GRD_BEREITSTELLEN	#.kv.simbas.GRD_BEREITSTELLEN #.kv.simbas.TAR_EINSCHRAENKEN

On the negative side all object calls must be updated. The only method to avoid this would be to transfer the whole, old, flat workspace into the new main namespace. Even leaving aside the aforementioned workspace specific object name prefixes, this would not lead to a "real" Dyalog workspace.

2.2 Object structure

2.2.1 Header and functionality

Over the years the functional objects in APL+Win acquired a gradually more schematic structure. This schema was overhauled, formalised and documented as part of the migration. It provides for localisations pertaining to error handling and system variables in the head line.

The Dyalog header of (traditional) functions and operators itself is multi-line, starting with one line each for right arguments and left parameters, regardless of their existence, and ends with a line of necessary temporary objects, like *V* or *ns*. Documented conventions govern the use of such objects. In between the header follows the object structure with a line for each section that needs local objects.

The differences to APL+Win are hinted at in an old header with more than 300 characters and the associated distribution of parameters

```
ERG ← PRMS RGL_TM_BER_ST_KVTB0171 ARG;⎕elx
      ;TARIFE_TAID_DEF;TARIFE_TAID ;HINW_UEB_DEF;HINW_UEB
...
...                                              ... ;V;VV;I
...
(TARIFE_TAID_DEF TARIFE_TAID   HINW_UEB_DEF HINW_UEB
       HINW_UEB_NR_MAX   HINW_VOR_DEF HINW_VOR) ← 7 ↑ ARG
...
(BILANZ DATUM_VON DATUM_BIS STELL_KAV   CONNR   MSGBN
    PROT   WBNR XLN_PRMS   RESP_MIN RESP_MAX) ← 11 ↑ PRMS
...
```

compared to the new one

```
(TARIFE_TAID rgl hinw) ← {PRMS} BER_ST_KVTB0171 ARG
                           ;⎕ML;ERR_MSG;ERR_MSG_BEG;⎕TRAP
 ;TARIFE_TAID_DEF;TARIFE_TAID;rgl;hinw      ⍝ Argument
 ;DATUM_VON;DATUM_BIS;regst;db;msgb;sondst  ⍝ Parameter
...
 ;V;VV;I;II;ns                              ⍝ temporär
...
 (TARIFE_TAID_DEF TARIFE_TAID   V V) ← ARG
                                        ← 4 ↑ ARG , ...
 'rgl' 'hinw' ⎕NS¨ ARG[3 4]
...
```

```
 (DATUM_VON DATUM_BIS V   db   msgb V) ← PRMS
                                              ← 6 ↑ PRMS , ...
 'regst' 'sondst' ⎕NS¨ PRMS[3 6]
...
```

The multi-line header represents the biggest structural change at object level during migration. A similar structure in one line had been striven for in APL+Win. Sometimes however it was almost impossible to handle headers with literally thousands of characters.

The header is complete in the sense that no semi-globals are used. In case global sources or targets are needed they are addressed via full namespace paths. In many cases these non-functional interactions are documented by separate result variables. In most cases dedicated functions named *GLOBAL_NAMES* specify the necessary paths as for example in

```
    #.div.gener.GLOBAL_NAMES 'globals'
#.globals
    1 2 3 #.div.gener.GLOBAL_NAMES 'com' 'xl' 'rng'
#.globals.com.xl.wbc.wb_01.sc.s_02.rng_03
    1 #.kv.gener.GLOBAL_NAMES 'simbas' 'rgl' 'hgb'
#.globals.simbas.rgl.hgb_01
    #.rgl.gener.GLOBAL_NAMES 'path' 'ist'
H:\Aktuariat\Simulationsbasis\Rechnungsgrundlagen
                             fachlich\aus IST eingelesen\
```

The forfeiture of semi-globals means that many Dyalog objects are strictly functional. In many others the only deviation from locality is interaction with external data sources.

2.2.2 Passing arguments

Another significant change concerning the header is the use of namespaces. For one thing references that pass main results reduce in some cases the size of the header dramatically. Namespaces local to the function further reduce the amount of main part names needing localisation.

Finally the aggregation of parameters with similar function in so called "small namespaces" further reduces the size of the header and makes the object syntax clearer. However this changes mean also work on the conceptual level. It is not always self-evident, why, say, "regular steering" **regst** would be a meaningful aggregation in a specific workspace and what exactly it should encompass. For an example compare the above multi-line header with its old counterpart. (In this example the same object model is used for a range of similar functions, whereas the old workspace contained many variants.)

The convention on namespaces passed to an object is that a reference should be kept when the contents are only to be used. If they are to be modified in contrast, a copy is created. The aforementioned example uses the latter.

2.2.3 Schematic internal structure

At the start of each function or operator some preparatory steps are undertaken, like setting the Migration Level to 1. Immediately after the header first the right arguments and then the left parameters (and, where applicable, the operands) are handled. This means that passed arguments are filled to the length specified by the object syntax and then distributed to corresponding variables or namespaces. In almost no case the number of arguments is enforced by the data model.

Parameters are immediately controlled. The checks depend on the underlying functionality, and may be purely technical (rank, depth, data type, ...) or veer to the business logic side. Failed checks produce easily readable messages. The same goes for problems encountered in the main part.

The end part is always reserved for clean-up and marked with the label `ENDE:`. During clean-up all errors are ignored. Conceptually this part could be defined by some appropriate keyword. APL+Win had introduced `:FINALLY` and `:TRYALL` but those where not used in the old code.

All those conventions are not new. They are just strictly enforced during migration in cases where they were not obeyed under APL+Win. Checks on parameters in particular are enriched, among other things to allow more precise automated testing.

3 Dyalog methods and Classes

3.1 New/Different Dyalog methods and capabilities

3.1.1 Namespaces

Namespaces, as a defining difference between APL+Win and Dyalog, have already been discussed in the previous section. There they defined the high level structure and cooperation between workspaces as well as that of the latter. They also made the syntax of many objects simpler and more readable by collecting similar parameters in "small namespaces" and passing a reference.

As mentioned there in passing, local namespaces are also used in functional objects. Often a multitude of similar variables is needed in a function or operator. For example a complex manipulation of actuarial data may collect the changes themselves as well as statistical data on them.

Under APL+Win separate variables would be allocated, with their names alluding to their affiliation — usually with suitable prefixes. The obvious improvement under Dyalog is to define local namespaces as containers for the variables of one kind. This is done during migration, and not only reduces localisation and header size significantly, but also enhances readability and highlights connections between variables.

In the following schematic example, a part of the comparison of actuarial data, in which many cases must be distinguished and statistics collected, 4 namespaces replaced almost 50 variables

```
'stat' 'hinw_ign' 'hinw_ber' 'vorg' ⎕NS¨ ⊂''
V ← (0 , ≢rgl_def.DEF) ⍴ 0

stat.BEARBL ← 'DELETE' 'UPDATE' 'INSERT'
stat.HINW_DEF  ← ''
...
hinw_ign.(ALT_NUR_SOLL ALT_NUR_IST) ← ⊂V
...
hinw_ber.(F_DSP_SOLL F_DSP_IST) ← ⊂V
...
vorg.(RGL_D RGL_U RGL_I) ← ⊂V
```

3.1.2 Defined Objects and Tacit Code

An important new coding method offered by Dyalog are dfns and dops. Although most of the functions and operators that have to be migrated are strictly functional, very few of them could be easily transformed using the Defined paradigm. The main reason, beside error handling and reaction, is complexity. Almost all of them heavily use control structures.

On the other hand many simple algorithms, like "trim leading and/or trailing blanks" were considered too small for a traditional function in APL+Win and were implemented inline through snippets like

```
(- + / ∧ \ ' ' = ⌽ V) ↓ (V ← VAR)
```

as part of many functions. During migration to Dyalog some of these base utilities were implemented as dfns, for example "trim right" with additional options as

```
TRIM_R ← {α ← ' '
...
1 ≥ |≡ω : (- + /  ∧ \  (⌽  ω) ∊   α) ↓  ω
1 ≥ |≡α : (- + /¨ ∧ \¨ (⌽¨ ω) ∊¨ ⊂α) ↓¨ ω
          (- + /¨ ∧ \¨ (⌽¨ ω) ∊¨  α) ↓¨ ω
         }
```

At least for the time being, this is the envisioned use of Defined Objects: small, strictly functional, internal algorithms.

Similarly, small algorithms, that are not so generic in nature, still appear in many functions. Here dfns are (also) very useful and are used. In same cases the implementation is "inline", meaning the use of a nameless dfn as part of the code. In others a temporary function is defined. As a convention the generic local name `fn` is used in those cases. In the following example a nested dfn is created on the fly for grouped scattering

```
...
CODEL ,← '{((+ / ω * 2) ÷ (≢ω)) * ÷2}'
 '{((+  /  (ω  -  (((+/)÷≢) ω))  *  2)  ÷  (≢ω))  *  ÷2}'
```

```
...
CR ← ('↓'  '{((×⍺) × (≢⍵) ⌊ |⍺) ↑ ⍵}') [VV ← ...
CA ← (,¨ (⊂'G< AUSG[99999] >') ⎕FMT¨ V)  ,¨  CR
CS ← (''  ' {⍵[⍋⍵]}'  ' {⍵[⍒⍵]}') [1 + + ≠ ...
...
CODE ←  ...  (V ⊃¨ ⊂CODEL)  ,¨  CA  ,¨  CS  ,¨ ...
FN ← ⍎  '{'  ,  ((≢TR) ↓ CODE)  ,  '}'
```

whereas in

```
...
VVV ← '{⍺ ← 0
 ⋄ ''?'' ≡ ⍵                                       : 1
 ⋄ (0 ≥ ⍺)  ∧ (I ← ~ (⎕DR ⍵) ∊ 80 160 320 326) : 0
 ⋄ 0 ≥ ⍺                                           : 1
 ⋄ I                          : (⊂⍴⍵) ∊ ,¨ (0⍴0) 1 ⍺
 ⋄ (V ← ⎕VFI ⍵) [1] ∊ (,1) (⍺⍴1)                   : 1
 ⋄ 0}'
...
CHECKT ⍪← (,¨ (⊂ 'G<(9999⊃CnD.DATA\_TYPE)>, < ' , VVV
       , ' >, G<(9999⊃CnD.VAL)>')  ⎕FMT¨ ,[0.5]¨ 2 /¨ ⍸I)
 , V ,[1.5]  ((⊂'Nur alphanumerische Einträge sind beim
                          Control "') ,¨ VV ,¨ (⊂'" erlaubt'))
```

code is prepared for later, inline use during checks on GUI-input.

Getting used to trains (of thought) is not easy for older (measured in years of living as well as coding!) developers... Still some first examples, like the preparation of grouped averages

```
...
OPER   ,← '⍉'                 '⌽'        '⊖'
CODER  ,← '((+ /) ÷ ≢)'       '((× /) * (÷ ≢))'
                                         '((+ / |) ÷ ≢)'
CODES  ,← '((+ \) ÷ (⍳ ≢))'  '((× \) * (÷ (⍳ ≢)))'
                                         '((+ \ |) ÷ (⍳ ≢))'
...
```

have appeared in migrated code.

3.1.3 Primitives

Some of the (newer) primitives not available in APL+Win were used during migration, others not. The main reason for the distinction was that for the first kind of primitives there were obvious use cases.

More specifically, some of the possible uses of Squad Equal ⌸ as Key had been implemented under APL+Win as complicated algorithms in base utilities. In those cases it was only natural to take advantage of new opportunities during migration. The kernel of the old objects was replaced by Key, leaving the

syntax and use of the cover function (more or less) unaltered. For example the algorithm dealing with representatives of equivalence classes and their numbers

```
VV ← DATEN[⍋DATEN;]
I ← 1 , ∨ / (1 ↓[1] VV) ≠ (¯1 ↓[1] VV)
V ← I / ⍳ ↑ ⍴VV
L ← (1 ↓ V , (1 + 1 ↑ ⍴VV)) - V
(I ⌿ VV) , L
```

by `({⍺ , (≢⍵)} ⌸) DATEN`), and the one for grouped operations, here sums,

```
I ← 1 , ∨ / (1 ↓[1] DATEN) ≠ (¯1 ↓[1] DATEN)
V ← I / ⍳ ↑ ⍴DATEN
L ← (1 ↓ V , (1 + 1 ↑ ⍴DATEN)) - V
MAX ← ⌈ / L
II ← , L ∘.≥ ⍳MAX
VV ← ((⍴V) , MAX , (1 ↓ ⍴WERTE)) ⍴ II ⍀ WERTE
DATEN[V;]  ,  (+ /[2] VV)
```

where replaced by `DATEN ({⍺ , + ⌿ ⍵} ⌸) WERTE`).

Similarly the enhanced abilities of Iota `⍳` as Index Of for matrices (and not only vectors) where used to replace the kernel of other base utilities implementing just that, namely searching and indexing arrays of rank 2, like in

```
IND ← (1 ↑ ⍴DATEN) ⍴ 1  ⋄  N_TR ← 1 + 1 ↑ ⍴REF
LISTE ← REF ,[1] DATEN
Z_NR ← (- ⍳ 1 ↑ ⍴REF)  ,  (⍳ 1 ↑ ⍴DATEN)
V ← ⍋LISTE
LISTE ← LISTE[V;]  ⋄  Z_NR  ← Z_NR[V]
BEG ← (1 , ∨ / (1 ↓[1] LISTE) ≠ (¯1 ↓[1] LISTE))
                                                  / ⍳ 1 ↑⍴LISTE
V ← (+ \ 0 < Z_NR)  [¯1 + 1 ↓ BEG , (1 + ⍴Z_NR)]
V ← V - 0 , ¯1 ↓ V
IND[(0 < Z_NR) / Z_NR] ← V / N_TR - N_TR | 0 ⌊ Z_NR[BEG]
```

by `IND ← REF ⍳ DATEN`.

The case of Iota Underbar `⍸` as Where is a bit different. There are countless examples in functions of all kinds, where a boolean mask was applied to a Index Generator of some axis of an array to select relevant indexes like in `I / ⍳ ↑ ⍴I`. The simplification with the use of the new primitive into `⍸I` was very easy to code while migrating. Use in a check may look like

```
:If ∨ / (I ← ~ BEARB ∊ V)
    ERR_MSG ← 'Unerlaubte Anweisung(en) in den Spalten '
              , (⍕ ⍸I) , ' "' , (⍕ I / BEARB)
              , '" angegeben, erlaubt:' , (⎕UCS 13) , ⍕V
    →ENDE
:EndIf
```

whereas an index selection may utilise

```
:For NR :In ⍸REL_I
...
:EndFor
```

Some other primitives like At @ as At or Star Dieresis ⍣ as Power also found their way in the migrated code. The use of At is mostly trivial, replacement of indicated positions with fixed values

```
FN ← {⎕UCS (32 @ (⍸ V ∊ 9 10 13 14 133)) (V ← ⎕UCS ⍵)}
(I  / ,DATENS[;V]) ← FN¨ I  / ,DATENS[;V]
(II / ,DATENI[;V]) ← FN¨ II / ,DATENI[;V]
```

but there are some somewhat more substantial examples like

```
(VV VVV) ← (⊂¨ '<ignorieren>' '<kein>')
               ({⍺} @ (⍸ ~ LIST_DEF ∊ LISTS_DEF))¨  VV VVV
```

or

```
IND ← TYPES ⍳ (⍉ (~ @ 4) ⍉TYPES)
```

Power is used for case distinctions during clean-up like

```
V ← ('unbedingt abbauen' '' CnP.CONNR)
                          (##.ado.CONNECT ⍣ VERB_LOK_I)  ''
```

or in dfns for parameter processing

```
(DEF  STSP   GBSP GESP) ← ,¨
          {(⊂ ⍣ (((⎕DR ⍵) ∊ 80 160 320) ∧ (0 < ≢⍵))) ⍵}¨
                                      DEF  STSP   GBSP GESP
ZSP ← , {(⊂ ⍣ ((⎕DR ⍵) ∊ 80 160 320)) ⍵} ZSP
```

The only “real” use at the moment is as part of an automated check...

An open point is the use of Squad ⌷ as Index. Under APL+Win indexing was done almost exclusively via bracket notation. (This had historic reasons, very old problems and misunderstandings concerning Squad under z/OS APL2 and APL+Win.) Probably the best strategy is to wait until Dyalog implements the proposed new selection primitive and then decide on indexing conventions.

3.1.4 System functions

Similarly to new or different Dyalog primitives, system functions not available under APL+Win were used to improve code during migration. A case of tidying up was the replacement of the three generations of APL+Win system functions for component files with the one set provided by Dyalog.

The MS File Scripting Object was used for existence checks and erasure of files under APL+Win. More specifically some small, self-written cover functions were written for this purpose, using methods like

```
V ← 'fso' ⎕wi 'Create' 'Scripting.FileSystemObject'
EX_DR  ← 'fso' ⎕wi 'XDriveExists'      (↑ DATN)
EX_DAT ← 'fso' ⎕wi 'XFileExists'         DATN
PATH   ← 'fso' ⎕wi 'XGetParentFolderName' DATN
BASE   ← 'fso' ⎕wi 'XGetBaseName'        DATN
'fso' ⎕wi 'XDeleteFile'   OBJN
'fso' ⎕wi 'XDeleteFolder' OBJN
```

During migration their use was replaced by ⎕NEXISTS and ⎕NDELETE respectively, enhancing readability. ⎕NPARTS is also used in this context to analyse file names.

Similarly the "compact" system functions ⎕NGET and ⎕NPUT replaced code parts that bound a native file, read/processed contents and freed it or (respectively) bound a file, appended text and freed it. Sequences like

```
NNR ← ¯1 + ⌊ / 0 , ⎕nnums , ⎕xnnums
DATN ⎕xntie NNR
→ (0 = (V ← ⎕nsize NNR)) / ENDE
DATEN ← ⎕nread (NNR , 82 , V)
⎕nuntie NNR
DATEN ← (~ ((1 ↓ II , 0)   ∨   (II ← (0 , ¯1 ↓
           ⎕tcnl = DATEN)  ∧  (⎕tclf = DATEN)))) ⊂ DATEN
DATEN ← (- + /¨ ∧ \¨ ' ' =¨ ⌽¨ DATEN) ↓¨ DATEN
```

where replaced by `PROT ← ↑ 1 ⊃ ⎕NGET (DATN 1)`. There are some small problems with the approach. The first are text matrices as well as encoding errors and the second borderline file types.

In the first case Dyalog is considering enhancements. The second case concerns mainly files that shall not contain a line feed or other marker at the end of the *last* line, for example a file to be bulk inserted into an SQL Server database or one that contains the comment for a SVN revision. Here the old way of coding must be used, because the file does not fulfil the requirements of the new system functions. Dyalog has provided a corresponding Variant option, but it has not been tested yet…

The newer system function ⎕DT is used for simple date arithmetic, replacing self-written functions containing algorithms with case distinctions for different months and for leap years. Calls like

```
DATUM_VOR ← ← 2 ⊃ 2 DATE ¯1 + (2 ⊃ DAYS DATUM)
```

for subtracting a day where replaced by snippets as in

```
DATUM_VOR ← 100 ⊥ 3 ↑ ⊃ 1 ¯1 ⎕DT  ¯1
                          +  ¯1 1 ⎕DT ⊂ 1E4 1E2 1E2 ⊤ DATUM
```

The system function is also used for checking (GUI-) input, done via own, primitive algorithms in the old code, with the help of the dfn

```
DATE_ISO ← {
                     V ← (⊂ ⍣ (80 = ⎕DR ⍵)) ⍵
                     I ← 80 = ⎕DR¨ V
0 = ≢ (VV ← I / V) : I
     I ← I \ (⊂ ,0) ≡¨ ('\A\d{4}-\d{2}-\d{2}\Z' ⎕S 0)¨ VV
0 = ≢ (VV ← I / V) : I
                     VV ←  2 ⊃¨ ⎕VFI¨ ('-' ⎕R ' ') VV
          I  \  (0 < ⊃¨ VV) ∧  (0 ⎕DT 1 + 2000 | ¯1 + VV)
           }
```

It would be nice, if the system function could also work with ISO-formatted date fields and/or accept dates after 4000-02-28.

A major change in coding, that has just started, is the use and incorporation of regular expressions in the code, facilitated by the system functions ⎕S and ⎕R. Of course this encompasses more than just swapping a code snippet for a new one. Regular expressions are a different way of thinking about patterns than, say, Epsilon Underbar ⍷ as Find. Usage includes some trivial replacements like

```
:For V :in 'SOLL' 'IST'   'SOLL-Duplikate' ...
    SNL ,← ⊂ ('<art>' ⎕R V) CnP.SNT
:EndFor
```

as well as some a bit more complicated like

```
:For (V VV) :In ('\A\s*' '')  ('\s*\z' '')  ('\s+' ' ')
                            ('\s*(?=[\(\)\=\+\-\<\>,])' '')
                               ('[\(\)\=\+\-\<\>]\K\s*' '')
    ⎕EX 'FN'   ⋄  FN ← V ⎕R VV
    (I  / ,DATS[;VVV]) ← FN¨ I  / ,DATS[;VVV]
    (II / ,DATI[;VVV]) ← FN¨ II / ,DATI[;VVV]
:EndFor
```

One difficulty is deciding when APL-methods are the right way to proceed and when regex. A disadvantage of the latter is that empty arrays can lead to problems and must by circumvented through conditional execution. Also the performance may decline dramatically when using longer strings.

There are also some APL+Win system functions that do not exist under Dyalog. The one most used in the old code was ⎕VGET. It gave access to the value of a variable at different stack levels. ⎕STATE can do something similar under Dyalog with some restrictions. However the code parts using it dealt with ambiguous semi-globals.

The setting was usually something like the following. FOO calls GOO which calls HOO. The latter needs to access LIST in the context of FOO, without knowing if it is localised in GOO and in spite of it being local to HOO itself. In those case the concept was changed and the code rewritten using complete namespace paths. Now #.ws3.ns3.HOO searches for #.ws1.ns1.LIST because FOO resides in **#.ws1.ns1**. The case where both FOO and HOO reside in the same namespace is neither supported any more nor practically needed.

Another possible (but not actual) use of those system functions would have been reduction of Hydrant ⍎ as Execute in the case of variable names that are set at runtime. Major changes of the sort will be done when the proposed system functions ⎕VSET as well as ⎕VGET are implemented by Dyalog.

3.2 Object Oriented Programming with APL syntax

3.2.1 Classes

The old code contains almost no Classes. The main reason is a specific decision. Functional and object oriented programming do not mix too well. The main applications implemented deal with large numbers of contracts all at once and are not well suited to the OOP paradigm. That fore Classes were restricted to the specific case where more than one instance of something may be needed at once.

There are a few utilities fulfilling the criterion, not much more than (visible) running protocols and statistics. The code controlling those Classes had to be more or less rewritten from scratch, as APL+Win supports OOP only rudimentary. It allows only the modification of existing Windows Objects, with Instances residing “somewhere” in the workspace. For Classes that are based on a Form this was sufficient. Still the controlling functions could not really be migrated.

Under Dyalog proper Classes where created. Each Instance contains a Form. Whereas under APL+Win the name of the “Instance” was passed between functions, under Dyalog it is a reference. Using a Method or Property requires a syntax completely different from the old one. That means that the use had to be adjusted everywhere, causing a significant migration workload.

A tricky point was, that APL+Win automatically provided Instances that could “survive” the end of a (main) function and then be forced to auto-destroy by pressing a Button on the Form itself. The concept of Class does not provide a canonical method for the auto-destruction of an Instance. The solution was an additional, global, Instance-depended reference in a well defined namespace path, and internal Methods that create or remove those references.

Because “migration” already meant “rewriting”, some Methods that were missing under APL+Win were added, notably one for timestamping messages — the most common use. That reduced code in many functions and made it more readable. Similarly the usual way for asking the user to decide if to proceed or abort was encoded in another new method. An optional timeout was added in order to allow for interaction-free automated tests. Some trivial examples with a lapsed decision timeout of $3s$ demonstrate the usage

```
      ⎕NC ⊂'#.div.udc.MESSAGE_BOX'
9.4
      msgb ← ⎕NEW #.div.udc.MESSAGE_BOX
      msgb.Message ← ''
      msgb.TimeStampMessage 'What''s up?'
      msgb.Message
```

```
 2024-08-15 12:02:30  What's up?
    msgb.WaitOn 'Decision' 'Decide!' 3
    ↑ msgb.Message
2024-08-15 12:02:30
What's up?
2024-08-15 12:04:01
Fortfahren durch fehlende Benutzeraktion nach
                                                        Entscheidung
```

3.2.2 Schematic GUIs

The implemented applications use some interaction with the user, mostly of a rather schematic kind. The reason is, that most of the processes involved consist of a more or less lengthy run that has to be started with appropriate parameters. Most workspaces therefore need a main GUI offering some Buttons, each starting one of the main function. Each of those offers the opportunity to interactively decide on run parameters.

The main GUIs were implemented in very similar ways in each of the old APL+Win workspaces. The parameter GUIs were only described (names, types, defaults, positions, . . .) in each workspace, the interactive part was outsourced to a utility function. All those functions had to be more or less rewritten, because of the differences in the interaction with Windows. That fore some additional changes were made.

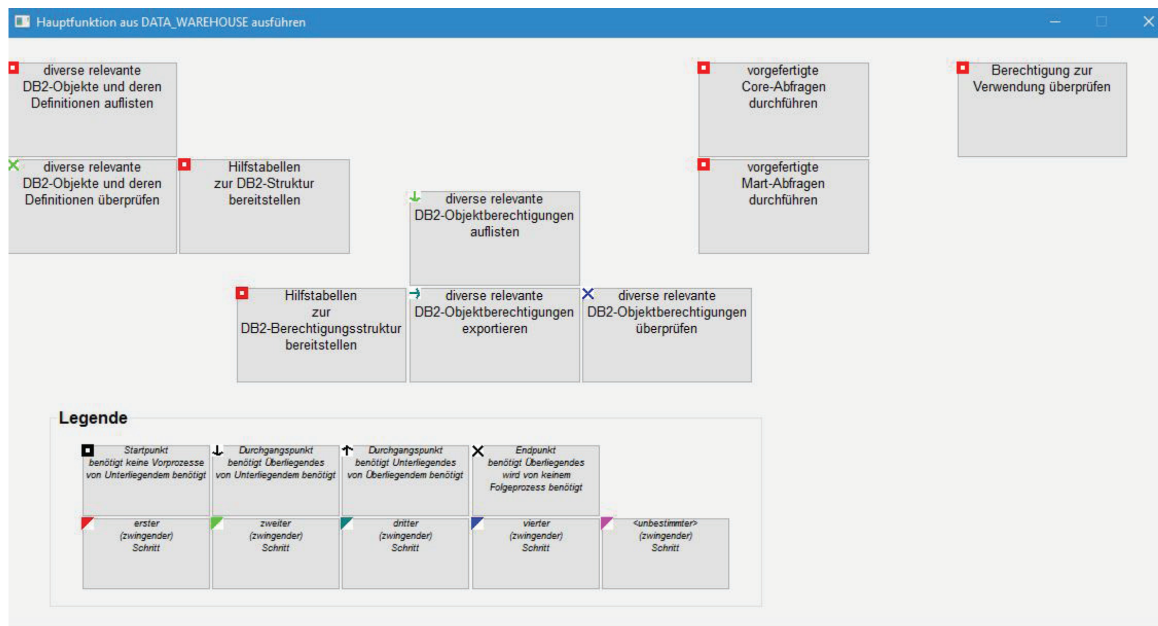
For each type of GUI (main, parameter) one controlling utility was written. It is not a Class, because it is expected that only one instance of each GUI will exist at any time. Direct access to the namespace is used on the Dyalog side, with very few exception where only one of the old style system functions ("⎕WC") provided the necessary functionality. For example the creation of objects looks like the following. (Remember that some properties may only be set during creation.)

```
obj ← F ← ⎕NEW 'Form'  (('Sizeable' 1)  ('SysMenu' 1)
           ('MinButton' 1)  ('MaxButton' 1)) ⍝ only here!
obj.(Caption Coord CursorObj Moveable) ← FD.Caption
                                           'RealPixel' 0 1
obj.onClose     ← 1
obj.onConfigure ← 'FD.Configure'
obj.on9999      ← 1

obj ← F.(TF ← ⎕NEW ⊂'TipField')

obj ← F.(SF ← ⎕NEW ⊂'SubForm')
obj.(Posn Size) ← (0 0)  (¯50 0 + F.Size)
```

All GUIs use, under Dyalog, relative coordinates at all object levels. Some older options for setting coordinates had never been used and where removed. The necessary functionality of multiple Grids was added. Screenshots 6 and 7

show respective examples. Some additional points like creating needed file paths were added to the utility. As a result all places where GUIs where used had also to be overhauled and became significantly shorter. One thing did not change, the utilities are called with GUI *names*, the GUI Instance and its containing namespace exist *only* within the utilities.

Figure 6: Example of schematic main GUI



3.2.3 COM-interfaces

COM interfaces are exposed as Classes. Under APL+Win the use could be cumbersome, because it did not fit well into the rest of the language. Instances were treated more or less like Windows objects and some tricks like "re-directional syntax" were needed to get hold of children of objects. The main use of COM is Excel and ADO (for database connections and SQL statements). All associated code parts had to be rewritten more or less completely.

Fortunately the interaction with COM is strictly contained in utilities. The new code uses the APL syntax for namespaces provided by Dyalog and is more compact and much easier to read. For example the use of redirectional syntax in APL+Win

```
⎕wi 'wb.xStyles > xl.wb.sts'
⎕wi 'wb.sts.xItem > xl.wb.sts.nrm' 'Normal'
⎕wi 'wb.sts.nrm.xFont > xl.wb.sts.nrm.fnt'
⎕wi 'wb.sts.nrm.fnt.xName' 'Arial'
⎕wi 'wb.sts.nrm.fnt.xSize' 10
```

was replaced by namespace syntax in Dyalog

```
wb.Styles.Item[⊂'Normal'].Font.(Name Size) ← 'Arial'  10
```

Figure 7: Example of schematic parameter GUI

The function calls of the utilities did not change. As before the underlying Instances are not passed as a reference. Instead some numbers, workbook, sheet and range or connection and recordset, are given to the utilities. Those numbers unambiguously define a global object name that is a reference to an Instance.

A perspectively important point was the introduction of "Microsoft Information Protection" (MIP) in the corporation immediately after migration. It enforces the classification, by the creator, of each and every Excel workbook according to its content. The implementation in the COM interface on the side of Microsoft is rather opaque and not well documented. The implementation on the Dyalog side was done by a new developer, brightening the long term perspective for APL in ERGO health significantly!

In the long term the future of those interfaces is questionable. Excel interaction could be implemented through .Net (or even pure APL), SQAPL could by used instead of ADO for databases. However the use of Active Directory (AD) that is exposed by ADO as a non-relational database, could not be facilitated by SQAPL. Of course any change would be confined to the relevant utilities.

Neue Funktionen in GNU APL

Jürgen Sauermann
mail@juergen-sauermann.de

In den letzten Wochen hat es einen Schub neuer Funktionen in GNU gegeben. Die meisten dieser Funktionen wurden von einem GNU APL Benutzer (Henrik Moller) beigesteuert, während die übrigen vom Autor stammten. Alle Funktionen betreffen den Bereich Lineare Algebra. Die neuen Funktionen wurden erst nach dem GNU APL 1.9 Release entwickelt und werden daher erst mit dem GNU APL 2.0 Release offiziell werden. Die meisten GNU APL Benutzer benutzen jedoch **git** oder **subversion** und können so die neuen Funktionen schon heute verwenden.

Bevor wir zu den Funktionen im Einzelnen kommen, scheint es angebracht, das Konzept der Funktions-Gruppen in GNU APL zu erläutern.

Exkurs: Funktionsgruppen in GNU APL

Kurz nach der Fertigstellung des GNU APL Kern-Interpreters, also der Summe der standardisierten APL Funktionen, stellte sich das Problem. dass noch keine File Operationen (File öffnen, File schreiben, File schreiben, File schließen, usw.) vorhanden waren. Der Blick auf einen alten APL1 Interpreter (lies: APL 68000 von MicroAPL) zeigte, dass dort diese Funktionen durch eigene APL Symbole (⍐, ⍗, ⍈ und ⍇) repräsentiert wurden. Dieser Ansatz, also für neue Funktionen neue APL Symbole zu verwenden, schied aber aufgrund seiner Seiteneffekte (⎕AV, Tastatur, Fonts) als Erstes aus.

Die nächste Überlegung, für jede File Operation eine eigene Systemfunktion (⎕OPEN, ⎕READ, ⎕WRITE, ⎕CLOSE, usw.) einzuführen, wurde aufgrund des damit einhergehenden Implementierungs-Aufwandes verworfen.

Die Lösung bestand seinerzeit darin, eine einzelne Systemfunktion **⎕FIO** (für File-I/O) für alle File Operationen einzuführen, wobei die einzelnen File Operationen über das numerische Achsen-Argument von **⎕FIO** ausgewählt wurden. Die entsprechende Syntax, die noch heute für alle Systemfunktionen mit mehreren Sub-Funktionen gültig ist, war damals:

```
Zi ←    ⎕FIO[ 1] ''        ⍝ errno (of last call)
Zs ←    ⎕FIO[ 2] Be        ⍝ strerror(Be)
Zh ← As ⎕FIO[ 3] Bp        ⍝  fopen(Bs, As) filename Bp mode As
Zh ←    ⎕FIO[ 3] Bp        ⍝ fopen(Bs, "r") filename Bp
Ze ←    ⎕FIO[ 4] Bh        ⍝ fclose(Bh)
Ze ←    ⎕FIO[ 5] Bh        ⍝ errno (of the last call using Bh)
Zb ←    ⎕FIO[ 6] Bh        ⍝ fread(Zi, 1, 5000, Bh) 1 byte per Zb
Zb ← Ai ⎕FIO[ 6] Bh        ⍝ fread(Zi, 1, Ai, Bh) 1 byte per Zb
Zi ← Ab ⎕FIO[ 7] Bh        ⍝ fwrite(Ab, 1, ⍴Ab, Bh) 1 byte per Ab
```

Wie man sieht, können einzelne Sub-Funktionen sowohl monadisch als auch dyadisch sein. Die Valenz war also eine Eigenschaft der Sub-Funktion und nicht von **⎕FIO** selbst.

Was kurz danach geschah war, dass GNU APL Benutzer einen Workspace namens **FILE_IO.apl** entwickelten, in dem für jedes numerische Achsen-Argument von ⎕**FIO** eine separate, und in der Regel einzeilige, APL definiert war. Also zum Beispiel:

```
∇Zi ← FIO∆errno
⍝⍝ errno (of last call)
 Zi ← ⎕FIO[1] ''
∇


∇Zs ← FIO∆strerror Be
⍝⍝ strerror(Be)
 Zs ← ⎕FIO[2] Be
∇
…
```

Dieser **FILE_IO.apl** Workspace konnte dann per)**COPY** eingefügt werden und führte dazu, dass der APL Code lesbarer wurde als Code mit numerischen Achsen-Argumenten. GNU APL hat dann diese Idee aufgegriffen, indem als Alternative zu numerischen Achsen-Argumenten auch Strings (mit Name für die Subfunktionen) verwendet werden konnten. Die Syntax war dann z.B.:

```
Zi ←    ⎕FIO['errno'] ''
Zs ←    ⎕FIO['strerror'] Be
Zh ← As ⎕FIO['fopen'] Bp
Zh ←    ⎕FIO['fopen'] Bp
Ze ←    ⎕FIO['fclose'] Bh
Ze ←    ⎕FIO['errno_B'] Bh
Zb ←    ⎕FIO['fread'] Bh
Zb ← Ai ⎕FIO['fread'] Bh
Zi ← Ab ⎕FIO['fwrite'] Bh
```

Ein Nachteil von sowohl numerischen als auch String Argumenten war noch der, dass das Achsen-Argument ein APL Wert ist. APL Strings, also APL Vektoren deren Elemente Buchstaben sind (wie 'errno', 'strerror' usw.) sind ungleich komplexer als interne Strings im GNU APL Interpreter. Daher ist ihre Verarbeitung auch langsamer. Außerdem war die Syntax nicht wirklich elegant. Der letzte Schritt in der Entwicklung der Syntax von Funktionsgruppen wie ⎕**FIO** bestand daher in der vollständigen Eliminierung von APL Werten (und damit von Achsen-Argumenten). Die neue Syntax benutzt den `'.'` Operator in Kombination mit der Funktionsgruppe als linkem Argument. Im Falle von ⎕**FIO**:

```
Zi ←    ⎕FIO.errno ''
Zb ← Ai ⎕FIO.fread'] Bh
Zi ← Ab ⎕FIO.fwrite'] Bh
Zs ←    ⎕FIO.strerror Be
Zh ← As ⎕FIO.fopen Bp
Zh ←    ⎕FIO.fopen Bp
Ze ←    ⎕FIO.fclose Bh
Ze ←    ⎕FIO.errno_B Bh
Zb ←    ⎕FIO.fread Bh
```

Die neuen APL Funktionen

Zurück zu den neuen Funktionen. Sie verteilen sich auf 2 Funktionsgruppen **⎕MX[ ]** und **⌹[ ].** Die Systemfunktion **⎕MX** (für **M**atri**X** operationen) ist nei und wurde von Henrik Moller beigesteuert. Das APL Primitive ⌹ ist wohlbekannt und wurde nur um Varianten mit Achsen-Argument erweitert.

Die Funktionsgruppe ⎕MX[ ]

Die Funktionsgruppe **⎕MX[ ]** besteht aus z.Zt. 14 Funktionen:

```
    ⎕MX.determinant B          ⍝ (monadic) Determinant of B
{A} ⎕MX.cross_product B        ⍝ (nomadic) Cross product
  A ⎕MX.vector_angle B         ⍝ (dyadic)  Angle between vectors A & B
    ⎕MX.eigenvector B          ⍝ (monadic) Eigenvector(s) of B
    ⎕MX.eigenvalue B           ⍝ (monadic) Eigenvalue(s) of B
    ⎕MX.ident B                ⍝ (monadic) B×B Identity matrix
    ⎕MX.rotation_maxtrix B     ⍝ (monadic) Rotation matrix
  A ⎕MX.homogeneous matrix B   ⍝ (dyadic)  Homogeneous matrix
    ⎕MX.norm B                 ⍝ (monadic) Norm of B
{A} ⎕MX.randoms B              ⍝ (nomadic) Random numbers
{A} ⎕MX.covariance B           ⍝ (nomadic) Covariance
  A ⎕MX.histogram B            ⍝ (dyadic)  Histogram
  A ⎕MX.print B                ⍝ (dyadic)  Print B to file A
    ⎕MXset_rng_seed B          ⍝ (monadic) Set RNG seed
```

Die Details der einzelnen Funktionen sind in der GNU APL Dokumentation (siehe **https://www.gnu.org/software/apl/apl.html**) erklärt so dass sich eine Wiederholung an dieser Stelle erübrigt.

Die Funktionsgruppe ⌹[ ]

Die Funktionsgruppe **⌹[ ]** besteht aus den folgenden 6 Funktionen. Jede dieser monadischen Funktionen berechnet eine Faktorisierung der reellen oder komplexen Matrix **B**.

```
(Q R Ri) ← ⌹.qr_fact_helzer B    ⍝ QR factorization of B (Helzer algorithm)
(Q R Ri) ← ⌹.qr_fact_gsl B    ⍝ QR factorization of B (libgsl algorithm)
(Q R Ri) ← ⌹.rq_fact B        ⍝ RQ factorization of B (libgsl algorithm)
(Q R Ri) ← ⌹.lq_fact B        ⍝ LQ factorization of B (libgsl algorithm)
(Q R Ri) ← ⌹.ql_fact B        ⍝ QL factorization of B (libgsl algorithm)
(P U L)  ← ⌹lu_fact B         ⍝ LU factorization of B (libgsl algorithm)
```

Dabei gilt:

- **Q** ist orthogonal (also $\mathbf{Q}^{-1}$ = ⍉ **Q**),
- **R** ist eine (obere) Dreiecksmatrix,
- **Ri** ist $\mathbf{R}^{-1}$, also die inverse Matrix von **R**,
- **P** st eine Permitation (von **B**)
- **U** ist eine (obere) Dreiecksmatrix, und
- **D** ist eine (untere) Dreiecksmatrix.

Auch diese Funktionen finden sich unter https://www.gnu.org/software/apl/apl.html.

Alle neuen Funktionen (mit Ausnahme von **⌹[1]**) basieren auf der C Bibliothek **libgsl** (aka. **G**NU **S**cientifi **L**ibrary) die installiert sein muss, damit die Subfunktionen verfügbar sind. Die Subfunktion **⌹[1]** basiert hingegen auf dem in APL Quote-Quad 1990 veröffentlichten APL Algorithmus von Gary Helzer.

Development in Dyalog APL with Modern Tools

Kai Jäger

kai@aplteam.com

I gave a talk regarding my personal workflow when developing in Dyalog APL at the Dyalog User Meeting 2024 and APL Germany's 2024 autumn meeting.

On both occasions I had just 30 minutes at my disposal, which forced me not only to be pretty fast but also to leave out some details I would have preferred to mention. Therefore, I take the opportunity to add some information to the talk.

The talk is available online[1].

First things first, it all starts with preferences: I use external editors like VS Code rarely; I prefer to work from within APL. In fact, most of the time, I modify and add code directly from the Tracer.

The option to modify/add code while running it is, together with `⎕WC`, the reason why APL is still around. When you walk through the offices at SimCorp, where far more than 100 APL developers are working, you see the Tracer everywhere.

Version control

In today's world, every professional developer is using software for version control, be it Git, SubVersion, or anything else. I am using Git for two reasons: First, my last two clients, Dyalog and Carlisle Group, use Git; second, GitHub offers a wide range of very useful features. I dislike the complexity[2] of Git, but the many useful features offered by GitHub make up for that.

I started using GUIs for doing Git business, but that forced me to change between APL and the GUI, and I also found it difficult to find help when the GUI was not offering what I needed because practically all the help refers to the Git shell commands. In the end, I created a set of user commands named `]APLGit2` that allow me to gather information from Git from within APL, and I do everything that changes data or is complex from the Git shell. That approach also comes with the benefit of making me a true Git professional in the long run, although it will still take considerable time ;)

Search and Replace

One of my most essential tools is Fire — short for “FInd and REplace.” Its search capabilities are sophisticated and tailored to the needs of an APL programmer. For instance, Fire allows you to filter out comments or text, scanning either only the code or only the comments.

This already makes Fire a powerful asset for any APL programmer. The only drawback is that Fire is currently available only on Windows, but this is likely to change in 2025.

Fire’s real strength, however, lies in its ability to replace text across multiple objects. This capability is invaluable when modifying a large number of items. Fire also generates detailed reports on the potential impact of each replacement, ensuring full control over the process.

Comparing APL Objects and Files

Comparing APL objects—whether from different development stages or across various commits — is frequently necessary, but the abundance of comparison tools can complicate the process.

To address this, I developed the `CompareFiles` project, which serves as a generalized interface for comparison utilities[3]. It supports popular tools like BeyondCompare, Meld, WinMerge, and CompareIt!. Adding new utilities is simple, allowing developers to integrate their preferred tools.

This project enables seamless integration of comparisons within user commands such as `]APLGit2` or `]CompareWorkspaces`.

Testing

For testing, I rely on the `Tester2` class[4]. While I've heard complaints about its complexity, several simpler test frameworks have also been attempted. `Tester2` is designed for testing applications, not (necessarily) single functions, meaning that complexity becomes inevitable. Indeed, some of the initially simpler solutions have gradually expanded, eventually matching `Tester2` in terms of complexity.

The complexity can be viewed in two ways:

1. **Internal Complexity**

   `Tester2` is indeed intricate under the hood.

   However, because the project is actively maintained, users can safely overlook this. Reported issues are addressed promptly.
2. **Usage Complexity**

   Understanding the basic concepts isn't as intuitive as it could be. The documentation covers every feature extensively, which can be overwhelming for new users. To mitigate this, videos specifically aimed at beginners will be produced in 2025.

Code Coverage

When testing an application, it's crucial to know which parts of the code are covered by tests and which aren't. `Tester2` automatically checks for the presence of the `CodeCoverage` class[5] and offers to use it if available.

This integration makes it easy to identify which parts of an application are fully covered by tests, which are only partially covered, and which lack coverage entirely.

While `CodeCoverage` does face some technical limitations that cannot be resolved without changes to the Dyalog interpreter or `⎕PROFILE`, it remains a highly valuable tool for analysing test coverage.

`]GitHub`

For the time being, this group of user commands offers just three commands:

`]GitHub.GoToGitHub` opens the homepage of the given repository on GitHub.

`]GitHub.ListIssues` returns a list of open issues for the given owner/project.

`]GitHub.ListRepos` returns a list of all repositories for the given owner.

I am planning to add more commands in 2025: at the very least, I want to be able to create and close issues directly from within APL.

`]CodeBrowser`

The user command `]CodeBrowser` is designed to support code reviews by making it easy to hide and show code blocks and jump between locations.

`]Latest`

The user command `]Latest` [6] lists, by default, all objects that got changed on the last day any object was modified.

Parameters allow you to specify either the exact number of objects or the exact number of days.

It's the perfect tool to jog your memory if you haven't touched a project for a while and want to continue where you left off.

`]Snippets`

The user command `]Snippets` [[7]] allows you to manage functions, operators, scripts, and variables, and even simple pieces of APL code that do not qualify as a package but are too complex to input manually on demand.

This command lets you save, inspect, list, and import snippets into the workspace.

The package manager Tatin

Being able to load packages implemented by third parties while leaving the management of dependencies to the package manager will be essential for the future of APL. The Tatin package manager[[8]] addresses this.

Also, Tatin offers an easy and straightforward path to updating packages.

All user commands mentioned are available as Tatin packages. That makes installing them easy. This makes installing them easy.

The way user commands are managed will change, not in 20.0 but one release later. Since the details have not yet been finalised, we can only discuss how to install user command packages in 18.2, 19.0, and 20.0.

Installing Tatin packages as user commands

The easiest way to install a user command package on any of the supported platforms isto install them into the folder `MyUCMDs/`. Tatin understands the alias `[MyUCMDs]`; therefore, to install, say, the user command `]Latest`, the following suffices:

```
]Tatin.InstallPackages latest [myucmds]
```

Because no registry is specified, it will load the package from the principal Tatin Registry[[9]].

Details are discussed in an article on the APL wiki: “Dyalog User Commands”[[10]].

Footnotes

1. The talk online:
   https://dyalog.tv/Dyalog24/?v=aIqDxwlcoVU
2. Regarding Git's featurism:
   https://www.interaction-design.org/literature/book/the-glossary-of-human-computer-interaction/featuritis-or-creeping-featurism
3. The user command “CompareFiles”:
   https://github.com/aplteam/CompareFiles
4. The user command “Tester2”:
   https://github.com/aplteam/Tester2
5. The user command “CodeCoverage”:
   https://github.com/aplteam/CodeCoverage
6. The user command “Latest”:
   https://github.com/aplteam/Latest
7. The user command “Snippets”:
   https://github.com/aplteam/Snippets
8. The Tatin package manager:
   https://github.com/aplteam/Tatin
9. The URL of the principal Tatin Registry:
   https://tatin.dev
10. Article regarding Dyalog user commands on the APL wiki:
   https://aplwiki.com/wiki/Dyalog_User_Commands

www.apl-germany.de

APL Germany e.V.