5

AN OUTLINE OF ARIZONA

by

John Alan McDonald

TECHNICAL REPORT No. 131

July 1988

Department of Statistics, GN-22

University of Washington

Seattle, Washington 98195 USA

An outline of Arizona. *

John Alan McDonald, Dept. of Statistics, U. of Washington

July 29, 1988

1 Introduction

This paper outlines a system called Arizona, now under development at the U. of Washington. Arizona is intended to be a portable, public-domain collection of tools supporting scientific computing, quantitative graphics, and data analysis, implemented in Common Lisp[31] and CLOS (the Common Lisp Object System)[4].

Although there is substantial implementation of some of the modules described below, this paper is more a description of a design than of an actual program. One excuse for writing a paper on not-yet existing software is that Arizona is intended primarily as a research vehicle: it is hard to predict when, if ever, it will mature and stabilize to the point of robust production-quality code. However, we hope that the ideas embodied in its design are of interest in themselves and of use in future scientific computing and data analysis systems (eg. a "New New S"[2]).

Discussion of the philosophy underlying Arizona can be found in [22,23,18,21,24,32]. Briefly, the design is motivated by our belief that an ideal system for scientific computing and data analysis should have:

- One language that can be used for both for line-by-line interaction or defining compiled procedures.
- Minimal overhead in adding new compiled procedures (or other definitions).
- A language that supports a wide variety of abstractions and the definition of new kinds of abstractions.
- Programming tools (editor, debugger, browsers, metering and monitoring tools).
- Automatic memory management (dynamic space allocation and garbage collection).

*This research was supported by the Office of Naval Research under Young Investigator award N00014-86-K-0069, the Dept. of Energy under contract FG0685-ER2500. (The system has benefited from ideas (and sometimes code) contributed by many people, including Rick Becker, Andrew Bruce, Andreas Buja, Pat Burns, John Chambers, Bill Dunlap, Robert Gentleman, Peter Huber, Catherine Hurley, John Michalak, Wayne Oldford, Jan Pedersen, Steve Peters, Werner Stuetzle, and Alan Wilks.)

- Portability over many types of workstations and operating systems.
- A community of users and developers.
- Access to traditional Fortran scientific subroutine libraries or equivalents.
- A representation of scientific data directly in the data structures of the language.
- Comprehensive numerical, graphical, and statistical functionality.
- Device independent static output graphics.
- Window based interactive graphics.
- Support for efficient and concurrent access to large databases.
- Documentation and tutorials, both paper and on-line.

The first nine points (through "access to Fortran") come for free with standard Common Lisp environments. The remaining six are the research aspects of Arizona.

Because of limitations of space, for the rest of this paper we are assuming that the reader is familiar with Common Lisp and CLOS or, at least, Lisp and object-oriented programming in general. Others who wish to read this paper should review some of the references first.

1.1 The modules

Arizona is divided into a number of modules, with limited interdependencies, to permit individual modules to stabilize and be "released" before the whole system is complete.

The modules are divided into two groups: a numerical, quantitative kernel and an interactive, window-based, scientific graphics part.

The non-graphical quantitative kernel is more developed at present, because it can be implemented in an efficient, portable way using existing standards for Common Lisp and CLOS. The quantitative kernel consists of:

- *Basic Math*, which requires Common Lisp,
- *Collections*, which requires Common Lisp and CLOS,
- *Linear Algebra*, which requires Basic Math and Collections,
- *Probability*, which requires Linear Algebra,
- *Database*, which requires Collections, and
- *Statistics*, which requires Database and Probability.

The current design for the graphics part is fairly tentative. Implementation of a portable scientific graphics toolkit requires a standardized interface between Common Lisp/CLOS and the large variety of proprietary or proposed standard window systems for workstations and personal computers (eg. Symbolics Genera [36], NeWS[33], X[29], etc.). This standard (sometimes called Common Windows) is the subject of intense activity in the Common Lisp community[13,28]. I have identified three modules:

- *Constraints*, which requires Common Lisp and CLOS. (This module might very well be part of the non-graphical kernel, but most of the applications we have in mind at present are in graphics.)
- *Quantitative Graphics*, which requires Common Windows, Collections, Constraints, and Linear Algebra.
- *Data Analysis Graphics*, which requires Quantitative Graphics and Statistics

2 The quantitative kernel

2.1 Basic Math

Basic Math consists of things that can be reasonably implemented with Common Lisp functions and primitive Common Lisp data structures; it does not use CLOS. Included in Basic Math are: machine constants, special functions (eg. beta, gamma) extended vector operations (analogous to the BLAS[15] used in Linpack[8]), evaluation and interpolation (eg. generic continued fractions) 1d numerical integration, and basic random number generators.

2.2 Collections

The Collections module has two parts: *Abstract Sets* and *Enumerated Collections*.

Instances of an Abstract Set class are used to represent one of the sets or spaces that arise in mathematical computing. Examples are `Integer-Interval`, `Float-Interval`, and `Vector-Space`, which are used in the Probability and Linear-Algebra modules.

The Enumerated Collection classes are in part modeled on the Collection classes in Smalltalk-80[10]; instances are used for traditional compound data structures, eg. Trees, Queues, Enumerated Sets, Dictionaries, Indexes, etc. Enumerated Collections are heavily used by the Database module.

An Enumerated Collection basically serves as a framework for iterating over its elements. A simple collection might be represented by a list; more complex collections permit more efficiency for specialized access. (Eg. a time series might use a doubly linked list to give efficient access to lagged observations; discrete data might use an n-dimensional array for quick access to the cells of a contingency table.)

2.3 Linear Algebra

The Linear Algebra module is discussed in detail in [21], where it is referred to as *Cactus*. It provides approximately the same functionality as Linpack[8] and Eispack[30]. However, CLOS allows Cactus to operate at a level of abstraction chosen to match the initial, high-level, geometric descriptions of algorithms given in standard numerical analysis texts[11]. The use of object-oriented programming makes the implementation of standard algorithms (eg. a QR decomposition) easier to understand and modify than the versions in the best Fortran libraries—without sacrificing efficiency in either space or time. In addition, it is much easier to use information about regular structure, patterns of sparsity, etc., to get improved performance in special problems. Also, the higher level of abstraction permits extensions to, for example, computations on Hilbert spaces[14].

The Linear Algebra module provides: class definitions for `Vector-Spaces`, class definitions for `Vector-Transformations` (`Matrix`, `Positive-Definite-Matrix`, `Householder`, `Product`, etc.), methods for the protocol corresponding to the algebra of linear transformations (`transform`, `compose`, `scale`, `add`), methods for "matrix" decompositions (LU, QR, LQ, SVD, eigen, etc.), and the ability to solve systems of linear equations and least squares problems using a generic `pseudo-inverse` function that can be applied to any linear transformation.

2.4 Probability

Inference and Monte Carlo simulation (including Bootstrapping) are supported in a unified framework through a protocol for `Probability-Measure` classes. Probability measure objects are responsible for generating samples from themselves, computing their quantiles, and computing the probabilities of appropriate sets, including tail probabilities. The defined probability measure classes includes the standard one- and higher-dimensional parametric densities and discrete distributions, and non-parametric measures, either resulting from density estimates or the empirical measure of a data set. (It's worth noting that simple descriptive statistics like mean, median, etc., are generic functions in the probability measure protocol and are applied to data sets by viewing them as empirical distributions.)

2.5 Database

The Database module has two parts. The first concerns the representation of statistical data by collections of objects and is fairly well developed. The second is concerned with providing true database facilities: efficient concurrent access to large (gigabyte) collections of objects whose identities persist beyond the lifetime of a particular Lisp address space. The second part is a major research topic in the database and object-oriented programming communities[25,26].

2.5.1 Collections of objects

In most statistical packages, data sets are represented as 2 dimensional arrays of floating point numbers. Each row represents an individual and each column represents a variable. This is an awkward representation, for example, for categorical data, and for data sets with more complicated structure, such as clustering trees. It is impossible to represent simple, but important, contextual information, such as the fact the a negative value for height must be an error or that height at age 2 should be greater than height at age 1. An array representation makes it difficult to sort and select subsets without losing track of important correspondences, such as the fact the row 17 in the array of subsurface coal producers represents the same company as row 25 in the array of all coal producers and average sulfur content is column 3 in subsurface coal producers and column 5 in all coal producers.

In Arizona, statistical data is represented by collections of objects. The advantages of this are discussed in detail in [18]. Individuals are represented by objects, instances of CLOS classes. Variables are represented by generic functions. A dataset is represented by a collection, typically a list or one-dimensional array.

For example, in analyzing energy consumption data for cities in the US, the data on each city would be collected into an instance of the `City` class. A particular instance might look like:
`{City Seattle :population 450000 :cooling-degree-days 300 ···}`.

Statistical variables are represented by generic functions. To get at the values in the slots we use automatically defined accessor functions: `(population {City Seattle})`. The use of generic accessor functions gives a unified way to refer to slots or arbitrary functions of slots; we can ask for `(log-population {City Seattle})`, where `log-population` is the obvious Lisp function.

This might seem inefficient, compared to conventional systems, where defining a new variable means adding a column to an array, because it looks as if we would have to call a procedure every time we wanted a value of the `log-population` variable. However, standard Lisp programming techniques (lazy evaluation and memo-ization [1]) make it possible to represent variables by functions, hide the additional complexity from the user, and so that the `log-population` procedure is not called any more often than is absolutely necessary.

Each object has an identity and existence independent of any collection. So the same object can be in many collections; the unique object `{City Seattle}` would be a member of both **`All-Cities`** and **`Northwest-Cities`**. Similarly, generic functions are defined independently of any collection and can be applied to any object (for which there is a method). The independent identities maintain the important correspondences that can be hard to keep track of in an array based system.

Also, a collection may contain objects of more than one type. For example, in energy production data, it might prove useful to analyze coal and oil producers together, but to define separate coal and oil producer classes—to allow for the fact that `acres-strip-mined` is not a relevant slot for oil producers. In that case, `all-energy-producers` would contain instances of at least two different classes.

2.5.2 Persistent Objects

A true database requires objects that persist beyond the lifetime of the address space in which they were created. Arizona will be used for research into a hierarchy of functionality relating to persistent objects:

1. Making a copy of the current state of an object, in the same address space. (There are some non-intuitive difficulties in this seemingly trivial task; see [27].)
2. Saving objects to disk.
3. Automatic checkpointing.
4. Objects that can undo certain changes.
5. Objects that can recover some number of previous states.
6. Objects that can recover any previous state.
7. Object identities that persist beyond a particular address space (rebooting).
8. Objects that can recover a valid state after catastrophic hardware or software failure[35].
9. Sharing objects by more than one user/address space.
10. Efficient, concurrent access to large, persistent, shared databases.

2.6 Statistics

The Statistics module represents the usual descriptive statistics by generic functions that are thought of as functionals on measures. (All the usual descriptive statistics can be thought of as functionals on measures if we consider a dataset to be a measure with total mass N.)

Simple statistical functionals take a collection and one or more variables (Lisp functions) as arguments. For example: `(median All-Cities #'log-population)`,
where `All-Cities` is a `Collection` of City objects and also an `Empirical-Measure`.

`Median` returns a number; more complex statistical functional return instances of a `Description` class. A `Description` object remembers its training sample and can update itself in response to changes in the training sample. Of particular interest are `Model` objects, which are `Description`'s that are also functions.

For example, least squares linear regression takes as arguments a collection intended as the training sample, a generic function representing the response, and a list of generic functions representing the predictors. The result of the regression is a `Regression-Model` object. The `Regression-Model` object fits itself to the training sample by 1) extracting a linear transformation by applying the predictor functions to each object in the training sample, 2) extracting a response vector by applying the response function, 3) computing a generalized inverse of the transformation via QR or SVD, and 4) applying the generalized inverse to the response vector. The regression model is also a function in the sense that it can be applied to any appropriate object (whether or not in the training sample) to predict a value for the response. In addition, the regression object is able to compute and report appropriate diagnostics and update its fit in reaction to inserting or deleting objects in the training sample or functions in the predictor list.

3 Scientific Graphics

The kernel described in the previous section is useless as a data analysis system—because it lacks any graphics. An important reason for the popularity of systems like S is their convenience and flexibility in showing pictures of data.

Our primary goal is to make it easy to improvise new kinds of plots without losing the performance needed for interactive and motion graphics. The Quantitative Graphics module supports this goal in two major ways: a defining a protocol for the representation of plots by hierarchical display objects and implementing mechanisms for maintaining constraints between the components of a display object (layout constraints) and between a window and the object(s) being shown in the window (viewing constraints).

3.1 Hierarchical Display Objects

We represent a plot as a tree of *Display-Node* objects. Every Display-Node has:

- a *parent* Display-Node. The root of the display has no parent.
- a list of *children* Display-Nodes, which is empty for terminal nodes.
- a *local coordinate system,* chosen to be convenient for describing the appearance or position of the node. For example, the local coordinate system might consist of **xyz** position coordinates, **rgb** color coordinates, a **size** coordinate, a **theta** orientation coordinate, and so on. The coordinate system is represented by an instance of an abstract set class, something like the vector spaces used in the Linear Algebra module.
- *appearance and position parameters* that allow the node to be treated as an element of the local coordinate system.
- a *local viewing transformation,* which takes local coordinates to the local coordinates of the parent. For the root node, it take local coordinates to screen coordinates—that is, pixels and pixel-values representing color. The relationship between the local viewing transformation and the coordinate systems is like the relationship of the linear transformations and vector spaces in the Linear Algebra module.
- a list of *layout constraints* which make assertions about relations between the sizes, shapes, viewing transformations, or local coordinate systems of descendants of the current node.

For efficiency in motion graphics, Display-Nodes may add:

- a *total viewing transformation*, which is obtained by composing all the local transformations between the node and the root of the tree.
- a *factoring* of the total viewing transformation into time-varying and constant parts.
- a *cache* holding the result of applying the constant factor.

For example, many implementations of rotating scatterplots implicitly factor the viewing transformation into constant translation and scaling and a time-varying rotation. If the scaling is chosen carefully, the rotation can be computed in integer arithmetic and produce exact screen coordinates, increasing the speed of rotation by as much as 100 times—at the cost of non-modular, machine specific drawing routines. However, we can implement the basic idea in a modular way, by providing methods for factoring viewing transformations analogous to the matrix decompositions provided by the Linear Algebra module. The same paradigm has been used in higher dimensional graphics[12] and is also applicable when color or shape is changing over time (rather than just position).

For efficient handling of input (deciding which node the mouse is pointing at) Display-Nodes may pre-compute and cache screen coordinates—sometimes of a single pixel, but more frequently of one or more rectangular regions.

Some Display-Nodes are *Presentations*, which means that they serve as a visible representation of some other object in the programming environment—the *subject* of the presentation. (This discussion is very loosely related to the concept of presentation given in [9] and used in the Symbolics Genera system[36] and on the Model-View-Controller user interface architecture used in Smalltalk[7].) For example, a point in a scatterplot is a presentation of a record in a data set.

A presentation is related its subject by a *viewing constraint*, discussed in the next section.

3.2 Constraints

Constraints are abstractions that arise naturally in many statistical, scientific, or graphics problems[1,17,16]. A constraint language allows the programmer to make assertions whose truth is automatically maintained in the course of subsequent computation. Spreadsheets are a widely used, if limited, form of constraint language. A full-fledged constraint language is a major research undertaking in itself [6,34,16]. We intend to implement at least two less ambitious constraints systems:

3.2.1 The Viewing Constraint

The basic idea is that a window is a view of one or more objects and should always show the current state of those objects. We have a fairly good understanding of how to implement this type of constraint. The basic technique is similar to Active Values in LOOPS[5]. The system automatically triggers appropriate computation whenever some presentation's subject is modified. The triggered computation may take place immediately or may be put off until a valid state of the presentation is needed (eg. until the window is exposed).

The viewing constraint between a presentation and its subject determines (1) how the state of the subject is reflected in the presentation and (2) how input received by the node affects the subject. For example, if the subject is a city object in an energy consumption database and the presentation is a point in a scatterplot, the viewing constraint is responsible for supplying the presentation with, for example, population as the x coordinate, altitude as the y coordinate and average particulate ppm as a color variable. When the user selects the point with the mouse, the viewing constraint is responsible for performing the appropriate action on the subject, such as producing an editor window that lets the user inspect and possibly alter the slots and values for that particular city.

In a simple case, the presentation and subject share a *display style* object. The display style has parameters like color, size, orientation, etc. The presentation takes its appearance directly from the display style. Whenever the subject changes its display style, the presentation is automatically notified to redraw itself.

Support for the viewing constraint makes it easy to implement and generalize brushing scatterplots[20,19,3,32]. Earlier versions of brushing were based on a special plot that contained several scatterplots, each showing different variables. The basic design could not be easily extended for use in a window system where arbitrary scatterplots might be visible at any time, or to other kinds of plots besides simple scatterplots.

In Arizona, brushing is implemented in the following way: as the cursor (or brush) moves over a point in a scatterplot, the presentation is "painted" with the display style that was loaded on the brush. The constraint system causes the display style of the subject (a record in the database) to be updated automatically which in turn causes the display styles of all other presentations of that subject to be updated. A consequence of this design is that all exposed plots are automatically involved in painting. No plot needs to know what other plots are on the screen.

Extensions to other types of plots are reasonably straightforward.

3.2.2 Layout constraints

Plot layout is a more open-ended and difficult constraint problem. The idea is to provide the data analyst with a language for making and enforcing assertions about the relative sizes, shapes, or positions of the components of a plot.

A typical example—conceptually trivial but difficult to program—is centering labels around the sides of a scatterplot. The source of programming difficulty is conflicting coordinate systems. The center of the data region is naturally expressed in data coordinates. Heights and widths of label strings can usually only be determined in pixels, for a given font. The mapping of the data region into pixels cannot be determined until we know how much room is left by the labels, but we can't position the labels, choose a font, and determine the label widths and heights until we know where the data region is in device coordinates.

What we will need to support layout constraints is:

- a specification language.
- internal representation.

- general purpose satisfier.
- hooks for user supplied satisfier code.
- fast specialized satisfiers that respond to common perturbations from a solution.
- effective ways of identifying and reporting under/over constrained problems.

References

[1] H. Abelson, G. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.

[2] R.A. Becker, J.M. Chambers, and A.R. Wilks. *The New S Language.* Wadsworth and Brooks/Cole, Pacific Grove, CA, 1988.

[3] R.A. Becker and W.S. Cleveland. *Painting a Scatterplot Matrix: High-Interaction Graphical Methods for Analyzing Multidimensional Data.* Technical Report, AT&T Bell Laboratories , 1984.

[4] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S. Keene, G. Kiczales, and D.A. Moon. *Common Lisp Object System Specification.* ANSI, 1987. under preparation.

[5] D.G. Bobrow and M. Stefik. *The LOOPS Manual.* Xerox PARC, 3333 Coyote Hill Road, Palo Alto, Ca. 94304, 1983.

[6] A.H. Borning. The programming language aspects of thinglab. *ACM TOPLAS*, 3(4):353-387, 1981.

[7] B.J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Mass., 1986.

[8] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. *LINPACK Users' Guide.* SIAM, Philiadelphia, 1979.

[9] Ciccarelli E.C. *Presentation Based User Interfaces.* Technical Report 794, MIT AI Lab, 1984.

[10] A. Goldberg and D. Robson. *Smalltalk-80, The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983b.

[11] G.H. Golub and C.F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, 1983.

[12] C. Hurley. *The data viewer: a program for graphical data analysis.* PhD thesis, Dept. of Statistics, U. of Washington, 1987.

[13] Intellicorp. *Intellicorp Common Windows Manual.* 1975 El Camino Real West, Mountain View, Ca 94040-2216, 1986.

[14] G.E. Kopec. The signal representation language SRL. *IEEE Trans. Acoustics, Speech, and Signal Processing.* ASSP-33(4):921-932, 1985.

[15] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM TOMS.* 5(3):308-371, 1979.

[16] Wm Leler. *Constraint Programming Languages.* Addison-Wesley, Reading, MA, 1988.

[17] D. Levitt. Machine tongues X: constraint languages. *Computer Music Journal.* 8:9-21, 1984.

[18] J.A. McDonald. Antelope: data analysis with object-oriented programming and constraints. In *Proc. of the 1986 Joint Statistical Meetings, Stat. Comp. Sect.*, 1986.

[19] J.A. McDonald. Exploring data with the Orion I workstation. 1982. A 25 minute, 16mm sound film, which demonstrates programs described in McDonald (1982) It is available for loan from: Jerome H. Friedman, Computation Research Group, Bin # 88, SLAC, P.O. Box 4349, Stanford, California 94305.

[20] J.A. McDonald. *Interactive Graphics for Data Analysis.* Technical Report Orion 11, Dept. of Statistics Stanford, 1982. Ph.D. thesis, Dept. of Statistics, Stanford.

[21] J.A. McDonald. *Object-oriented design in numerical linear algebra.* Technical Report 109, Dept. of Statistics, U. of Washington, 1987.

[22] J.A. McDonald and J.O. Pedersen. Computing environments for data analysis I: introduction. *SISSC*, 6(4):1004-1012, 1985.

[23] J.A. McDonald and J.O. Pedersen. Computing environments for data analysis II: hardware. *SISSC*, 6(4):1013-1021, 1985.

[24] J.A. McDonald and J.O. Pedersen. Computing environments for data analysis III: programming environments. *SISSC*, 9(2):380-400, 1988.

[25] N. Meyrowitz, editor. *OOPSLA '86 Proc.*, ACM, November 1986. SIGPLAN Notices 21 (11).

[26] N. Meyrowitz, editor. *OOPSLA '87 Proc.*, ACM, December 1987. SIGPLAN Notices 22 (12).

[27] S. Mittal, D.G. Bobrow, and K.M. Kahn. Virtual copies: at the boundary between classes and instances. In *OOPSLA '86 Proc.*, 1986.

[28] R.B. Rao. *Towards interoperability and extensibility in window environments via object-oriented programming.* Master's thesis, MIT EECS, 1987.

[29] R.W. Scheifler and J. Gettys. The X window system. *ACM TOG*, 5(2):79–109, 1986.

[30] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler. *Matrix Eigensystem Routines—EISPACK Guide, 2nd Edition.* Springer-Verlag, Berlin, 1976.

[31] G.L. Steele. *Common Lisp, The Language.* Digital Press, 1984.

[32] W. Stuetzle. Plot windows. *JASA*, 82(398):466–475, 1987.

[33] Sun Microsystems, Inc. *NeWS Manual.* Sun Microsystems, Inc., 2550 Garcia Ave, Mountain View, Ca. 94043, 1987. Part No. 800-1632-10.

[34] G.J. Sussman and Jr. Steele, G.L. Constraints—a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[35] S. Thatte. Persistant memory: merging AI knowledge and databases. *TI Engineering J.*, 3(1), Jan-Feb 1986.

[36] J. Walker, D.A. Moon, D.L. Weinreb, and M. Macmahon. The Symbolics Genera programming environment. *IEEE Software*, 4(6):36–45, 1987.