Using APL2 to Create an Object-Oriented Environment for Statistical Computation

Michael FRIENDLY* and John FOX†

Since its introduction, APL has frequently been touted as an ideal programming language for statistical applications. Among the attractive features of APL for statistics are its extensibility, the presence of primitives for operations such as sorting, matrix inversion, and arranging data, and powerful facilities for handling matrices and other arrays. Newer programming languages incorporating some of the features of APL—notably New S and XLisp-Stat—have been designed specifically for statistical applications. Is there still a niche for APL?

We believe that APL2 continues to offer some significant benefits for statistical computation, including user-defined operators, nested arrays, and convenient implementation of arrays of any dimension. We use these characteristics of the language in the course of designing an extensible computing environment for data analysis and programming based on APL2 that incorporates some of the features of modern statistical programming languages, such as data objects, symbolic model specification, missing-data handling, and automatic search of a path of files. The system, which includes interactive statistical graphics, general linear models, and robust estimation methods, has been implemented using IBM's APL2 for PC-compatibles—both the standard version of this system and the freeware TryAPL2. The latter provides students with a free environment for modern data analysis and one in which they can explore the design of statistical software.

Key Words: Bootstrapping; Data objects; Graphics; Missing data; Nested arrays; Robust regression; Statistical operators.

1. INTRODUCTION

The initials APL stand for *A Programming Language,* the title of a book written in 1962 by Kenneth Iverson, describing a comprehensive, simple, and consistent mathematical notation suitable for computer translation. By the late 1960s an interpreter and interactive computer system were created for a version of APL (Falkoff and Iverson 1968). Soon after its appearance, and occasionally in the interim, APL has been touted as an ideal programming language and environment for statistical computation (see Anscombe 1981): The language has powerful primitives for manipulating data arrays; it is naturally

*Associate Professor, Psychology Department, York University, Toronto, Ontario, Canada M3J 1P3
†Professor, Department of Mathematics and Statistics and Department of Sociology, York University, Toronto, Ontario Canada M3J 1P3

©*1994 American Statistical Association, Institute of Mathematical Statistics, and Interface Foundation of North America*
Journal of Computational and Graphical Statistics, Volume 3, Number 4, Pages 387–407

suited to interactive computation; and it is extensible. The second generation of APL, called APL2, has been available for nearly a decade. APL2 adds a variety of significant capabilities to the language, the most important of which are generalized arrays—arrays whose elements are themselves arrays, which may be nested to arbitrary depth—and user-defined operators—programs whose arguments are functions.

APL2 is arguably the most powerful language yet developed for expressing statistical computation. One's ability to get work done, however, depends as much on the programming environment as on the primitives of the language. Statistical developments in APL have largely taken the form of collections of APL functions, sometimes quite extensive, for statistical computations (Anscombe 1981; Friendly 1991; Heiberger 1989; Polhemus 1983; Thompson 1992). Newer models of programming environments designed specifically for statistical computation, graphics, and data analysis such as New S (Becker, Chambers, and Wilks 1988) and XLisp-Stat (Tierney 1990) have added significant concepts to the vocabulary of computational and graphical statistics, such as dynamic, interactive graphics (e.g., brushing and linked data displays) and object-oriented computing.

We describe here an environment for data analysis and programming implemented in APL2 and incorporating some of the features of these statistical programming languages, such as data objects, symbolic model specification, missing-data handling, and automatic search of a path of files. To give a feeling for the capabilities of the language, we describe some features of APL2 that offer significant benefits for statistical computation, including user-defined operators, nested arrays, and convenient computation on arrays of any dimension. We do not claim any overall superiority of APL2 as a language or APL2STAT as a statistical programming environment. Our goal instead is simply to describe some features of both that should be of interest to designers of systems for statistical computing and to those who may have used APL in the past. It is inappropriate for us to undertake a substantial tutorial on APL within the confines of this article. There are several good books on APL programming; we particularly recommend Gilman and Rose (1984) and Brown, Pakin, and Polivka (1988). An extended version of this article (Friendly and Fox 1993) that contains more tutorial examples is available from the authors.

Nevertheless, we wish to communicate the essential characteristics of APL2 as a computer language and to illustrate its use in programming statistical applications. Section 2 describes several features of APL2 that have been particularly useful in the development of APL2STAT. Section 3 provides a sample APL2STAT session to give the flavor of working in this environment. The basis for a simple object-oriented design for data and statistical methods in APL2STAT is described in Sections 4 and 5. Section 6 illustrates how interaction with the APL interpreter can be enhanced.

2. OVERVIEW OF APL2 AND APL2STAT

APL is typically implemented in a comprehensive system or environment that includes facilities for interacting with an APL interpreter and for writing, editing, and debugging APL programs. IBM's APL2/PC product, for example, provides a debugging

```
      ⍝-- USING NESTED ARRAYS FOR PARTITIONED MATRICES
      Y←7 10 12 6                          ⍝ create vector Y

      ⎕←X←10×⍳4                            ⍝ create vector X
10 20 30 40

      ⎕←YX←Y,1,(4 1⍴X)                     ⍝ matrix with Y,1,X
 7 1 10
10 1 20
12 1 30
 6 1 40

      Z←⊃¨(1 1 1 1)⊂[1](1 2 2)⊂[2]YX      ⍝ partition columns

      DISPLAY Z                            ⍝ show nested structure
┌→1 2────────────┐
↓ ┌→4┐  ┌→4 2┐   │
│ ↓ 7│  ↓1 10│   │
│ │10│  │1 20│   │
│ │12│  │1 30│   │
│ │ 6│  │1 40│   │
│ └~─┘  └~───┘   │
└∊───────────────┘

      DISPLAY ⍉⍉¨Z                         ⍝ transpose
┌→2 1──────────────┐
↓ ┌→1 4────────┐   │
│ ↓7 10 12 6│   │
│ └~─────────┘     │
│ ┌→2 4────────┐   │
│ ↓ 1  1  1  1│   │
│ │10 20 30 40│   │
│ └~──────────┘    │
└∊─────────────────┘

      DISPLAY (⍉⍉¨Z) TIMES Z               ⍝ multiply nested arra
┌→2 2──────────────────┐
↓ ┌→1 ┐  ┌→1 2───┐     │
│ ↓329│  ↓35 870│     │
│ └~──┘  └~─────┘     │
│ ┌→2 ┐  ┌→2 2─────┐   │
│ ↓ 35│  ↓  4  100│   │
│ │870│  │100 3000│   │
│ └~──┘  └~───────┘   │
└∊─────────────────────┘
```

*Figure 1. Using Nested Arrays for Partitioned Matrices. The primitive partition function (⊂) creates a partitioned matrix, splitting along rows (⊂[1]) or columns (⊂[2]); the left argument to partition specifies the rows and columns in each partition. The **DISPLAY** function shows the type, shape, and structure of each level of nesting. Numbers on the top line of each box give the dimensions of the object contained. The **TIMES** function multiplies simple or nested arrays.*

editor that automatically loads an offending function, with the cursor at the point of error, and allows single-stepping and tracing with convenient access to all defined variables. Modern APL environments include a "session manager" that maintains a log of the APL session, allows the user to edit on-screen APL expressions, and provides access to system facilities such as printers and files. Facilities for downloading APL fonts to printers and video displays, for reading and writing native (ASCII) files, and communicating with other operating-system services such as windowing systems and pointing devices are also provided. While the characteristics of the APL language are admirably standardized, these auxiliary characteristics are often implementation-specific and we do not describe them further. Here we focus on three features of APL2 that have been particularly useful in developing APL2STAT. These are nested arrays, iteration with data and with an "each" operator, and user-defined operators.

2.1 NESTED ARRAYS

In the arrays of most computer languages, each element is a scalar—a single number or character. APL2 also provides nested arrays, in which each element can be a scalar or another (nested) array of any rank (number of dimensions), shape, depth, and type (numeric, character, or mixed). Hence, APL2 nested arrays are general, recursive, tree-like data structures. This arbitrary embedding of one array inside another gives APL2 arrays similar expressive power to the lists of Lisp. APL2 arrays are, however, more powerful (or at least more convenient) than lists in Lisp, because (a) APL2 arrays themselves implicitly contain the information about the structure (rank, shape, depth) of the elements at any level, and (b) APL2 contains a rich set of primitive functions for constructing data structures, selecting and assigning substructures, and applying functions to the data at the nodes at particular levels of a data structure.

For statistical computation, nested arrays are particularly useful for the following kinds of operations (among others): (a) representing and operating on partitioned matrices; (b) parsing expressions entered as character strings; (c) generalizing univariate computations to multivariate ones; and (d) constructing data and analysis objects. For example, Figure 1 illustrates the use of nested arrays to compute the matrix expression $\mathbf{Z}'\mathbf{Z}$, where $\mathbf{Z}$ is the partitioned matrix $[y \,|\, \mathbf{X}]$ of a linear model. (The ι function generates an integer sequence; ρ reshapes its right argument to an array of shape given by its left argument, and the **,** function joins arrays. The APL primitive for transpose is ⍉ for a simple array. For a partitioned matrix, transpose each submatrix and transpose the result, which is ⍉⍉¨ in APL2.)

2.2 ITERATION WITH DATA AND EACH

APL2 is unlike most other programming languages in that there are minimal facilities for "control structure": there are no built-in operations like "do loops" for iteration. The only primitive control structure in APL is the "branch arrow," →, which operates as a go-to construct in defined functions. Although it is possible to write user-defined functions and operators to implement repetitive control structures such as "do," "while," and "until" loops (Eusebi 1985), this is usually not done.

Instead, APL2 supplies implicit iteration over data structures in several ways that usually make explicit iteration unnecessary. These array operations are entirely data driven: computation is performed over an array and the data themselves control the limits of operation. First, all scalar functions extend to arrays, producing element-wise results of the same shape automatically (see the `NORMAL_DENSITY` function in Figure 4, p. 393). Second, many APL2 functions and operators can be applied over specified axes of any array. For example, the reduction operator, `/`, iterates any primitive or defined function over one or more axes of an array. Thus, `+/[1]M` sums a matrix `M` over its first axis, rows. Similarly, the expression, `M÷[1]+/[2]M`, divides each row by the row total, and `+\ M÷[1]+/[2]M` computes cumulative row proportions, an operation that would require two or more loops in conventional programming languages.

In APL2, the each operator (¨) provides iteration over the items in simple or nested arrays. Again the limits of iteration are implicit in the data. A problem can often be

```
    ∇
  [0]  yhat←lowess_fit xi;localx;localy;near;w;ΔWEIGHTS
  [1]  ⍝Return lowess fitted value at x = xi
  [2]   uses¨'tricube' 'wls'
  [3]   near←r↑⍋|x-xi          ⍝ find indices of r x-values
  [4]   localx←x[near]         ⍝ .. closest to xi
  [5]   localy←y[near]
  [6]   w←|xi-¯1↑localx
  [7]   ΔWEIGHTS←ΔDELTA[near]×tricube(localx-xi)÷w
  [8]   yhat←(1,xi)+.×localy wls 1,[1.5]localx     ⍝ get fit at xi
    ∇
    ∇
  [0]   z←parm LOWESS yx;e;fraction;it;iterations;n;r;x;y;yhat
  [1]  ⍝ LOcally WEighted Scatterplot Smoother.
  [2]  ⍝.L (parm) 2-vector giving the smoothing fraction and no. of
  [3]  ⍝.L         iterations or a scalar giving the smoothing fraction.
  [4]  ⍝.L         Defaults: fraction=.5, 3 iterations.
  [5]  ⍝.R <yx>    A 2-column matrix of y-values and x-values, or a
  [6]  ⍝.R         quoted expression evaluating to a matrix.
  [7]  ⍝.Z <z>     A 2-col. matrix of smoothed y- and sorted x-values.
  [8]  ⍝.O ΔDELTA  Final robustness weights.
  [9]   uses¨'select' 'get_data' 'lowess_fit' 'bisquare' 'MEDIAN' 'COL'
 [10]   'parm' default 0.5 3
 [11]   (fraction iterations)←2↑,parm,3
 [12]   yx←get_data yx          ⍝ handle matrix or string expression
 [13]   yx←select yx            ⍝ filter missing & omitted data
 [14]   n←⍋x←yx[;2]             ⍝ extract x, get indices to sort
 [15]   y←yx[n;1]               ⍝ extract y, sorted by x values
 [16]   x←x[n]                  ⍝ sort x
 [17]   ΔDELTA←(n←⍴x)⍴1
 [18]   r←⌊0.5+n×fraction
 [19]   it←0
 [20]  iterate:→(iterations<it←it+1)/return
 [21]   yhat←lowess_fit¨x                    ⍝ fit at each x
 [22]   e←y-yhat
 [23]   ΔDELTA←bisquare e÷6×MEDIAN|e         ⍝ new robustness weights
 [24]   →iterate
 [25]  return:z←yhat,COL x
    ∇
```

Figure 2. Functions for Locally Weighted Smoothing. The **LOWESS** *function uses the each operator to find the lowess fit at each abscissa.*

solved for a single case, and then generalized to work over a range of cases using each. For example, the function `lowess_fit` in Figure 2 finds the robust, locally weighted smoothed value for a single abscissa in a bivariate scatterplot. To find the lowess fit at several abscissas, it is then required only to `lowess_fit¨` *X*, as shown in line `[21]` of the *LOWESS* function.

Not all iteration can be subsumed by array operations and operators like each. The *LOWESS* function calculates weights for each observation to produce a robust smoothed curve. When the data (weights) change on each iteration, as they do in lines `[20-24]` of the *LOWESS* function, iteration usually must be programmed explicitly.

2.3 OPERATORS

APL2 generalizes the APL concept of *operators* (e.g., reduce and each, explained previously), which extend or alter the operation of functions. For example, all primitive scalar functions `f`, such as + and ×, can have the reduction operator `f/` applied to produce *derived functions* `+/`, `×/`, and so forth. The derived reduction functions then apply the

```
      ⍝-- REDUCTION OPERATIONS
      +/7 5 3 11                ⍝ sum reduction
26
      2+/7 5 3 11               ⍝ running pairwise sums
12 8 14
      ×/7 5 3 11                ⍝ times reduction
1155
      ⌊/7 5 3 11                ⍝ minimum
3

      ⍝-- SCAN OPERATIONS
      +\7 5 3 11                ⍝ cumulative sum
7 12 15 26
      ×\7 5 3 11                ⍝ cumulative product
7 35 105 1155
      ⌊\7 5 3 11                ⍝ sequential minima
7 5 3 3

      ⍝-- GENERALIZED OUTER PRODUCTS
      1 2 3∘.*1 2 3             ⍝ ⍳3 raised to each power
1 1  1
2 4  8
3 9 27
      1 2 3∘.≥1 2 3             ⍝ lower triangular matrix
1 0 0
1 1 0
1 1 1
      V∘.!V←0 1 2 3 4           ⍝ Pascal's triangle to order 4
1 1 1 1 1
0 1 2 3 4
0 0 1 3 6
0 0 0 1 4
0 0 0 0 1

      DF←2*⍳3                                  ⍝ degrees of freedom
      P←.95 .975 .99 .995                      ⍝ probabilities
      (' ',P),[1]DF,(DF∘.CHISQUARE_QUANT P)    ⍝ make a table
     0.95   0.975   0.99   0.995
2   5.9915 7.3778 9.2103 10.597
4   9.4779 11.132 13.265 14.848
8   15.505 17.532 20.087 21.952
```

Figure 3. Some Expressions Using APL2 Primitive Operators. The reduce operator (`f/`) applies any function `f` across one or more axes of an array. Scan (`f\`) produces cumulative functions, and outer product, `∘.f` applies a function `f` between pairs of items from its left and right arguments in all combinations. These operators work in the same way with user-defined functions, as shown by the use of outer product with the `CHISQUARE_QUANT` function to generate a table of χ^2 quantiles.

function argument to the operator between items of its data argument and give, for $+$ and $\times$, the sum and product, respectively, of the data values over the last (or specified) coordinate.

Sum- and product-reduction operators are so generally useful that they have been implemented in New S, XLisp-Stat, and various matrix languages such as SAS/IML and Gauss. Although other languages permit functions as arguments to functions, APL2 operators are particularly powerful, in that (a) they apply to all primitive functions of a particular class (such as scalar functions), (b) they apply equally to user-defined functions, and (c) the operators themselves may be user-defined.

Other APL2 operators include *scan*, $f\backslash$, and *generalized outer product*, $\circ.f$, some applications of which are illustrated in Figure 3. The derived functions of scan apply the function $f/$ to each of the first i items of its right argument for $i = 1, 2, \ldots,$ giving cumulative sums and products for $+\backslash$ and $\times\backslash$. Outer product, $\circ.f$, generalizes

```
    ⍝-- NUMERICAL INTEGRATION BY SIMPSON'S RULE
    ∇
[0]   Z←ERROR(F INTEGRATE)RANGE;AR
[1]  ⍝Recursive integration operator
[2]   'ERROR' default 0.00001
[3]   →(ERROR<|(2(F simpson)RANGE)-Z←4(F simpson)RANGE)/RECUR
[4]   →0
[5]  RECUR:AR←RANGE,0.5×+/RANGE
[6]   ERROR←ERROR÷2
[7]   Z←(ERROR(F INTEGRATE)AR[1 3])+ERROR(F INTEGRATE)AR[3 2]
    ∇

    ∇
[0]   R←N(F simpson)RANGE;T;⎕IO
[1]  ⍝Simpsons rule operator
[2]   ⎕IO←1
[3]   T←(--/RANGE)÷N←N+2|N
[4]   R←(T÷3)×((1,(N-1)⍴4 2),1)+.×F RANGE[1]+T×0,⍳N
    ∇

    ⍝-- Integrate exponential function from 0 to 1
    1E¯8 * INTEGRATE 0 1
1.718282

    ∇
[0]   Z←NORMAL_DENSITY X
[1]  ⍝Calculate standard normal density function
[2]   Z←(*-0.5×X*2)÷(○2)*0.5
    ∇

    NORMAL_DENSITY 0 1 2
0.3989423 0.2419707 0.05399097

    ⍝-- Integrate the normal density
    NORMAL_DENSITY INTEGRATE ¯1 1
0.6826896
    NORMAL_DENSITY INTEGRATE 0 10
0.5000004
    1E¯8 NORMAL_DENSITY INTEGRATE 0 10
0.5
```

Figure 4. Defined Operators for Numerical Integration of a Scalar Monadic Function. The left operand to ***INTEGRATE*** *specifies the function, which can be an APL primitive or user-defined function of a single argument. The first example integrates the exponential function, ⋆, over the range (0,1). The remaining examples find areas under the standard normal curve by integrating the* ***NORMAL_DENSITY*** *function.*

construction of a multiplication table, applying the function *f* between pairs of items from its left and right arguments in all combinations.

User-defined operators add considerably to the expressive power of APL2 for statistical computing, because they provide the basis for algorithms that apply automatically to an entire class of functions. A simple operator for numerical integration of an arbitrary function is illustrated in Figure 4. The *INTEGRATE* function (after Sykes and Hawkes 1989) also illustrates the use of tree-recursion in APL functions and operators. Simpson's rule is used on the original interval with 2 and 4 subintervals and the operator stops if the difference between these two approximations is less than the accuracy specified. Otherwise, *INTEGRATE* is applied recursively to both halves of the original interval.

Similarly, one can define an operator for bootstrap resampling of an arbitrary statistical estimator. The *BOOTSTRAP* operator, shown in Figure 5, applies to any monadic (one-argument) function that takes a vector or matrix as a right argument and returns an estimated quantity (which may be an array) as an explicit result. Figure 6 illustrates

```
      ∇
[0]   r←reps(fn BOOTSTRAP)data;n;bootstrap
[1]   ⍝ Operator for computing bootstrapped sample estimates.
[2]   ⍝.L (reps) Number of bootstrap replications. Default: 100
[3]   ⍝.L <fn> A function that takes a data matrix as a right
[4]   ⍝.L      argument and returns a vector or estimates.
[5]   ⍝.R <data> Input data matrix or vector.  Rows of the data
[6]   ⍝.R         matrix are sampled repeatedly with replacement.
[7]   ⍝.Z <r> A matrix containing (by rows) the result of fn for
[8]   ⍝.Z      each bootstrap replication.
[9]    r←⎕FX 'r←(fn bootstrap)r' 'r←,fn data[?n⍴n;]'
[10]   uses¨'default' 'select' 'get_data' 'null' 'COL'
[11]   'reps' default 100
[12]   n←1↑⍴data←COL select get_data data
[13]   ('Result for full sample: ')(fn data)
[14]   'Beginning BOOTSTRAP replications '
[15]   r←fn bootstrap¨⍳reps
[16]   'BOOTSTRAP replications completed.'
[17]   r←⊃[2]r
    ∇
```

Figure 5. An APL2 Operator for Bootstrap Sampling. The essential work is done by the **bootstrap** *sub-operator, defined here as a local operator in line* [9]. *The expression* **data**[?n⍴n;] *generates a random sample with replacement of the integers from 1 to n, which select the corresponding rows of* **data** *to pass to the function* **fn**. *The iteration required for* **reps** *replications is carried out in line* [15] *of* **BOOTSTRAP**.

```
      ⍝ Define a function to return mean and variance
    ∇
[0]   R←STATS X
[1]   R←(MEAN X),VARIANCE X
    ∇

      DATA←* NORMAL_RAND 50          ⍝ 50 lognormal values

      R←STATS BOOTSTRAP DATA         ⍝ bootstrap means and variances
 Result for full sample:    1.5357 3.5881
Beginning BOOTSTRAP replications
BOOTSTRAP replications completed.

      ('MEAN' 'VAR'),[1]5↑[1]R       ⍝ first 5 bootstrap replications
 MEAN   VAR
 1.5824 3.8957
 1.4367 2.6835
 2.1826 7.8176
 1.9824 5.5228
  1.174 1.1709

      DESCRIBE R                     ⍝ summarize
                        Col_1  Col_2
 Mean                  1.5651 3.7373
 Standard deviation 0.25711 1.3982
 Minimum                1.126 1.1541
 Lower hinge (Q1)       1.381  2.651
 Median                1.5329  3.652
 Upper hinge (Q3)      1.7305 4.6877
 Maximum               2.3215 7.8176
 N (selected, ≠'.')       100    100
```

Figure 6. An APL2 Operator for Bootstrap Sampling. The **BOOTSTRAP** *operator is applied to the mean and variance of a log-normal sample.*

the computation of bootstrap means and variances of a random lognormal sample. Other APL2STAT operators include *MD* (see Subsection 4.2), which extends all scalar functions to handle missing data, *QUANTILE_PLOT*, for Q-Q plots of any distribution function, a *TIMER* operator to time the execution of any function, and *TABLE* (Subsection 4.4) which calculates multiway tables of any (array-valued) statistic.

Finally, we simply mention two related APL programming features that enhance the capabilities of functions and operators: (a) The primitive execute function (⍎) takes a character-string argument and executes that string as if typed to the interpreter. The character-string argument, of course, can be constructed by other functions and operators under program control. (b) APL functions themselves can be turned into character arrays containing their definitions, and character arrays can be defined as functions under program control. Functions so defined can be made local to the function that creates them (see Figure 5), in which case they disappear when that function completes. These features allow the use of unnamed function-expressions similar to the lambda-expressions of Lisp (Benkard 1990) and pure functions of Mathematica (see Figure 13, p. 403).

3. APL2STAT

APL2STAT is an integrated set of over 300 APL2 programs for statistical analysis, with an emphasis on statistical graphics. The programs use a simple object-oriented system and employ common procedures for accessing data and storing results. Because APL2STAT is not a statistical package, but rather is built upon the APL2 programming language and environment, it is simple to modify and supplement.

In its current form, APL2STAT includes, in addition to standard statistical summaries, functions for least-squares linear regression and linear models, for robust linear models using iteratively reweighted least squares, for regression with autocorrelated errors, and for categorical data analysis using loglinear models and dichotomous and polytomous logit models. Extensive graphics capabilities include a variety of scatterplots, boxplots, scatterplot matrices, partial regression and residual plots, regression influence plots, Box–Cox and Box–Tidwell transformation constructed-variable plots, lowess scatterplot smoothing, robust multivariate outlier plots, and kernel-density estimation. Whenever sensible, points may be interactively identified, deleted, moved, and/or highlighted using a pointing device (such as a mouse) or the cursor keys.

The APL2/PC implementation provides high-resolution (VGA, 800x640) full-screen graphics or splitscreen graphics with a session-manager text portion at the bottom, but only a single graphics window may be active at any time. All APL2STAT graphics may be captured to graphics objects, replayed, or printed to a Postscript-capable device.

All APL2STAT functions are self-documenting: each function contains comment lines describing the purpose, syntax, and result (see the *LOWESS* function in Figure 2, p. 391, or the *BOOTSTRAP* operator in Figure 5. One function, *HOW*, extracts, formats, and displays the description of any function, and another function, *APROPOS*, searches the active workspace and all APL2STAT libraries for the names of functions and operators that contain a given string.

3.1 SAMPLE APL2STAT SESSION

The illustrative session shown in Figure 7 is meant to convey the flavor of using the APL2STAT system; the illustration is necessarily abbreviated, however. When the APL2STAT *SHELL* workspace is loaded, the *SHELL* function (see Section 6) is automatically invoked. Subsequent lines are entered under the control of the *SHELL*, as evidenced by the statement-number prompt.

- In line 1, for example, the *PRINT* function and the *DUNCAN* data set object (Subsection 4.1) are automatically copied into the active workspace from files located on

```
      )LOAD SHELL
SAVED 1993-05-21 10.55.52 SHELL

APL2STAT (SHELL)  [Version 2.0] 1993  John Fox & Michael Friendly
Enter  )QUIT  to exit

1>  PRINT 'DUNCAN'
                                     INCOME EDUCATION PRESTIGE
 accountant_for_a_large_business      62        86        82
 airline_pilot                        72        76        83
 architect                            75        92        90
 ...                                 ...       ...       ...
 janitor                               7        20         8
 policeman                            34        47        41
 restaurant_waiter                     8        32        10

2>  USE 'DUNCAN'
 Variables created: INCOME EDUCATION PRESTIGE

3>  USING
 Current dataset:  DUNCAN
 Contains   45   observations
 45  observations selected
 45  observations with valid data

4>  PAIRS 'PRESTIGE, INCOME AND EDUCATION'   ⍝ SCATTERPLOT MATRIX

5>  LINEAR_MODEL 'PRESTIGE←INCOME+EDUCATION'

6>  ⎕PP←5    ⍝ show 5 significant digits

7>  PRINT 'LAST_LM'
 GENERAL LINEAR MODEL: PRESTIGE←INCOME+EDUCATION

                  Coefficient   Std.Error    t         p
  CONSTANT          ¯6.0647       4.2719 ¯1.4197  0.16309
  INCOME            0.59873      0.11967  5.0033  0.00001061
  EDUCATION         0.54583     0.098253  5.5554  1.7452E¯6

                  DF    SS     F      p
  Regression       2  36181 101.22 4.6728e¯11
  Residuals       42 7506.7                 0
  Total           44  43688

  R-SQUARE = 0.82817   SE = 13.369   N = 45

   Source         SS      DF     F       p
   CONSTANT     360.22   1 42   2.0154   0.16309
   INCOME       4474.2   1 42   25.033   0.00001061
   EDUCATION    5516.1   1 42   30.863   1.7452E¯6

8>  INFLUENCE_PLOT 'LAST_LM'   ⍝ RSTUDENTS, HATVALUES, Cook'S D'S

9>  GRAPHICS 'OFF'             ⍝ exit split-screen graphics mode

10>  BIGGEST GET 'LAST_LM' 'COOKS D'
 minister                          0.56638
 railroad_conductor                0.22364
 reporter_on_a_daily_newspaper     0.098985
 railroad_engineer                 0.080968
 building_contractor               0.058524
```

Figure 7. Sample APL2STAT Session.

the search path (Section 6). The generic *PRINT* function employs a print method (Section 5) appropriate for a data set object to display the Duncan data set, labeling the output with observation and variable names. (In the interest of brevity, most of the output is excluded.)

- The *USE* function makes the Duncan data set current, defining appropriately named variables as vectors in the active workspace, and performing other tasks such as defining observation names and setting the data-selection vector (Subsection 4.3). *USING* reports the current data set.
- The *PAIRS* function creates the scatterplot matrix shown in Figure 8. The "highlighted" observations and their names were identified interactively using a mouse.
- The *LINEAR_MODEL* function returns an object, named *LAST_LM* by default, produced by regressing prestige on income and education. Note that the right argument to *LINEAR_MODEL* is a symbolic model formula, potentially specifying interactions, variable transformations, nesting, et cetera. If either independent variable

```
11>  OMIT¨ 'minister' 'railroad_conductor'

12>  'NEW_FIT' LINEAR_MODEL 'PRESTIGE←INCOME+EDUCATION'

13>  PRINT 'NEW_FIT'
 GENERAL LINEAR MODEL: PRESTIGE←INCOME+EDUCATION

                  Coefficient   Std.Error    t          p
  CONSTANT           ¯6.409        3.6526 ¯1.7546  0.086992
  INCOME              0.8674      0.12198  7.1113  1.3428E¯8
  EDUCATION          0.33224      0.09875  3.3645  0.0017048

                 DF    SS      F       p
  Regression      2   36815 141.26 1.4015e¯10
  Residuals      40  5212.6                  0
  Total          42   42028

  R-SQUARE = 0.87597   SE = 11.416   N = 43

    Source          SS      DF      F         p
   CONSTANT       401.2    1 40   3.0787   0.086992
   INCOME          6590    1 40   50.57    1.3428E¯8
   EDUCATION     1475.1    1 40   11.32    0.0017048

14>  HOW 'HYPOTHESIS'
Purpose:
    Tests the linear hypothesis given by <h> for <model>
    Chooses an appropriate method for the type of <model>

Usage:
    r←model HYPOTHESIS h
    <model> is the name of a model object, enclosed in ' '
    <h> is a vector or matrix specifying a linear hypothesis

Examples:
    'MODEL1' HYPOTHESIS 2 4 ρ 0 1 0 0  0 0 1 0
    ⍝ MODEL1 has 4 coefficients

15>  'NEW_FIT' HYPOTHESIS 0 1 ¯1
 SS = 848.68   F[  1 40  ] =  6.5126    p =  0.014646

16>  'LAST_LM' HYPOTHESIS 0 1 ¯1
 SS = 12.195   F[  1 42  ] =  0.068233    p =  0.7952

17>  )QUIT
```

Figure 7. Continued.

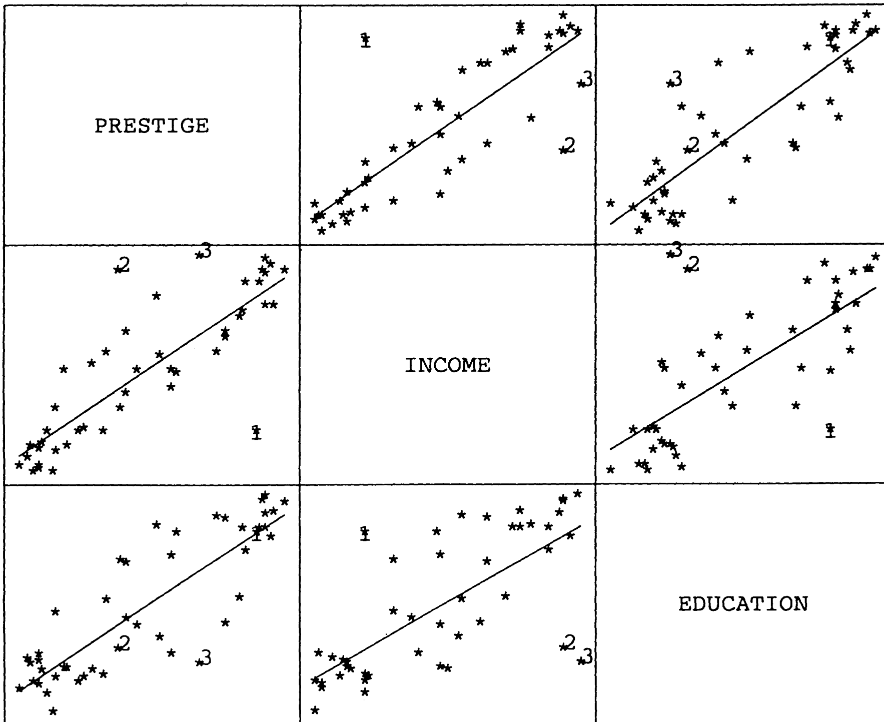

*Figure 8. Scatterplot Matrix Produced by the **PAIRS** Function. Clicking on a point causes the observation to be labeled in each panel. 1 = minister; 2 = railroad_conductor; and 3 = railroad_engineer. In the original, the correspondence between names and flagged points appears in the display.*

had been qualitative (i.e., a character variable), then an appropriate set of dummy regressors or contrasts would have been generated automatically. Printing the linear-model object produces a summary report (cf. the result of printing a data set).

- The *INFLUENCE_PLOT* function constructs the graphical display shown in Figure 9. Again, the labeled data points were identified interactively.
- The *BIGGEST* function reports (by default) the five largest Cook's D's. These values are extracted by the object-accessor function *GET* from the linear-model object (Subsection 4.1).
- The *OMIT* function excludes two observations from the subsequent analysis by locating the position of these observations in the data set and setting to zero the corresponding elements of the selection vector (see Subsection 4.3). The linear model saved in the object named *NEW_FIT* is therefore based on the remaining 43 observations.
- The *HOW* function extracts and prints documentation for the *HYPOTHESIS* function, which is then used to test linear hypotheses for the two regression models fit to the Duncan data. *HYPOTHESIS*, like *PRINT*, is a generic function which chooses a method appropriate to the type of the object that it processes.

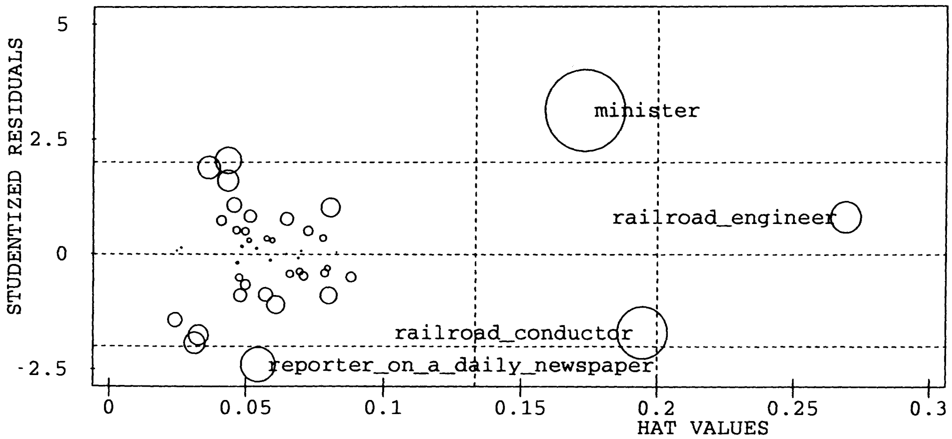

Figure 9. Influence Plot Showing Studentized Residuals, Hatvalues, and Cook's D's for Duncan's Regression. The circles plotted have areas proportional to Cook's D. The labeled observations were identified interactively. The broken horizontal lines are at 0 and ± 2; the vertical lines are at 2× and 3× the average hatvalue.

- The last line (`)QUIT`) exits the the `SHELL` function, returning the user to the APL2 interpreter. This and other APL system commands are simulated under the `SHELL`.

4. APL2STAT DATA

The generalized arrays of APL2 alone are rich enough to handle the problem of both character-valued and numeric-valued variables in a single data object. A convenient statistical computing environment must also provide for (a) meaningful variable names and observation labels; (b) uniform handling and propagation of missing data; and (c) selection of the variables and observations to be used in a particular analysis or plot. This section describes how the representation of DATASET objects in APL2STAT provides for these requirements.

4.1 DATASET OBJECTS

APL2STAT employs a simple object system for organizing data sets and the results of analysis functions and operators, such as the result of fitting a linear model or building a table of frequencies or other statistics. The design of the APL2STAT object system was influenced by Alfonseca's (1989a; 1989b) work on object-oriented programming in APL2. An APL2STAT object is a two-column nested array consisting of slot names and slot values, with one row for each slot. All objects inherit properties (slots and methods) from other objects, with the exception of the root object, `OBJECT_PROTO`.

All objects contain the slots `PARENT` and `TYPE`. The `PARENT` slot gives the name of the immediate ancestor of the object, which is `OBJECT_PROTO` for the data set prototype (and `DATASET_PROTO` for dataset objects created from this prototype). In the present version all objects inherit from a single parent; multiple inheritance would be relatively easy to implement, however, by making the value of the `PARENT` slot a nested vector.

```
        SAMPLE
PARENT          DATASET_PROTO
TYPE            DATASET
VARIABLES        EDUCATION INCOME GENDER STATUS
OBSERVATIONS     FRED VALERIE ANITA MARIA JOHN JIM ABIGAIL JUAN
DATA             12 25 M 1:Low
                 16 .  F 3:Hi
                  9 18 F 2:Med
                 12 21 F 2:Med
                 20 55 M 1:Low
                 12 30 M 1:Low
                 14 45 F 3:Hi
                 .  51 M 3:Hi
CODEBOOK         Sample data set. The variables are:
                  [1] Education (years)
                  [2] Income ($000)
                  [3] Gender ('M'/'F')
                  [4] Status ('Low'/'Med'/'Hi')
```

Figure 10. A Sample DATASET Object. A DATASET object is a 6 × 2 nested matrix. The items in the first column are enclosed (scalar) character names of slots; those in the second column give the slot values.

DATASET objects also contain the following slots: (a) *VARIABLES*—a nested vector of variable names; (b) *OBSERVATIONS*—a nested vector of observation labels; (c) *DATA*—the data matrix; and (optionally) (d) *CODEBOOK*—a character matrix giving descriptive information about the source of data, variable encodings, and so forth. A sample DATASET object is shown in Figure 10. Data set values may be numeric or character scalars (including character strings of arbitrary length enclosed as nested scalars). The '.' values in the *DATA* slot represent missing data, as explained in the following subsection.

By convention, character variables are treated by many functions as categorical (e.g., *LINEAR_MODEL*). Character data beginning with a left bracket, parenthesis, or a numeral (e.g., *'1:Low'* or *'[1]Low'*), are treated as ordered categories where appropriate, the order specified by the implicit sort-order of APL characters.

DATASET objects may be defined directly using APL2STAT functions or by reading data from DOS files. Consider the following illustration, assuming that the data already reside in a three-column matrix *SAMPLE_DATA*; that observation names are contained in the (nested) vector *NAMES*; and that a codebook for the dataset is contained in the character matrix *SAMPLE_CB*.

```
MAKE 'DATASET_PROTO' 'SAMPLE'
PUT 'SAMPLE' 'DATA' SAMPLE_DATA
PUT 'SAMPLE' 'VARIABLES' ('EDUCATION' 'INCOME' 'GENDER')
PUT 'SAMPLE' 'OBSERVATIONS' NAMES
PUT 'SAMPLE' 'CODEBOOK' SAMPLE_CB
```

The function *MAKE* constructs a new instance of a prototype, filling in the *PARENT* and *TYPE* slots. *PUT* changes the value of the named slot of an object or creates that slot if it does not exist. A corresponding function, *GET*, retrieves the value in any slot. (See line 10> of Figure 7 for an example.) The function *SLOTS* lists an object's slot names.

```
      ∇STANDARDIZE[⎕]∇
  [0] Z←MS STANDARDIZE X;M;S;shape
  [1]⍝Standardize a data matrix or vector
  [2]⍝.L (MS) 2-vector giving mean & standard deviation of result
  [3]⍝.L      Default: 0 1
  [4]⍝.R <X> Data matrix or vector
  [5]⍝.Z <Z> Standardized data matrix or vector
  [6] uses¨'MEAN' 'VARIANCE' 'COL'
  [7] uses¨'default' 'select' 'get_data' 'deselect'
  [8] 'MS' default 0 1
  [9] (M S)←MS
 [10] shape←⍴X←select get_data X
 [11] Z←(X-[2]MEAN X)÷[2](VARIANCE X←COL X)*0.5
 [12] Z←deselect M+S×Z←shape⍴Z

      X←12 19 '.' 5 2 11

      STANDARDIZE X
 0.3328 1.3917 . ¯0.72611 ¯1.1799 0.18153
```

Figure 11. The **STANDARDIZE** *Function Applies the* **select** *and* **deselect** *Filters to Process Missing Values Properly.*

4.2 Missing Data

Missing data in APL2STAT objects are represented by the period character, '.'. All APL2STAT high-level functions handle missing data automatically by applying a filter (`select`) to remove observations with missing data at the start of an operation. An inverse filter (`deselect`) is then applied to the result, filling in missing value codes where appropriate. As an illustration, consider the *STANDARDIZE* function shown in Figure 11 that recenters and scales each column of a data matrix to specified (or defaulted) mean and standard deviation. These computations are sandwiched between the `select` filter in line `[10]` and the `deselect` filter in line `[12]`.

Although high-level APL2STAT functions handle missing data appropriately, the APL2 primitive arithmetic functions do not accept the '.' missing data character. These functions typically signal a *DOMAIN ERROR* and halt computation, as illustrated in Figure 12. To avoid this difficulty, APL2STAT has a missing data operator, *MD*, that applies any scalar function (primitive, user-defined, or derived) to its arguments, returning a missing value whenever either argument is missing. The *MD* operator is used mostly to perform calculations or transformations directly in APL2 on data that may contain missing values. Even if the argument(s) are not missing, *MD* traps the computation; when an error is generated, *MD* prints a warning message and returns a missing value.

4.3 Selecting Observations

In statistical analysis one often wishes to analyze the data for a subset of observations in a data set, either to perform parallel analyses of different groups or to gauge the effect of setting certain observations aside. Most statistical systems require creating new data sets for each such selection. In APL2STAT the same mechanism used to filter missing data allows observations to be easily included or excluded from an analysis without modifying the data.

APL2STAT maintains a global system variable, *△SELECT*, a binary vector of length

```
      ⎕←SCORES←3 3⍴15 11 6 '.' 8 9 13 16 12
15 11  6
 .  8  9
13 16 12

      MEAN SCORES                    ⍝ MD not required
14 13.5 9

      ⍟SCORES                        ⍝ log does not compute
DOMAIN ERROR
      ⍟SCORES                        ⍝ log does not compute
      ∧

      ⍟MD SCORES                     ⍝ '.' stays missing
2.7081 2.3979 1.7918
 .      2.0794 2.1972
 2.565 2.7726 2.4849

      +MD/SCORES                     ⍝ row sums
32 . 41

      SCORES[1;1]←0

      ⍟MD SCORES                     ⍝ log 0 is trapped
DOMAIN ERROR, Argument: 0
 .     2.3979 1.7918
 .     2.0794 2.1972
2.565 2.7726 2.4849
```

Figure 12. Operations With Missing Data. **MD** *allows direct APL computations with missing data. The function argument to* **MD** *may include a primitive (such as ⍟, natural log), user-defined, or derived functions (such as +/). Errors with nonmissing data issue a warning message and return a missing value.*

equal to the number of observations in the current data set. *△SELECT* is initially all 1s, corresponding to selection of all observations. The `select` function uses the APL *compress* operator, /, in the expression *Z*←(*△MISSING*×*△SELECT*)/[1]*X* to return the rows of the data vector or matrix, *X*, that are selected (*△SELECT*[*I*]=1) and that contain no missing values (*△MISSING*[*I*]=1). Hence, changing an element of *△SELECT* to 0 causes the corresponding observation to be ignored in subsequent computations, though it remains in the data set. Observations may be excluded with the *OMIT* function (for example: *OMIT*¨ *'Detroit' 'Toronto'*). The observation selection vector may also be set by a recode or as the result of a logical expression, as in *△SELECT*←*INCOME*>20000.

4.4 TABLE OBJECTS

In the analysis of multiway tables by linear and log-linear models, the factors that classify the observations are implicit in the table dimensions. Rather than force the user to generate the factor variables explicitly, as is done in GLIM, SAS, New S, and most other statistical systems, APL2STAT TABLE objects contain just the multiway table and information describing the variables and dimensions of the table.

TABLE objects may be defined by APL2STAT functions or from data entered at the keyboard. In addition, the *TABLE* operator constructs a TABLE object from a data matrix whose columns include a set of one or more factor variables. The operator applies any (monadic) function— primitive, defined, or derived from another operator—to calculate a statistic (which may itself be an array) for the observations classified into the cells

```
      N←500                                     ⍝ number of observations
      INCOME←10+?N⍴40                           ⍝ uniform [10,50]
      REGION←('EAST' 'CENTRAL' 'WEST')[?N⍴3]
      GENDER←'MF'[?N⍴2]

      8↑[1] INCOME, REGION AND GENDER           ⍝ first 8 observations
 50 WEST    M
 30 WEST    F
 15 EAST    F
 47 WEST    M
 34 CENTRAL F
 28 CENTRAL M
 43 EAST    M
 16 EAST    F

      PRINT MEAN TABLE 'INCOME ON REGION AND GENDER'
  Row      |   Column
  REGION   |   GENDER
           F        M
  CENTRAL  30.082  30.61
  EAST     31.652  32.881
  WEST     31.095  30.397

      '(⍴⍵)(MEAN ⍵)' TABLE 'INCOME ON REGION AND GENDER'
LAST_TABLE

      LAST_TABLE
  PARENT       TABLE_PROTO
  TYPE         TABLE
  TABLE          85  30.082    77  30.61
                 92  31.652    84  32.881
                 84  31.095    78  30.397
  VARIABLES     REGION  GENDER
  CONTENTS     INCOME ON REGION AND GENDER
  DIMENSIONS   3 2
  LEVELS         CENTRAL  EAST  WEST    F  M
```

*Figure 13. Calculating a Table of Means. The **TABLE** operator applies the **MEAN** function to **INCOME** for observations classified by **REGION** and **GENDER**. The result is stored in a* TABLE *object (named **LAST_TABLE**) that is passed to the **PRINT** function. The second example uses an anonymous function to construct a table whose entries are the sample size (ρω) and mean in that cell. In anonymous functions the right argument is represented by ω and the left argument (if present) by α.*

defined by the factor variables. For example, the initial lines of Figure 13 create 500 random observations on variables *INCOME, REGION*, and *GENDER*. The statement, *MEAN TABLE 'INCOME ON REGION BY GENDER'*, applies the *MEAN* function to the *INCOME* variable for cells cross-classified by *REGION* and *GENDER*. (*ON* and *BY* are APL2STAT functions that join columns to form a matrix. By convention, *TABLE* uses the first column as the analysis variable, and creates a TABLE object named *LAST_TABLE* by default.)

It is simple to use the *TABLE* operator to construct contingency tables (e.g., +/ *TABLE '1, GENDER BY REGION'*), but because this is a common operation, a *COUNT* function is provided. Objects created by *COUNT* descend from their own prototype, which itself is a descendent of the table prototype.

5. METHODS

The APL2STAT object system incorporates methods appropriate for classes of objects in the form of generic functions that automatically invoke class-specific subfunctions. For example, the *PRINT* function prints an object on the screen in an appropriate format.

Other examples of methods are *HYPOTHESIS*, which tests linear hypotheses on model objects, for example, by Wald or F-tests, and *TEST*, which compares nested models, for example, by likelihood-ratio or incremental F-tests. See the illustrative APL2STAT session in Figure 7 (p. 396) for the use of *PRINT* and *HYPOTHESIS* methods.

A generic function such as *PRINT* calls a specific subfunction appropriate for the type of object that it processes. By our convention, the specific subfunctions have names of the form *function_TYPE*: for example, *print_DATASET* and *print_LM*. The *METHOD* function searches in the active workspace and along the search path for a method appropriate to an object; if no such method exists, then the ancestors of the object are searched in order of proximity. Thus, for example, if no more appropriate method exists, an object is printed with the function *print_OBJECT*, belonging to the root object in the object tree.

In the current version of APL2STAT, methods exist as global functions, either in the active workspace or in library files (Section 6), but it would be simple to achieve greater encapsulation by adding a slot for each method specific to a given object containing the character representation of the corresponding method function (created by the APL2 system function ⎕*CR*). To access the method, the character representation of the function would be extracted from the object, defined as a local function (via the "fix" system ⎕*FX*), and executed. As in the current version of APL2STAT, methods would be inherited via the object tree, and indeed almost all methods would reside in prototype objects.

Finally, it is trivial to change the style of methods from generic function calls to messages sent to objects. A message function with syntax *MESSAGE 'object' 'method' optional arguments* simply invokes the corresponding method function, as in *MESSAGE 'LAST_LM' 'PRINT'*. We prefer the more direct style of *PRINT*, but both styles are supported.

6. THE APL2STAT SHELL

All interaction with APL, and therefore with APL2STAT, takes place under control of the APL interpreter, which operates in a "read-evaluate" loop, as in Lisp and other similar interactive systems. APL2STAT augments the APL2 interpreter and session manager by running under control of a shell program—an APL2 function that simulates, but extends, the normal read-evaluate loop. The *SHELL* program provides for dynamic loading of functions and data; implements simple memory management; and simulates (and modifies) the behavior of APL system commands, which cannot normally be issued under program control. The APL2STAT system currently consists of several hundred functions, operators, and system variables. Because APL programs can only access other programs and data contained in memory (the active workspace), it is usually necessary to load all functions and subfunctions before they are invoked. This is not a serious limitation for the commercial APL2 interpreter running on a well-endowed 386 or better machine (because the entire APL2STAT system occupies less than 500 Kbytes of memory), but it is a substantial limitation when running APL2STAT under the TryAPL2 system.

All APL2STAT functions are therefore written to check for the presence of auxiliary functions and variables, and to copy them from library files if they are not already present

in the active workspace. A global system variable, *∆PATH*, is a nested vector containing the pathnames of one or more library files. Each APL2STAT function begins with a call to the `uses` function, such as this line from the *STANDARDIZE* function shown in Figure 11:

```
uses¨'MEAN' 'VARIANCE' 'COL'
```

The `uses` function then searches through the files named in *∆PATH* for any function, operator, or variable that is not present in the active workspace. Because the APL objects in the APL2STAT library files are marked by type (function, operator, or variable), this search is relatively efficient, although there is usually a noticeable pause the first time many functions need to be loaded. (The `uses` function takes an optional left argument to specify the type of object sought, with a default of *'F'* for function.)

In addition, the *SHELL* parses each expression entered by the user for names of programs or data objects. Any objects not present in the workspace are searched and loaded along the *∆PATH*. Because the object type is not known, however, the *SHELL* searches first for a function, then for a variable, and finally for an operator. The APL2STAT function *ATTACH* adds the name of a file containing APL objects to the search path, allowing the position on the path to be specified. The search path therefore provides a convenient way to extend the APL2STAT system, to access data files, or to test and replace APL2STAT functions with new versions because the new versions will be found first if they are placed earlier on the path.

7. CONCLUDING REMARKS

In this article, and through the design of APL2STAT, we have attempted to demonstrate the potential of APL2 as a programming language for statistical applications, and to suggest that APL's reputation for opacity is undeserved. It is difficult, however, to convey one of the principal virtues of APL2: the ease with which applications can be developed and coded. One of us (Fox), for example, designed and programmed the basics of the APL2STAT object system, programming environment, and graphics system, along with perhaps two-thirds of the current APL2STAT functions and operators in six weeks.

An example of the ease with which nonstandard problems may be addressed with APL2STAT is given in Figure 14, which illustrates how a least-squares regression analysis can be bootstrapped by resampling the vector of residuals when the model-matrix is to be treated as fixed. A fixed model-matrix is at least arguably appropriate for Duncan's regression. The initial least-squares fitted values and residuals are extracted from an object returned by the *LINEAR_MODEL* function; 100 bootstrap samples of residuals are selected; and these resampled residuals, along with the fitted values, are employed to obtain bootstrapped replications of the regression coefficients. The solution to this problem in APL2 was produced, from idea to output, in less than five minutes. The computations themselves, including 100 matrix inversions, were essentially instantaneous on a PC with a 486 processor. Likewise, a general function for robust linear regression providing M-estimates (e.g., Huber 1981) for arbitrary, user-defined weight functions and scale estimates comparable to that of Heiberger and Becker (1993) was programmed in

```
      LINEAR_MODEL 'PRESTIGE←INCOME+EDUCATION'

      YHAT←GET 'LAST_LM' 'FITTED VALUES'

      E←GET 'LAST_LM' 'RESIDUALS'

      ⍝ 100 bootstrapped samples of residuals
      EE←⊂[2](100,⍴E)⍴E[?(100×⍴E)⍴⍴E]

      ⍝ 100 bootstrapped Y vectors
      YY←(⊂YHAT)+¨EE

      ⍝ bootstrapped coefficients
      B←⊃[2]YY⌹¨⊂1,INCOME AND EDUCATION

      MEAN B          ⍝ means of bootstrapped coefficients
¯5.827960301 0.5932116321 0.5459577236

      VARIANCE B      ⍝ variances of bootstrapped coefficients
16.45033015 0.0141370544 0.008286909976
```

Figure 14. Bootstrapping the Duncan Regression Model With a Fixed Model-Matrix

under an hour. Experiences such as these convince us that there is still a niche for APL in statistical computation for both research and teaching.

There are several directions in which we plan to develop APL2STAT. Because of the rich heritage of statistical applications programmed in APL (Friendly 1991), and because of the manner in which data are handled in APL2STAT, it will be relatively simple to incorporate additional statistical methods. We also plan to make the APL2STAT graphical system more object-oriented—at present, APL2STAT graphics objects are simply nested vectors of calls to a graphics processor, which can be saved, added to, or replayed, but which are not easily modified or linked to other graphs. This extension would be most useful in a graphical environment supporting multiple graphics windows, such as the IBM APL2 implementation for OS/2, RS/6000, and Sun/Solaris. Readers wishing to experiment with APL2STAT may obtain the APL2 and TryAPL2 versions by anonymous ftp from `watserv1.waterloo.edu` in the directory `/languages/apl/workspaces/apl2stat`.

[*Received July 1993. Revised June 1994.*]

REFERENCES

Alfonseca, M. (1989a), "Object-Oriented Programming in APL2," *APL Quote Quad,* 19, 6–11.

——— (1989b), "Frames, Semantic Networks, and Object-Oriented Programming in APL2," *IBM Journal of Research and Development,* 33, 502–510.

Anscombe, F. J. (1981), *Computing in Statistical Science Through APL,* New York: Springer–Verlag.

Becker, R. A., Chambers, J. M., and Wilks, A. R. (1988), *The New S Language,* Pacific Grove, CA: Wadsworth.

Benkard, J. P. (1990), "Nonce Functions," *APL Quote Quad,* 20, 27–39.

Brown, J.A., Pakin, S., and Polivka, R. P. (1988), *APL2 at a Glance,* Englewood Cliffs, NJ: Prentice–Hall.

Eusebi, E. V. (1985), "Operators for Program Control," *APL Quote Quad,* 15, 181–189.

Falkoff, A. D., and Iverson, K. E. (1968), *APL\360 User's Manual,* IBM Corp.

Friendly, M. (1991), "APL2 Tools for Multivariate Data Analysis," Report No. 200, York University, Department of Psychology.

Friendly, M., and Fox, J. (1993), "Using APL2 to Create an Object-Oriented Environment for Statistical Computation," Report No. 214, York University, Department of Psychology.

Gilman, L., and Rose, A. J. (1984), *APL: An Interactive Approach,* (3rd ed.) New York: John Wiley.

Heiberger, R. M. (1989), *Computation for the Analysis of Designed Experiments,* New York: John Wiley.

Heiberger, R. M., and Becker, R. A. (1993), "Design of an S Function for Robust Regression Using Iteratively Reweighted Least Squares," *Journal of Computational and Graphical Statistics,* 1, 181–196.

Huber, P. (1981), *Robust Statistics,* New York: John Wiley.

IBM Corporation, (1991a), *APL2 Programming: APL2 for the IBM PC User's Guide,* SC33-0600-02, San Jose, CA: Author.

IBM Corporation, (1991b), *TryAPL2,* IBM Santa Teresa, San Jose, CA: Author.

Iverson, K. E. (1962), *A Programming Language,* New York: John Wiley.

Polhemus, N. W. (1983), "Interactive Data Analysis and Statistical Graphics in APL," *APL Quote Quad,* 13, 45–52.

Sykes, A. M., and Hawkes, A. G. (1989), "Using APL2 in Statistics," *APL Quote Quad,* 20, 346–354.

Thompson, N. L. (1992), "APL2 as a Specification Language for Statistics," *IBM Systems Journal,* 30, 539–542.

Tierney, L. (1990), *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics,* New York: John Wiley.