*Sociological Methods & Research* 1995 23: 282

*As has often been recognized, several characteristics of APL make it especially well suited for programming statistical applications: APL treats arrays—vectors, matrices, and higher dimensional arrays—as data structures that can be processed without reference to their elements; it contains many powerful primitive functions and operators for manipulating data; and it is an extensible language, so that user-written programs are employed in the same manner as the primitives. APL2 incorporates several significant extensions of the language, the most important of which are generalized arrays—arrays whose elements may themselves be arrays. Using generalized arrays along with other features of APL2, the authors build an object-oriented programming environment for statistics and data analysis called APL2STAT.*

# Data Analysis Using APL2 and APL2STAT

JOHN FOX
MICHAEL FRIENDLY
*York University*

The programming language APL, designed by Iverson during the 1960s (Iverson 1962; Falkoff and Iverson 1968), has often been touted as an ideal programming language for statistical applications. As we shall see, APL treats data arrays, such as vectors, matrices, and higher dimensional tables, as data structures that can be processed without explicit reference to their elements. Moreover, APL contains a variety of powerful primitive (i.e., built-in) functions and operators for manipulating data, and APL is an extensible language, so that functions and operators written by users are employed in the same manner as are the primitives.

Previous treatments of APL for statistical computation have taken the form of illustrations of the power of APL for solving statistical problems (e.g., Anscombe 1981) or of collections of programs—sometimes quite extensive—implementing statistical algorithms in APL (e.g., Heiberger 1989; Friendly 1991). There has even been a comprehensive statistical program package (STATGRAPHICS) written in the language. What has been lacking, however, is an extensible programming environment based on APL that provides common

SOCIOLOGICAL METHODS & RESEARCH, Vol. 23 No. 3, February 1995 282-302
© 1995 Sage Publications, Inc.

facilities for handling statistical data and other objects that arise naturally in the process of analyzing data. We endeavor to provide this environment in APL2STAT.

APL2 is a modern dialect of APL that substantially extends the capabilities of an already powerful programming language. Most significantly, APL2 incorporates generalized (or nested) arrays in which the elements of an array—a matrix, for example—may themselves be arrays. Nested arrays are therefore like the lists in LISP but are more convenient because they can have two-dimensional or higher dimensional structure. We use nested matrices, for example, to build the object system in APL2STAT (adapting the work of Alfonseca 1989a, 1989b, on object-oriented programming in APL2 and borrowing ideas from S and Lisp-Stat).

## *USING APL2*

APL is usually implemented in an interpreted programming environment in which each APL statement entered at the keyboard by a user is translated and executed. Provisions are also made for storing APL statements as programs, which are later interpreted and executed as needed.

APL is a functional programming language. The APL primitive functions take data as arguments and return values as results. For example,

```
      1+2
3
```

Note that input from the user (`1+2`) is automatically indented six spaces, whereas output produced by the interpreter (`3`) appears at the left margin. Special characters, many of them not in the standard ASCII character set, are used to represent the APL primitives; thus, for example, × represents multiplication and ÷ represents division:

```
      2×3
6
      2÷3
0.6666666667
```

Primitive functions in APL may take one or two arguments; most symbols have both a one-argument (monadic) and two-argument (dyadic) use. Thus subtraction is the dyadic use of – whereas negation is its monadic use:

```
      1-2
¯1
      -2
¯2
      -¯2
2
```

APL distinguishes between negation, which is a function, and the minus sign that is part of a negative number (called the "high minus"). Monadic functions always take a right argument.

The result of a computation may be assigned to a variable, which may then be used in further computations. For example,

```
      SUM←1+2
      2×SUM
3
```

The assignment operator ← is usually read as "gets"; thus "`SUM` gets `1+2`."

APL directly supports only one data structure: the array, the elements of which may be numbers or characters (or, in APL2, other arrays). Arrays arise naturally in the course of computation or may be defined directly, as explained later. A single number, such as `1.3`, or character, such as `'A'`, is called a scalar and is considered a zero-dimensional array. A one-dimensional array of numbers or characters (e.g., `1 2 3` or `'ABC'`) is called a vector; a two-dimensional array is called a matrix. APL permits arrays of any dimension.

All scalar functions (such as +,–, ×, and ÷) apply in a one-to-one fashion to the elements of vectors, matrices, and higher dimensional arrays. For example,

```
      1 2 3 + 3 2 1
4 4 4
      2 × 1 2 3
```

```
2 4 6
       - 1 2 3
¯1 ¯2 ¯3
```

For these operations to make sense, however, the arguments must be of the same size and shape or one must be a scalar; otherwise, the interpreter signals a length error:

```
       1 2 + 1 2 3
LENGTH ERROR
       1 2+1 2 3
       ^
```

In mixed functions, there is a more complex relationship between the size and shape of the arguments and the size and shape of the result. For example, the index generator, the monadic use of ι (Greek iota), takes a nonnegative integer as an argument and returns a vector of integers up to the argument:

```
       ι6
1 2 3 4 5 6
```

The dyadic use of ρ (Greek rho) reshapes its right argument into an array according to its left argument:

```
       2 3 ρ 1 2 3 4 5 6
1 2 3
4 5 6
       2 2 3 ρ ι12
1 2 3
4 5 6

7 8 9
10 11 12
```

Note that the planes or layers of a higher dimensional array are displayed one below the other.

APL2 arrays may have other arrays as their elements and thus can represent a variety of complex data structures (such as lists and trees) in addition to the multiway tables provided by other languages. For example, the following three-element nested vector (broken across three lines for display in the text) might be used to represent a small data set:

```
          DATA←('Jim' 'Ted' 'Bernice')
               (3 2 ρ 23 'M' 29 'M' 28 'F')
               ('AGE' 'GENDER')
```

The first element of this vector, the nested character vector (`'Jim' 'Ted' 'Bernice'`), represents observation (i.e., row) names. The second entry is the data matrix, which consists of both numbers and characters:

```
23 M
29 M
28 F
```

The third entry of `DATA` is another nested character vector containing variable (column) names (`'AGE' 'GENDER'`).

Elements of arrays may be selected in many different ways, most straightforwardly using bracketed subscripts; dimensions are separated by semicolons. Arrays can also be indexed by arrays. Here are some examples:

```
       A←1 2 3
       A[2]
2
       B←2 3ρ ι6
       B
1 2
3 4
5 6
       B[2;1]
3
       B[1;]
1 2
  B[;2]
2 4 6
       B[1 3;2]
2 6
```

In most programming languages, there are several statement types and a few primitives. In APL2, there is essentially one type of statement and about 100 primitives. Because there are so many primitives, all are assigned the same precedence; precedence rules (e.g., exponentiation takes place before addition) would just be too difficult to remember. Moreover, apparently perversely but for good—if abstract—reason, APL expressions are executed from right to left. Thus `2×3-1` is `4`, not `5`. Parentheses can clarify what is going on here and can be used to change the order of evaluation:

```
        2×3-1
4
        2×(3-1)
4
        (2×3)-1
5
```

Thus far, we have illustrated the use of the APL2 system in immediate-execution mode; APL expressions are processed by the interpreter line by line as they are entered by the user. APL systems also support a function-definition mode in which lines entered by the user are saved in a user-defined function for subsequent execution. The inverted-delta character ∇, called "del," is used to switch between immediate-execution and function-definition modes. We illustrate by entering a user-defined function ("program") to compute the mean of a vector:

```
        ∇M←MEAN X;N
[1]     N←⍴X
[2]     M←(+/X)÷N
[3]     ∇
        MEAN ⍳10
5.5
        2×MEAN ⍳10
11
```

The `MEAN` function, although particularly simple, demonstrates several characteristics of defined functions. First, the function header includes the name of the function (`MEAN`), its formal argument (`X`), and the formal result variable (M) returned by the function. Second, other variables named in the header (here N) are, along with the argument and the result variables, local to the function; they exist only so long as the function executes, and they "shadow" objects by the same name that exist globally in the workspace. If, for example, there is a variable named X defined globally in the active workspace, then it is inaccessible to MEAN and, more importantly, its value will not be changed by MEAN. Third, when the function is called with a real argument (here ⍳10), that value is passed through the formal argument to the function; the result returned by the function may be assigned to a (global) variable or used in a computation in exactly the same manner as the result of a primitive function.

The function MEAN uses the monadic shape function ⍴ to return the number of entries in the vector X. Here is a simpler example:

```
      ⍴ 5 4 1 2
4
```

MEAN also uses the reduction operator / to sum the entries in X. Operators are "functions" that take functions as arguments, producing a derived function that then may be applied to a data argument or arguments. The reduction operator takes a single scalar dyadic function as its left argument and, in effect, places that function between entries in its right argument. Thus +/X is the sum of the entries in the vector X:

```
      +/ 1 2 3 4
10
```

For a matrix X, the expression +/X or +/[2]X produces row sums (i.e., across the second, or column, coordinate), whereas +/[1]X produces column sums (across the first, or row, coordinate).

By contrast, the expression ×/X produces a continued product rather than a sum:

```
      ×/1 2 3 4
24
```

Because they may be used with many functions, both primitive and user defined, operators such as reduction solve a class of algorithmic problems rather than a single problem.

## *USING APL2STAT*

The APL2 system[1] includes a full-screen session manager that provides a variety of services including logging sessions to disk files, permitting the user to scroll backward and forward in a session, and supporting the editing and reexecution of lines. APL2STAT provides a SHELL function to enhance the session manager; among other services, the SHELL will automatically search along a modifiable path of files containing APL functions and variables. This notion is adapted from S. To bring the SHELL into memory (the "active workspace") and to execute it, we load the SHELL workspace via the APL ) LOAD command:

```
      )LOAD SHELL
APL2STAT (SHELL)   [Version 2.01] 1993 John Fox &
                    Michael Friendly
 Search path:      C:\APL2\STAT\UPDATE.201
                   C:\APL2\STAT\APL2STAT.200
                   C:\APL2\STAT\DATA.TRY
```

The file DATA.TRY contains a variety of data sets distributed with APL2STAT, including Duncan's occupational prestige data. APL2STAT.200 contains the functions and operators for version 2.00 of APL2STAT, whereas UPDATE.201 contains new functions and modified functions that differentiate version 2.01 from version 2.00. Because UPDATE.201 is the first file on the path, functions will be located here first.

To print the Duncan data set, we type

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

Most of the output is suppressed for brevity; the line number (`1>`) is supplied by the `SHELL`, which located the `PRINT` function in `APL2STAT.200` and the `DUNCAN` data set in `DATA.TRY`. `PRINT` is an example of a generic function; it picks a method appropriate for printing the object whose name is given as its argument. A data set object is printed with variable names labeling columns and observation names labeling rows. As we see later, other types of objects (e.g., a linear-model object) are printed differently.

APL2STAT objects are two-column nested matrices. The first column contains "slot names"; the second column holds the contents of each slot. A data set object includes the following slots:

```
2> SLOTS 'DUNCAN'
 PARENT
 TYPE
 VARIABLES
 OBSERVATIONS
 DATA
 CODEBOOK
```

Every object in APL2STAT has a `TYPE` and a `PARENT`; the `DUNCAN` data set object is of type `'DATASET'` and has the parent `'DATASET_PROTO'`. The `VARIABLES` slot contains variable names, the `OBSERVATIONS` slot contains observation names, the `DATA` slot contains the data matrix, and the `CODEBOOK` slot contains a description of the data set. Specific methods (e.g., for printing) are determined from an object's type; slots and methods may be "inherited" from an object's parent, from the parent's parent, or from a more remote ancestor.

Any number of data sets may be kept on file or in memory, but only one data set is active. A data set is made active with the `USE` function:

```
3> USE 'DUNCAN'
 Variables created: INCOME EDUCATION PRESTIGE
4> USING
Current dataset: DUNCAN
 Contains 45 observations
 45 observations selected
 45 observations with valid data (in last operation)
 3 variables: INCOME EDUCATION PRESTIGE
```

USE creates APL variables corresponding to the columns of the data matrix and stores the observation names in the nested vector ΔOBS_NAMES. Note that APL2STAT supports missing data (encoded, as in SAS, by the character '.'), although there is no missing data in Duncan's data set.

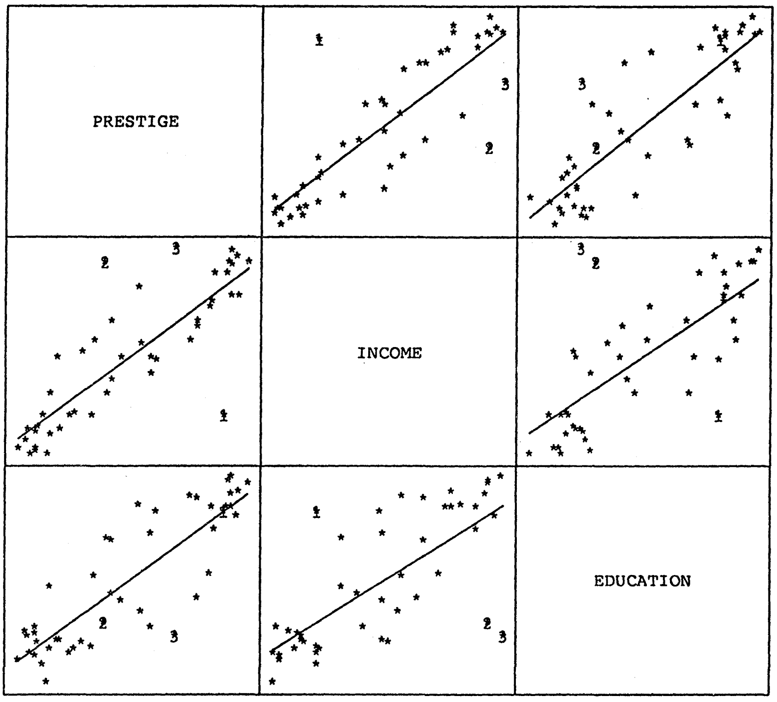
Several facilities exist in APL2STAT for obtaining information. APL2STAT functions and operators are self-documenting, and the documentation may be accessed via the HOW function. For example, to obtain information about the PAIRS function, which creates a scatterplot matrix,

```
5> HOW 'PAIRS'
Purpose:
       Scatterplot matrix with interactive point
       identification. To identify a point, move
       the cursor near the point using the cursor
       keys or a mouse and press any key or the
       left mouse button...
Usage:
       type PAIRS vars
       . . .
```

(The information returned by HOW is abbreviated here.) To create a scatterplot matrix for the three variables in the DUNCAN data set, we enter

```
6> PAIRS 'PRESTIGE, INCOME AND EDUCATION'
```

The resulting graph is shown in Figure 1; the flagged observations were identified with a mouse. It is apparent that ministers have unusually high prestige given their relatively low level of income and



1.minister
2.railroad_conductor
3.railroad_engineer

**Figure 1: A Scatterplot Matrix for Duncan's Occupational Prestige Data**
NOTE: The highlighted points were identified interactively with a mouse.

that railroad conductors have relatively low prestige given their relatively high income. Railroad conductors and engineers have high incomes given their levels of education, whereas ministers have low income given their education. These observations will likely require special attention in a regression of prestige on income and education.

To perform least-squares regression, we employ the `LINEAR_MODEL` function:

```
7> LINEAR_MODEL 'PRESTIGE←INCOME+EDUCATION'
LAST_LM
```

The LINEAR_MODEL function takes a symbolic model specification as a right argument. The syntax is similar to that in SAS or S. A more complex model could include specifications for interactions, transformations, or nesting, for example. Because both INCOME and EDUCATION are quantitative variables, a linear regression model is fit; character variables (e.g., a REGION variable with values 'East', 'Midwest', etc.) are treated as categorical, and dummy regressors are suitably generated to represent them. APL2STAT also makes provision for ordered-category data.

Rather than printing a report, the LINEAR_MODEL function returns a linear-model object; because no name for the object was specified, the function used the default name LAST_LM. This object contains the following slots (for brevity, most are suppressed):

```
8> SLOTS 'LAST_LM'
 PARENT
 TYPE
 COEFFICIENTS
 COV MATRIX
 RESIDUALS
 FITTED VALUES
 RSTUDENT
 HATVALUES
 COOKS D
 ...
```

We could retrieve the contents of a slot directly, using the GET function, but it is more natural to process LAST_LM using functions meant for linear-model objects. PRINT, for example, will find an appropriate method to print out a report of the regression:

```
9> 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     0
  Residuals       42     7506.7
  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
```

Likewise, the INFLUENCE_PLOT function plots studentized residuals against hatvalues, using the areas of plotted circles to represent Cook's D influence statistic. The result is shown in Figure 2, where the labeled points were identified interactively.

```
10> INFLUENCE_PLOT 'LAST_LM'
```

This analysis (together with some diagnostics not shown here) led us to refit the Duncan regression without ministers and railroad conductors:

```
11> OMIT 'minister' 'railroad_conductor'
12> 'NEW_FIT' LINEAR_MODEL 'PRESTIGE←INCOME+
    EDUCATION'
NEW_FIT
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
 . . .
```

The OMIT function locates the observations in the observation-names vector and places corresponding zeroes in the observation-selection vector ΔSELECT. (APL2STAT functions and operators use ΔSELECT to determine which observations to include in a computation; observations corresponding to entries of one are included and those to entries of zero are not.) Notice that we explicitly name the new fitted

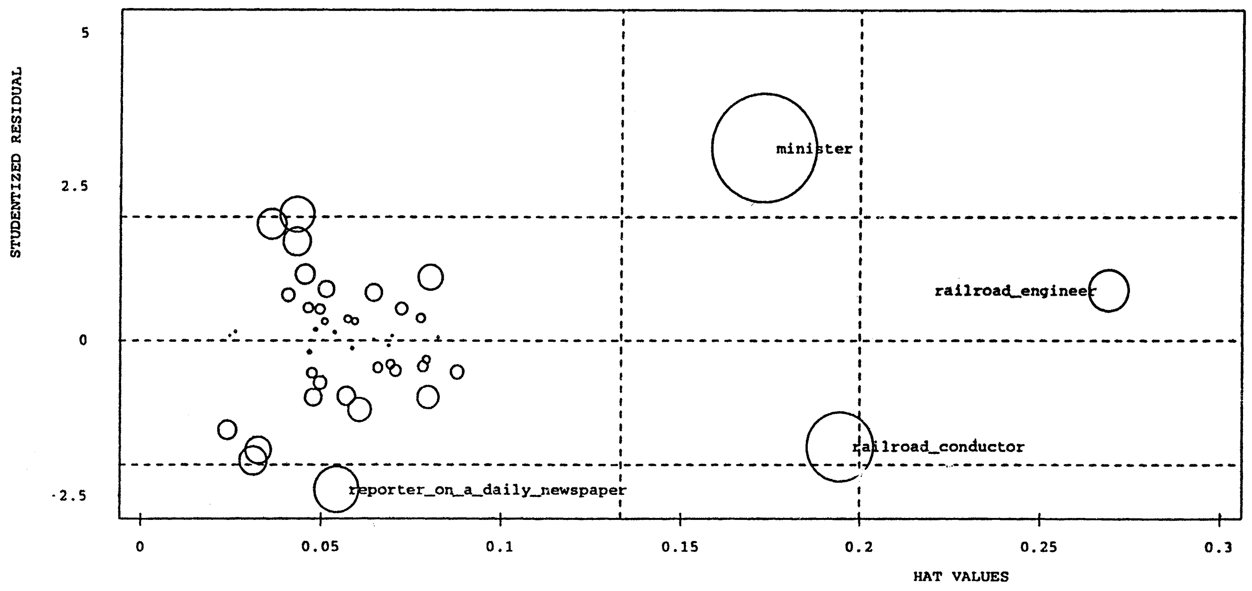


**Figure 2: A Plot of Studentized Residuals by Hatvalues for Duncan's Regression**
NOTE: The areas of the circles are proportional to Cook's D statistic, a measure of influence in the regression. The labeled points were identified interactively with a mouse. The horizontal lines are at studentized residuals of zero and ±2; the vertical lines are at twice and three times the average hatvalue.

model NEW_FIT to avoid overwriting LAST_LM. The income coefficient is substantially larger than before, and the education coefficient is substantially smaller than before.

## *PROGRAMMING (AND BOOTSTRAPPING) ROBUST REGRESSION IN APL2*

APL2STAT includes a general and flexible function for robust regression, but to illustrate how programs are developed in APL2, we write a robust-regression function that computes an M-estimator using the bisquare weight function. To keep things as simple as possible, we bypass the general data handling and object facilities provided by APL2STAT. Similarly, we do not make provision for specifying alternative weight functions, scale estimates, tuning constants, and convergence criteria, although it would be easy to do so.

A good place to start is with the bisquare weight function, which may be simply written in APL in the following manner:

```
    ∇WT←BISQUARE Z
[1] WT←(1>|Z)×(1-Z*2)*2
[2] ∇
```

The monadic function | in APL2 is absolute value, * is exponentiation, and the relational function > returns 1 for "true" and 0 for "false." In addition to the weight function, we require a function for weighted least-squares (WLS) regression:

```
    ∇B←WT WLS YX;Y;X
[1] Y←,1↑[2]YX
[2] X←1↓[2]YX
[3] WT←WT*0.5
[4] B←(Y×WT)⌹X×[1]WT
[5] ∇
```

- The function WLS takes weights as a left argument and a matrix, the first column of which is the dependent variable, as a right argument.
- The first line of the function extracts the first column of the right argument, ravels this column into a vector (using ,), and assigns the

result to Y; the second line extracts all but the first column, assigning the result to X.

- The square roots of the weights are then computed, and the regression coefficients B are calculated by ordinary least-squares regression of the weighted Y values on the weighted Xs. The domino or quad-divide symbol (⌹) used dyadically produces a least-squares fit. (Used monadically, this symbol returns the inverse of a nonsingular square matrix.)

Our robust-regression function takes a matrix right argument, the first column of which is the dependent variable, and returns regression coefficients as a result:

```
      ∇B←ROBUST YX;ITERATION;LAST_B;X;Y
[1]   Y←,1↑[2]YX
[2]   X←1,1↓[2]YX
[3]   B←Y⌹X
[4]   ITERATION←0
[5]   LOOP:→(100<ITERATION←ITERATION+1)/NO_CONVERGE
[6]   LAST_B←B
[7]   E←Y-X+.×B
[8]   SCALE←MEDIAN|E-MEDIAN E
[9]   WEIGHTS←BISQUARE E÷6×SCALE
[10]  B←WEIGHTS WLS Y,X
[11]  →(∨/(0.0001×1E¯6+|B)<|B-LAST_B)/LOOP
[12]  →0
[13]  NO_CONVERGE:'MAXIMUM ITERATIONS EXCEEDED'
[14]  ∇
```

- The first two lines of the function extract the dependent-variable vector Y and independent-variable matrix X, appending a column of ones to X for the constant term.
- Line 3 sets the regression coefficients B initially to the least-squares estimates.
- Lines 4, 5, and 11 set up the iterative calculation of the robust-regression estimates. Unlike most programming languages, APL has no standard control structures for iteration; because arrays are processed as units, however, looping is usually necessary only for truly iterative processes. Line 4 initializes the loop index. Line 5 increments the index and checks whether the number of iterations exceeds 100; if so, the function branches (→) to NO_CONVERGE and prints a message. Line 11 checks for convergence; if any coefficient has changed from the previous

iteration (LAST_B, set in line 6) by more than 0.0001 times its absolute value, then another iteration is performed. If, alternatively, the convergence criterion is met, then the function exits, which is the effect of the branch to the nonexistent line 0 on line 12. The factor 1E¯6 (i.e., $1 \times 10^{-6}$) prevents computational instability due to values in B that are very close to zero.

- Residuals are calculated in line 7, where +.× is the matrix product (a special case of the general inner product operator .).
- The SCALE estimate in line 8 is the median absolute deviation from the median residual; the APL2STAT MEDIAN function is employed here.
- Weights are calculated in line 9 and applied to the WLS regression in line 10, using a "tuning constant" of 6 for the bisquare weight function.[2]
- Notice that the variables E, SCALE, and WEIGHTS are not in the function header and hence are global variables; their values will persist after the function terminates. We prefer to encapsulate quantities such as residuals and weights in objects.

Applying the robust-regression function to Duncan's data produces

```
14> ROBUST PRESTIGE ON INCOME AND EDUCATION
¯7.4934 0.84643 0.39014
15> SMALLEST WEIGHTS
  minister                                   0
  railroad_conductor                       0.026231
  reporter_on_a_daily_newspaper            0.044191
  insurance_agent                          0.2118
  trained_machinist                        0.38765
```

Because the APL2STAT functions ON and AND are column catenators, the expression PRESTIGE ON INCOME AND EDUCATION evaluates to a three-column matrix, which becomes the right data argument to the ROBUST function. The robust regression coefficients are very similar to those produced by least-squares regression after ministers and railroad conductors are deleted. Indeed, a look at the the weights (using the SMALLEST function, which prints the five smallest values by default) reveals that these two occupations are the most discrepant and therefore receive very small weights (zero in the case of ministers). Reporters are also given nearly zero weight because they have a relatively large residual; because reporters are at a low leverage point, however, their inclusion has little effect on the result (recall Figure 2).

To illustrate the expressive power of user-defined operators, we employ the APL2STAT BOOTSTRAP operator rather than write a function that just bootstraps robust regression. BOOTSTRAP works with any function (e.g., our ROBUST) that takes a right data matrix argument and returns a result containing estimates.

To obtain 100 bootstrap replications of the robust regression for Duncan's occupational prestige data:

```
16> REPS←100 ROBUST BOOTSTRAP PRESTIGE ON INCOME
AND EDUCATION
 Result for full sample: ¯7.4934 0.84643 0.39014
Beginning BOOTSTRAP replications
MAXIMUM ITERATIONS EXCEEDED
BOOTSTRAP replications completed.
17> ρREPS
100 3
```

Notice that the robust-regression estimator failed to converge in one of the 100 bootstrap samples, a risk of a "redescending" M-estimator such as the bisquare.

To analyze the bootstrapped estimates, we need to tell APL2STAT that the current data set has 100 observations. We can then easily compute the mean and standard deviation of the bootstrapped estimates:

```
18> OBSERVATIONS 100
19> MEAN REPS
¯7.3169 0.7634 0.44807
20> (VARIANCE REPS)*.5
3.5857 0.1894 0.1497
```

Likewise, we can compute simple confidence intervals from the quantiles of the bootstrap estimates. Here are 95% confidence intervals for the income and education slopes (which are in the second and third columns, respectively, of REPS):

```
21> .025 .975 QUANTILES REPS[;2]
0.35248 1.1218
22> .025 .975 QUANTILES REPS[;3]
0.15384 0.77911
```

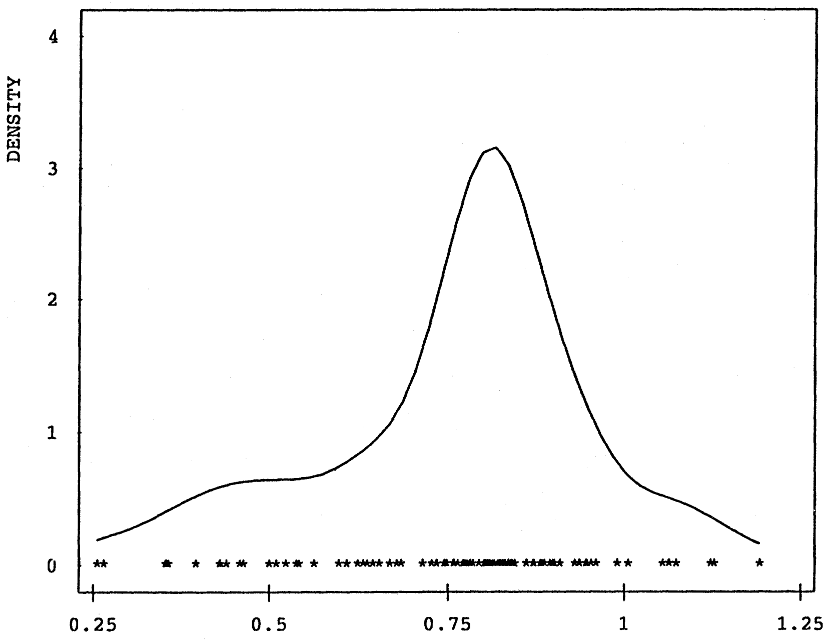


**Figure 3: Kernel Density Estimate for the Distribution of 100 Bootstrapped Bisquare Estimates of the Income Slope in Duncan's Regression**

NOTE: The estimates themselves are displayed as points near the bottom of the plot.

Finally, we use the DENSITY_ESTIMATE function from APL2STAT to plot a kernel density estimate for the bootstrapped income slopes (shown in Figure 3):

```
23> DENSITY_PLOT REPS[;2]
  Window width: 0.051837
```

## *CONCLUDING REMARKS*

Although APL is one of the oldest programming languages in use, APL2STAT is less than two years old and is still in the process of development. We plan to add to the capabilities of APL2STAT by incorporating additional statistical functions—a comparatively simple task, as there is a rich legacy of statistical methods programmed in APL. We also plan to generalize the APL2STAT object system, which was touched on only briefly in this article (and is described more fully in Fox and Friendly 1993 and in Friendly and Fox 1993). Finally, we plan to port APL2STAT to a windowed environment capable of supporting the type of object-oriented graphics featured in Lisp-Stat.

It is, unfortunately, difficult to convey the principal virtue of working in APL: the ease with which applications can be programmed. Relatively simple statistical programs (e.g., the BOOTSTRAP operator) can usually be written out virtually instantly. More complex tasks—such as the PAIRS function, which creates scatterplot matrices with point identification (as in Figure 1)—take perhaps an afternoon. Because a statistical programming environment invites the user to augment and improvise, such simplicity and ease of use are cardinal virtues. For more general information about programming in APL2, see Gilman and Rose (1984) or Brown, Pakin, and Polivka (1988).

## NOTES

1. By the APL2 system, we mean IBM's commercial APL2 PC system (IBM 1991a) or the similar freeware TryAPL2 (IBM 1992b). Other modern APL systems provide similar services. TryAPL2 and APL2STAT are available free of charge by anonymous ftp from watserv1.waterloo.edu. TryAPL2 is located in the directory /languages/apl/tryapl2. Version 2.0 of APL2STAT is available in archives compressed by PKZIP 2.04c in the directory /languages/apl/workspaces/apl2stat. The file a2stry20.zip is for use with TryAPL2; the file a2satf20.zip is for use with APL2/PC.

2. For efficiency when errors are normally distributed as well as robustness of efficiency and resistance to outliers, a tuning constant of 6 is a common choice for the bisquare weight function when the median absolute deviation is used as an estimator of scale (see, e.g., Goodall 1983).

## REFERENCES

Alfonseca, M. 1989a. "Frames, Semantic Networks, and Object-Oriented Programming in APL2." *IBM Journal of Research and Development* 33:502-10.

———. 1989b. "Object-Oriented Programming in APL2." *APL Quote Quad* 19:6-11.

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

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

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

Fox, J. and M. Friendly. 1993. *APL2STAT User Information.* (Distributed with APL2STAT)

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

Friendly, M. and J. Fox. 1993. "Using APL2 to Create an Object-Oriented Environment for Statistical Computation." Report No. 214, York University, Department of Psychology. (in press in the *Journal of Computational and Graphical Statistics*)

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

Goodall, C. 1983. "M-Estimators of Location: An Outline of the Theory." Pp. 339-403 in *Understanding Robust and Exploratory Data Analysis*, edited by D. C. Hoaglin, F. Mosteller, and J. W. Tukey. New York: Wiley.

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

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

———. 1991b. *TryAPL2.* San Jose, CA: IBM Santa Teresa.

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

*John Fox is a professor in the Department of Sociology and the Department of Mathematics and Statistics at York University, Toronto, where he also serves as coordinator of the Statistical Consulting Service in the Institute for Social Research. He has a continuing interest in social statistics and statistical computing, and his current research includes further development of APL2STAT. His most recent book is the monograph* Regression Diagnostics *(Sage, 1991).*

*Michael Friendly is associate professor of psychology and associate coordinator of the Statistical Consulting Service at York University. His research interests include cognition and statistical graphics. He is the author of* The SAS System for Statistical Graphics *and "Mosaic Displays for Multi-way Contingency Tables"* (JASA, *1994).*