The Interactive Data-analysis Language Reference Manual

BY **Ronald M. Kaplan, B. A. Sheil** AND **Eliot R. Smith**

DECEMBER 1978

SSL-1978-4

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

ABSTRACT

This document describes a computer system for the analysis of data from experiments and observational studies of the type carried out by social scientists. Initially developed at Harvard University, the version described here is an Interlisp implementation carried out at the Palo Alto Research Center of the Xerox Corporation.

Unlike most data analysis systems, IDL is constructed around a set of "basis operators" for data analysis, and emphasizes their combination to mould operators to match the user’s tasks, rather than the provision of pre-programmed ones. This implementation is a test bed both for this philosophy of application programming and also for the specific decomposition of linear statistics which it contains. Comments and interested users are most welcome.

KEY WORDS AND PHRASES

Data analysis, linear statistics, Interlisp, interactive systems, application programming.

CAVEAT

Both this manual and the system which it describes are, and will remain for the indefinite future, in a state of development. Although we both solicit and welcome users and will support them with consultation, bug fixes, and new features to the extent permitted by our other activities, we can guarantee neither the performance of the system, the validity of its results, nor the accuracy of its documentation.

BECOMING A USER

Anyone wishing to become a user of IDL is advised to contact one of the authors, so that their name can be added to the distribution list for announcements of new releases, facilities, and documentation. Specific instructions for accessing IDL on the Maxc2 time-sharing system can be found in Appendix A.

CONTENTS

ACKNOWLEDGEMENTS

IDL was initially designed by Beau Sheil and Eliot Smith under the auspices of the Computation Facility of the Center for Behavioral Sciences at Harvard University during 1974. That development was largely supported by a grant to the Center from the Harvard University Faculty of Arts and Sciences, and by computing resources made available by the Harvard University Center for Research in Computing Technology. The work profited greatly from the enthusiastic support and technical guidance of Dr. Klaus Allerbeck, at that time Director of the Computation Facility, and many others who provided advice and encouragement, including Karen Lease, Thomas Cheatham, Ronald Kaplan, David Kenny, and Joanne Martin.

The design was revised and extended by Ron Kaplan and Beau Sheil at Xerox Palo Alto Research Center, where the Interlisp implementation was carried out. Our thanks to Jan Derksen, Warren Teitelman, and Larry Masinter for all their help and assistance, and to our respective Laboratories for supporting this effort.

0. Introduction

The Interactive Data-analysis Language (IDL) is a programming language designed for the analysis of social science data. IDL is radically different from most previous computer systems for social science data analysis, as it is not based on the statistical "package" or subroutine concept, but on the concepts employed in the design of modern programming languages. Thus, IDL provides the user with powerful tools for the manipulation of the type of data most commonly used in social science research, rectangular numeric arrays, and the ability to combine these tools to yield a wide range of statistical analyses. The sophisticated computer user will find similarities between IDL and Iverson's APL, at least in the facilities for handling arrays, but IDL goes well beyond APL in providing capabilities especially designed for practical data analysis, such as the handling of labels (including the automatic generation of new labels by system routines) and missing data, and the ability to analyze very large data sets.

IDL is unlike most statistical systems in that its effective use requires that the user master a fair amount of material that does not, at first sight, seem at all relevant. As the reader might question our asking him to wade through five chapters of manual before the first explicitly statistical operation appears, the next section attempts to motivate this with a discussion of the advantages of this approach over the conventional one.

0.1 The philosophy of IDL

The concept of a statistical package system (like DATATEXT or SPSS) dates back to the first half of this century. At that time, data analysts were tightly constrained by the computational demands of their analyses. Certain techniques, such as factor analysis, large scale regressions and analyses of variance, were extremely expensive to carry out by hand and thus were rarely used even if the data were suitable. The primary motivation of the designers of the package systems was to relieve users from the computational load of a small number of common analyses.

This preoccupation with a small set of computationally intractable analyses led the package designers to a "shopping list" approach to data analysis: they provided a list of independent subprograms that consume data, print the results, and exit. Unfortunately for the user, if the analysis that he wants is not on the list, or if it is present but not quite right in some detail, the program is of no use to him. Thus, package designers increase the length of the list and the options available for each item in order to reduce the chances that a user's demands will not be met. This, in turn, makes the packages bigger and therefore more costly both to learn and to use. For the occasional user, the cost of switching packages is considerable, and although the consequent reluctance to do so is deplored by the sophisticated as "package dependence", it is a completely natural reaction under the circumstances.

Freed from many of the constraints of the pre-computer decades, users have become much more demanding in the sophistication and complexity of the analyses that they wish to apply to their data so there is simply no prospect that the packages' range can be increased enough to satisfy the user community. Instead, each new addition merely compounds complexity without any real increase in generality.

The approach being followed by IDL is to give the user a powerful set of statistical "building blocks", which provide basic analytic capabilities, and a mechanism for combining these basic operations together in new ways to extend those capabilities. These two components are complementary. The choice of the basic analytic routines is influenced by how useful they will be in defining further analyses. Thus, a routine which computes a multiple regression and prints the result is not a useful building block, as nothing else can be done with it. One that removes variance components from a covariation matrix and returns the new matrix as a result would be very useful, as many common statistics could be defined with it. It is this two part approach, the interaction of the basic tools and the composition mechanism, which constitutes the essential difference between IDL and a conventional system.

The way one uses IDL reflects this difference. With a package system, a closed set of commands is available; with IDL one can construct any command that is needed, from the elementary to the complex and specialized. However, both the base and the composition mechanisms must be understood before the benefits of the language are reaped, and this is our reason for asking your perseverance through so much preliminary material. Although this may be tedious, console yourself with the thought that it is like learning to drive a car. If your object is to get somewhere now, it is quicker to take a bus (provided that the bus goes where you want to go). In the long run, however, you see more places and get there faster if you can drive: the adventuresome can strike off into new terrain, while the timid will find maps to certain commonly visited places in Chapter 8.

0.2 The relation between IDL and LISP

IDL actually consists of a set of programs embedded in a large programming system, Interlisp-10 (hereafter, "Lisp"). These programs (or *functions*, to use the Lisp terminology) are the basic building blocks mentioned above. Each function has a descriptive name, such as PLUS, ANOVA or MOMENTS, and can be *applied* to some data objects by specifying its name and the appropriate data in an *expression* which is given to the Lisp system to be *evaluated*. Thus, the expression

```
(PLUS 3 2)
```

would cause the function PLUS to be applied to the numbers 3 and 2 to produce the value 5. In general, the first item in such an expression denotes the function to be applied, and the remaining items denote the arguments. The arguments themselves may also be expressions. If A represents a set of numbers, then

```
(MOMENTS (RANK A))
```

will apply the RANK function to A to compute the ranks of the numbers in A, and then apply MOMENTS *to those ranks* to compute their count, mean, and variance.

This simple notion of composition is the basic method by which data analysis operations are constructed from the statistical building blocks provided by IDL. Lisp provides several other ways in which functions and expressions may be composed to carry out new tasks. For example, sometimes it is desirable for a complicated expression to be treated as a function in its own right, so that it can be applied as a unit to the result of other computations. Thus, the expression

```
(QUOTIENT X (LOG X))
```

will scale the value X by its logarithm. The result of taking, say, the square root of X may be similarly transformed with the expression

```
(QUOTIENT (SQRT X) (LOG (SQRT X)))
```

but this entails both conceptual and computational problems. The SQRT function has been distributed throughout the original scaling transformation, so it is no longer clear that two conceptually distinct operations are being composed. Further, the expression is computationally inefficient because it stipulates that the SQRT computation is to be evaluated twice, even though both evaluations will yield the same result.

These problems disappear if the logarithm scaling is made into a function, since this function can then simply be applied to a SQRT argument expression. Functions in Lisp are constructed by enclosing the expressions defining the computations they are to perform in an "expression" beginning with the key-word LAMBDA. The LAMBDA is followed by a list of names which the enclosed expressions may use to refer to the values to which the function is applied. Thus, the scaling operation can be represented as the function

```
(LAMBDA (Y) (QUOTIENT Y (LOG Y)))
```

which can appear instead of a function name in the first position of an expression whose second element specifies the argument to which it is to be applied. The argument expression will be evaluated, and its value will become the value of the variable Y given in the LAMBDA-expression. When the body of the LAMBDA is evaluated, that value will become the numerator of the QUOTIENT and its LOGARITHM will be the denominator. The square root composition is then

```
((LAMBDA (Y) (QUOTIENT Y (LOG Y))) (SQRT X))
```

Notice that the scaling operation appears as a conceptual unit, and that the SQRT occurs only once. The scaling may be composed with other transformations by changing only the expression to which it is applied. For example,

```
((LAMBDA (Y) (QUOTIENT Y (LOG Y))) (SIN X))
```

applies logarithm scaling to the result of the trigonometric SIN of X.

If the computations encapsulated in a LAMBDA-expression are to be used repeatedly, it is convenient to associate a name with that function. The Lisp function PUTD (for *put d*efinition) makes the name-function association. Thus, after evaluating the expression

```
(PUTD (QUOTE LOGSCALE) (QUOTE (LAMBDA (Y) (QUOTIENT Y (LOG Y)))))
```

the computations described above may be specified by

```
(LOGSCALE (SQRT X))  and  (LOGSCALE (SIN X))
```

The QUOTEs in the PUTD expression are used to indicate that PUTD is to operate on the *actual expressions themselves*, rather than the result of evaluating those expressions. In other words, QUOTE is used to suppress evaluation of an expression. Its effect here is that the name LOGSCALE, *not* the value of the Lisp variable LOGSCALE, is associated with the definition which is the LAMBDA-expression itself, *not* the result of evaluating the LAMBDA-expression.

The function-defining and function-naming facilities permit elaborate computations to be packaged together and invoked by a single user-chosen name. In this way, the user can augment the IDL building blocks to construct operations specially suited for the particular kinds of analyses he needs to perform.

These and many other Lisp facilities for creating and manipulating functions and applying them to arguments have been maintained in IDL, and most IDL users would profit from a knowledge of them. The basic Lisp concepts necessary to understand the rest of this manual, and the minimal information necessary to start using IDL on a PDP-10/Tenex system can be found in Appendix A. Descriptions of the more sophisticated features of Interlisp are not contained in this manual, in the interests of keeping it concise. There is a separate manual for Interlisp (Teitelman, 1978), and any amount of time invested in reading it would be profitable, though it is perhaps a bit overwhelming for the casual user.

Programmers may wonder why Lisp by itself is an inadequate environment for data analysis. Others may wonder this about some other language, such as APL. The reason is that many facilities that must be provided for the data analyst are either not present or very difficult to implement in the confines of a conventional programming language. For example, it would be very difficult to redefine the arithmetic operations of APL to handle missing data.

0.3 The rest of this manual

The rest of the manual divides into five parts: the basic structure of the system, functions for manipulating the common data objects, functions for basic analyses, a guide to composing common analyses from the basic ones, and technical details. Chapters 1 and 2 cover the basic system and its handling of data. Chapters 3 and 4 cover the basic tools for manipulating data in array form. Both of these sections are essential reading. Chapters 5 and 6 cover the basic routines for data compression and analysis; knowledge of the routines one plans to employ is essential but complete knowledge is not necessary. Chapter 8 outlines the use of IDL for data analysis, and provides extended discussions of certain common cases which may not be obvious from the rest of the manual. The remaining sections cover technical details: Chapter 7 discusses ways in which data can be input to IDL, and the Appendices give a commented protocol from an IDL session, detailed definitions of the operation of the system in a format designed for quick reference (as opposed to the expository descriptions in the chapters), and miscellaneous other information. Finally, there is a glossary of common technical terms used throughout the manual.

Should any aspect of the manual or the IDL system prove awkward or incomprehensible, please feel free to complain and suggest improvements. IDL is a new system, and should continue to evolve as we find out more about the needs of its users. Please participate.

1. Data Representation

The basic element of data analysis is the *data item*, a single observation on a single subject. In IDL, a data item is a number that may be used to represent a measure, category membership, or any other attribute of the subject. IDL places no constraint or interpretation on the data items with which it deals. It is the responsibility of the data analyst to ensure that the operations performed on a data item are consistent with his interpretation of it.

A central property of data is its organization into aggregates. For example, a survey consists of a set of observations repeated for many different subjects. A central concern of the data analyst is the structure of these aggregates. Many data analysis tasks use data with "rectangular" structure, that is, data organized into vectors, tables, matrices or some similar structure. Even when the data is not strictly of this form (e.g., in a design in which one variable is nested within another), it is quite often conveniently represented by a slight variation on a rectangular structure. For this reason, IDL provides both a representation for such structures, the IDL *array*, and a large set of functions with which to manipulate them. Most of the later chapters of this manual are concerned with those functions. In this chapter, however, we will examine the structure itself, and its intrinsic properties.

1.1 Array structure

An IDL array is a collection of numeric data items with the following properties:

1. It has a fixed number of *dimensions*. For example, a vector has only one dimension, a matrix has two, a table of SEX by EDUCATION by VOTE has three, and so on. IDL arrays may have as high a dimensionality as the user desires.

2. It has a fixed number of *levels* on each dimension. This number can vary from dimension to dimension. The SEX by EDUCATION by VOTE table mentioned before might have two levels of SEX, four of EDUCATION, and three of VOTE, resulting in a total of 24 cells. The vector whose elements are the number of levels on each dimension of an array is referred to as the *shape* of the array. In the case of our table, we say that it has shape [2,4,3]. Note that the shape of an array is itself an array (a vector of length equal to the number of dimensions of the array). *Its* shape (i.e., the shape of the shape of the array) will be a vector with one element which is the dimensionality of the array.

3. The data items in it can be represented using one of two different representations for numbers (either *integer* or *floating*). For various technical reasons, all elements of an array must be of the same type, i.e., if one element of an array is floating, all must be floating. In addition, particular data items can be marked as "missing".

4. It can be labelled. This is a very important feature for the user, as the labels contain his view of what the array actually means. Labels are Lisp atoms. They may be attached to an array, maintained, removed, or changed by the user at will.

5. If it is a matrix, it can be stored in either of two formats. Some matrices are stored so that only the elements below the diagonal are present, and those above are assumed to be the same as their "mirror image" across the diagonal. Such arrays, e.g., correlation matrices, are called *symmetric* (as opposed to the usual storage format, which is *full*) and allow considerable savings in space. Although the user can control the storage format of a matrix, he will usually be content to let IDL choose it.

1.2 Terminology

At this point, we introduce some terminology that provides a concise way of stating the behavior of functions described later.

A *scalar* is a number of any form in any structure that is used to represent anything. For example, 34, 2, 7.3, and 1.12E12 are all scalars. The shape of a scalar is the empty vector.

A *vector* is an array with only one dimension. If it is necessary to specify its length, we will refer to it as a P-vector where P is an integer giving the length. When the elements of a small vector must be indicated in the text, we will enclose them in square-brackets. Thus, the 2-vector with first element 4 and second element 5 will be represented as [4 5].

A *matrix* is an array with only two dimensions. If it is necessary to specify its size, we will refer to it as a [P,Q]-matrix, where P and Q are integers giving the number of rows and columns respectively.

An *array* may be assumed to be without constraints on its shape. Such constraints are noted by specifying that an object is an N-array (indicating that it has N dimensions) or an [P,Q,...]-array where [P,Q,...] is the shape of the array.

A particular element of an N-array is referenced by a *subscript*, a sequence of N integers indicating its level on each of the N dimensions of the array.

There are occasions when we must enumerate for consideration all the elements of an array, one at a time. Most commonly, the elements will be enumerated in *row-major* order, which means that the last subscript varies fastest. That is, the elements of a [P,Q,...X,Y]-array will be considered in the order [1,1,...1,1], ... [1,1,...1,Y], [1,1,...2,1], ... [P,Q,...X,Y], etc.

1.3 Labels

An array may be labelled in a variety of ways. The first type of labelling an array may have is a *title*, a string which describes the array. A typical title might be: "Sociological Variables on 75 Subjects". Titles are completely arbitrary, although IDL will generate titles for newly created objects in a systematic way. Titles serve merely as comments on an array: IDL will print them, but otherwise pays no attention to them.

The title is associated with an array as a whole, but labels may also be associated with each of its dimensions. For each dimension, a whole block of label information may be present. This can include a *dimension label* for the dimension itself and a *selector label* (so called because it may be used to select sections out of the object) for each level of that dimension (e.g., row and column labels for a matrix).

Finally, labels may be associated with the values in the cells of an array. *Value labels* are used to indicate the meaning of numeric codes given to categorical variables. Thus, in a subjects by variables data matrix, one could attach value labels to a variable like SEX (e.g., the value 1 might be labelled MALE, and 2 FEMALE) to indicate the coding used. Each value label is associated with a numeric code, forming a pair, and the set of such pairs for a variable is referred to as its *codebook*. Value labels can only be attached to one dimension of an array (otherwise each cell would be in the scope of more than one set of value labels, making it impossible to decide which to use), and, even then, will only make sense for certain levels of that dimension. For example, one might have value labelled category variables such as SEX, but one would hardly use such labels for a continuous variable like AGE.

All labelling information is optional. If present, and if the user has requested label printout, the labels are printed whenever an array is printed, making it more easily comprehensible. For example, a simple data array, with two variables on three subjects, might print as:

```
    A Random Matrix
      Variable
 Subject          SEX       AGE
        1        MALE        24
        2           3        31
        3      FEMALE        28
```

Here, "A Random Matrix" is the title of the array, "Variable" is the dimension label for dimension two (the columns), while "Subject" is the label for dimension one (the rows). "SEX" is the selector label for the first column; "AGE" the selector for the second column. (More precisely, "SEX" is the selector for the first level on the second dimension). The first dimension has no selector labels, so integers are printed. Value labels are attached to the Variable dimension, and the code for SEX described above has been used. The value 3 does not have an associated code label for the variable SEX, so the numeric value appears in the table. This is sometimes useful in detecting "wild scores".

1.4 Missing data

With the introduction of aggregates of data comes the possibility of incomplete aggregates. These are troublesome as a "marker" or code for missing data must be stored within the aggregate, and must both be easily distinguished from real data yet act as if it *were* real data (so that one doesn't have to be perennially checking for it). IDL uses the value NIL as its missing data code. In most ways, NIL behaves like any other data value and may be stored into and retrieved from any cell of an array. However, as they have a data analytic interpretation as "undefined", missing data codes are treated specially by many IDL functions. For example, most arithmetic functions are defined so that, if given NIL as an argument, they return NIL (rather than producing an error). Thus, one may use elements from an array in arithmetic calculations without first having to check whether any of them are missing. However, if it is desired to treat missing cases specially, they can easily be detected, since the missing data code, unlike any genuine arithmetic value, compares equal (via the Lisp function EQP) to NIL.

1.5 Large arrays [not yet implemented]

Many statistical analyses are based on very large data sets, often ones that are far too large to fit into main memory. IDL allows the user to store arbitrarily large arrays in files and to access these arrays as though they were in main memory. In this way, arrays much larger than the available main memory can be manipulated. The ability to store and access file resident arrays will mainly be useful for arrays of raw data, as they are typically much larger than other arrays. Usually, the user will "compress" large data arrays down into smaller arrays that are suitable for further analysis. Nearly all commonly used statistical methods are based on such compressions and IDL offers a comprehensive set of compression functions which are documented in Chapter 5.

1.6 Function extension

IDL arrays may have an arbitrary number of dimensions, yet many operations used in data analysis are defined either for scalars or for arrays of a particular dimensionality. As examples, the square root function expects a scalar argument and returns its square root (a scalar), while the matrix-inversion function computes a matrix which is the inverse of its matrix argument. Such expectations will be violated by an argument with too few or too many dimensions. Every function decides for itself what it will do with arguments of too few dimensions. Thus, it makes no sense to invert a vector or scalar, so INVERT simply announces an error when it receives such an object. On the other hand, it is reasonable for matrix-multiplication to treat vectors as row or column matrices.

Arguments with too many dimensions are handled uniformly by a general *function extension* mechanism. On each call to an extended function, the extension mechanism inspects each argument and determines whether it is of greater dimensionality than the function is programmed to process. If so, the overly large object is broken down into smaller slices each of the correct dimensionality, and the function is applied to each of these in turn. The results of these separate applications are collected by the extension mechanism and amalgamated into a single array.

The scalar-to-scalar function SQRT is thus applied elementwise if it receives an array argument, so that the "square root" of the vector [9 16 25 36] is the vector of square roots [3. 4. 5. 6.]. If INVERT is given a [2,3,3]-array, then it will invert the two embedded [3,3]-matrices separately. For example, if the input array represented two within-cell covariation matrices of a one-way classification, the result would be a [2,3,3]-array containing the two within-cell matrix inverses.

A fuller description of the extension mechanism is provided in section 2.3. Ways in which the user may control the break-down of arguments are discussed, as are techniques for *extending* (i.e., embedding the extension mechanism in) functions defined by the user. We note here that, powerful as this mechanism is, it has one systematic limitation: the shape of the values of the extended function for each argument slice must be the same. This is necessary so that the partial results can be formed into an array, which must be rectangular. Therefore, if the value shape for one argument slice differs from the shape for other slices, the extension mechanism will cause an error and stop the computation.

Extension would fail, for example, if a function expects a scalar and returns a vector whose length is determined by that scalar (e.g., DEAL). Giving such a function the vector [2 4 7] as input would require the construction of a matrix whose first row was two elements long, whose second row was four elements long, and whose third row was seven elements long. Such an object is not rectangular and cannot be an array, so the extension will not be permitted.

All built-in IDL functions are extended unless it is stated otherwise in their description. The brief descriptions of each function found in Appendix D contain the expected dimensionality of each argument for each function.

1.7 Conversion from other representations

Many functions are described in this manual as requiring array arguments with such-and-such a property. If they were all to insist on their requirements being strictly met, IDL would be quite awkward to use. For example, a scalar is formally quite different from a 1-vector which is different from a [1,1]-array. However, all three contain exactly one element, and IDL will accept any of them in situations requiring a single data item.

Similarly, a Lisp list is formally quite distinct from an IDL array. List structures are, however, very easy to type in as the arguments to functions, and Lisp provides many tools for constructing and manipulating them. For convenience, every IDL function will attempt to convert a list structure argument into an IDL array, so that arrays and appropriate lists may be freely intermixed. The basic conversion rule is: a list of *k* N-dimensional IDL arrays all with the same shape will be converted to an N+1-dimensional array with *k* levels on its first dimension. The values in the first array in the list will be in the first level of the larger object, the second array goes into the second level, etc.

Thus if A is the 4-vector [1 3 5 7] and B is [2 4 6 8], the value of the expression

```
(LIST A B)
```

will be treated by functions expecting an array argument as the [2,4]-matrix

```
1 3 5 7
2 4 6 8
```

(The Lisp function LIST constructs a list whose elements are the values of its argument expressions.)

Since scalars behave as 0-dimensional arrays, and since the rule applies recursively, the same effect may be achieved by

```
(LIST '(1 3 5 7) B )   or
(LIST '(1 3 5 7) '(2 4 6 8) )   or, more compactly,
'( (1 3 5 7) (2 4 6 8) )
```

The last line makes use of the fact that if all the arguments to LIST are quoted, then all the internal quotes can be eliminated and the LIST itself can be replaced by QUOTE (or ', the CLISP equivalent).

The conversion will result in an error if the listed objects cannot themselves be converted to arrays of the appropriate dimensionality and shape.

2. System Facilities

This chapter describes the primitives IDL provides for operating on labelled, rectangular aggregates. These include functions for accessing and modifying selected array components, the extension mechanism that determines how most functions handle arrays with too many dimensions, and facilities for printing arrays in a readable format.

2.1 Selection

Selection is the operation for accessing individual elements, whole subsections, or the labelling information of an array. It is effected with the function AT, which is associated with the CLISP operator @.

at[a;sltr] where **a** is an array and **sltr** is a *selector* describing which components of **a** are to be selected, returns that selection.

Selectors may take several forms depending on the information that is to be extracted. The various possibilities will be illustrated in the context of a variable A bound to the matrix

Another Random Matrix

	Variable		
Subject	SEX	AGE	VOTE
1	1	24	2
2	3	31	1
3	2	28	3
4	1	25	2

2.1.1 Elements

The simplest selector is a list of integers, one for each dimension of A, specifying the level to be selected on each dimension of A. The result of this selection will be the element that lies at the intersection of these levels. For example, since A is a [4,3]-array, one can access the element of A which is at level 3 of the first dimension and level 2 of the second dimension (the scalar 28) with the expression

(AT A '(3 2)) or, using CLISP, A@'(3 2)

However, whole subarrays (rows, columns, planes, etc.) can be selected by using a vector, rather than a single integer, as the selector for a dimension. For example, one could select the first two elements of the first row of our [4,3]-array with

A@'(1 (1 2))

The list (1 2) is first converted into a vector as described in section 1.7. When applied as a selector, it will produce the 2-vector [1 24], the first element of which is A@'(1 1) and the second A@'(1 2). The size of such a multiple selection is the product of the number of elements selected on each dimension. An easy way to think of this process is to imagine the selected subscript values casting "shadows" down the rows and columns, etc. of the array. The result will be all elements that fall into exactly one "shadow" from every dimension.

The result of applying a vector selector to a dimension is formed by selecting the level corresponding to each of the vector's elements and packing these selections together. The same principle allows higher-dimensional arrays to be used as selectors, but in this case the result may have more dimensions than the original! Thus, if the selector on some dimension is a matrix, each element of the matrix is considered separately as a level selector for that dimension. The result of all these selections is formed by fitting them together into a two-dimensional plane which appears in the output in place of the selected dimension of the original. Thus, for M the matrix

```
1   2
3   1
```

the expression

```
A@(LIST '(1 2) M)
```

produces an object with a leading Subject dimension, but in place of the Variable dimension is a two-dimensional image of M:

```
Subject = 1
                 1        2
        1        1       24
        2        2        1
Subject = 2
                 1        2
        1        3       31
        2        1        3
```

Note that the column labels of A have been lost, because A's second dimension does not appear directly in the result and labelling information is attached by dimension. Since M is providing the structure, it is M's labels (in this case none) which are copied to the second and third dimensions of the result.

The list-structures in these examples are an easy way of typing in small selectors. However, each selector may be constructed by evaluating an expression that describes an arbitrary scalar or array computation. This technique can result in considerable economy of specification. For example, the vector generating function GENVEC (see section 4.4) produces vectors that are often useful as selectors:

```
A@(LIST (GENVEC 2 4) 3)
```

selects the second, third and fourth elements of the third column of A. (The Lisp function LIST is used instead of a quoted list so that the selector contains the *value* of the GENVEC expression, not the GENVEC expression itself.)

Two additional features in IDL's selection mechanism provide even greater conciseness of specification. First, although a subarray that includes all levels on a given dimension can be selected by listing those levels explicitly, e.g.

```
A@'(1 (1 2 3))
```

to select all elements of the first row of A, the same effect may be achieved by selecting with the distinguished symbol ALL:

```
A@'(1 ALL)
```

Second, if the selector list has fewer items than the dimensionality of the array, the selectors for the leading dimensions are defaulted to ALL. Thus,

```
A@'(1) is equivalent to A@'(ALL 1)
```

These facilities make selection a much more general operation than simply a way of selecting subsets. For example, one useful special case is where the selector for one dimension is a permutation (i.e., a reordering without addition or deletion) of the complete set of levels for that dimension. The result of such a selection is the corresponding permutation of the array. For example,

```
A@'((4 3 2 1) ALL)
```

produces an image of A with the order of the rows reversed! Note also that entries within a multiple selector may be repeated or arbitrarily reordered. Thus,

```
A@'((2 2) ALL)
```

is a matrix, both rows of which are the second row of the matrix A.

The levels to be selected may also be specified symbolically, by means of the corresponding level labels. A label appearing as a selector for a dimension is interpreted as the corresponding level index on that dimension. Thus, the label SEX can be used instead of the integer 1 to select the first column of A:

```
A@'(ALL SEX)
```

Similarly, when list-structure is converted into an array selector, labels appearing in the list are treated as level labels on the corresponding dimension, so that

```
A@'(ALL (2 SEX))
```

would select A's second and first columns. Note, however, that

```
A@'(SEX ALL)
```

could not be used to obtain the second row of A, because SEX is not a selector for A's first dimension. If this expression were evaluated, an error would occur.

Formally, a selector for array elements is a list of *n* items, where *n* is less than or equal to the dimensionality of the array being selected from. Each of these items may be:

a. a single integer or atom label, indicating the level to be selected on that dimension. Note that this dimension will be dropped from the result. For instance,

   ```
   A@'(1 ALL)
   ```

   is a 3-vector, not a [1,3]-matrix as

   ```
   A@'((1) ALL)
   ```

   would be.

b. a list-structure of integers or atoms that are selector labels for the corresponding dimension, indicating the levels to be selected on this dimension. This will be converted into an array and treated as in (c).

c. an array, indicating that the corresponding dimension of A is to be represented in the result by a set of *d* dimensions, where *d* is the dimensionality of the array used as selector. The elements on those dimensions of the result will be the result of applying the elements of the selector array as a selector to the corresponding dimension of A, and forming the results into an image of the selector array.

d. the value ALL indicating all levels for that dimension. (ALL is bound to itself, so it may be used interchangeably in quoted and non-quoted contexts.)

If *n* is less than the dimensionality of the array, ALL is assumed for the leading dimensions.

The general principle governing element selection is that the result is an array whose shape is e_1 !! e_2 !! e_3 !! ... e_N where !! indicates concatenation and e_i is the shape of the selector for the *i*th dimension. The behavior of the scalar and array selectors becomes clear when referred to this principle. The shape of a scalar is the empty vector, and when this is concatenated with other vectors it disappears. Thus, the result loses a dimension. The shape of an array is, in general, a vector of length equal to its dimensionality. When this is concatenated in the formula above, it lengthens the shape of the result, and thus increases the dimensionality.

2.1.2 Other array properties

In addition to being usable as selectors, the labels of an array are themselves accessible through selection. The appropriate selectors are constructed with the TITLE, LABEL and CODE functions. The selector produced by the TITLE function will access the title of an array, so that the value of

```
A@(TITLE)
```

is the string "Another Random Matrix".

The LABEL function is used to access the dimension and selector labels of an array. The label selected is determined by the number of arguments given to the function. A single argument to LABEL will select the corresponding dimension label. Thus,

```
A@(LABEL 1)
```

will return the atom Subject, the dimension label for the first dimension. In a similar fashion, one can select the selector labels by using LABEL with two arguments, a dimension and a level. Thus,

```
A@(LABEL 2 3)
```

would select the selector label for the third level of the second dimension, namely, VOTE.

The arguments to LABEL can be either integers or labels. The type of the last argument determines the result. If, as above, the last argument is an integer, then the corresponding label will be returned. If, on the other hand, the last argument is a label, the corresponding integer index is returned. For our example array above

```
A@(LABEL ’Subject)
```

would evaluate to one, the number of the dimension with label Subject. However,

```
A@(LABEL ’Subject 3)
```

would still evaluate to VOTE. The dimension label is used only to specify the dimension being accessed and does not affect the kind of value returned.

The value labels of an array are accessed in a similar manner using CODE. To illustrate how it works, let us assume that value labels are attached to the Variable dimension of A, and that the variable SEX is coded in the usual way. The matrix would thus print as

```
   Another Random Matrix
     Variable
Subject      SEX      AGE     VOTE
      1     MALE       24        2
      2        3       31        1
      3   FEMALE       28        3
      4     MALE       25        2
```

The function CODE can be called with zero, one, or two arguments. If called with none, it returns a selector that specifies the index of the dimension to which value labels are attached, so that

```
A@(CODE)
```

would evaluate to two. If called with one argument, that argument is interpreted as a selector for the value labelled dimension, and the result will access the codebook for that selector. A codebook is represented as a list of (code value) two-element lists, so the expression

```
A@(CODE ’SEX)
```

would evaluate to the list ((1 MALE) (2 FEMALE)). If CODE is given two arguments, then the value of the resulting selector depends on whether its second argument is a numeric value or a label. If the argument is a label, then the selector will cause AT to scan the codebook indicated by its first argument, and return the value associated with that label. Thus, in our example,

```
A@(CODE ’SEX ’FEMALE)
```

would evaluate to two. If the second argument is numeric, then the result will be the label paired with that value. Thus,

```
A@(CODE ’SEX 2)
```

would evaluate to FEMALE.

If there is no title, label, index, or code corresponding to a selector for those properties, the selection will return NIL.

The value of the function AT is always a "virtual" object, a window onto the array from which the selection is made. This means that any change made to the underlying array (via the function ASSIGN described below) will be visible through the selection. The only way to break this connection is to use the Lisp function COPY, which forms a new object from the components visible through the selection.

2.2 Assignment

Assignment is the operation of changing one of the components of an array. *Assignment is the only IDL function that causes a change in an existing value*. All other functions produce new values and do not have any "side effects" on their arguments. Rebinding of a variable is carried out with the Lisp function SETQ:

```
(SETQ A B)  or  A _ B
```

causes the value of B to become the new value of A and the old value of A to be lost. Aggregate objects such as IDL arrays raise the possibility of making partial changes instead of wholesale replacements. For example, some of the elements in an array may be changed while the organization of the aggregate (the shape, element type, storage format, etc.) is preserved. Such selective modifications are done with the ASSIGN function:

assign[target;source] where **target** is a selection (i.e., the value of a call to AT), and **source** is a value to be stored into **target**.

For example, (ASSIGN A@’(1 3) 7.6) will store the value 7.6 into A’s [1,3] cell, assuming, of course, that A is a matrix (the length of the subscript list implies A has at least two dimensions) which has a [1,3] cell. If the element type of A is INTEGER, the floating value 7.6 will be rounded to 8 before storing.

Whole subarrays may be modified with one assignment. For example,

```
(ASSIGN A@’((1 2 3) ALL)  0)
```

will set the first three rows of the matrix A to zero. The right hand argument of the assignment need not be a scalar. Thus, if A and B are both [4,4]-arrays, one could assign the second row of B into the third column of A with

```
(ASSIGN A@’(ALL 3)  B@’(2 ALL))
```

Such assignments are done serially in row-major order (i.e., both the source and the target are enumerated with the last subscript varying fastest). This can lead to counter-intuitive results when a segment of an array is assigned into an overlapping segment of itself.

The labels of an array can be changed by assigning into the result of a TITLE, LABEL, or CODE selection. For example,

```
(ASSIGN A@(TITLE) "New title")
```

replaces A's title, leaving the rest of A unchanged. Most IDL functions compose titles for their resulting arrays that indicate the function and arguments that produced the array. The internal convention for abbreviating such titles is available to the user: ASSIGN accepts list-structure titles as being equivalent to function-argument strings. Thus,

```
(ASSIGN A@(TITLE) (LIST 'Percentages X))
```

has the same effect as

```
(ASSIGN A@(TITLE) (CONCAT "Percentages of " X@(TITLE)))
```

(The Lisp function CONCAT places its argument strings together end to end to form a new string.)

The only information obtainable with a label selector but not changeable simply by assigning into it is the index of a dimension or selector. Thus, LABEL selectors appearing on the left hand side of assignments are considered to specify the corresponding label, even if they would evaluate to integers elsewhere. For example,

```
(ASSIGN A@(LABEL 'Varaible) 'Variable)  and
(ASSIGN A@(LABEL 2) 'Variable)
```

are equivalent methods of changing the (misspelt) dimension label Varaible, whereas

```
(ASSIGN A@(LABEL 'Varaible) 2)
```

(or any other number) is in error.

One can change the dimension to which value labels are to be attached by assigning a dimension specification (either integer or atom) into A@(CODE). This must be done before using other CODE assignments to set other value labels. If some other dimension is value labelled at the time this assignment is done, the old value labels will be removed. Codebooks and individual codes or value labels may be changed in the obvious way. For example,

```
(ASSIGN A@(CODE 'SEX) '((1 MALE) (2 FEMALE)) )
```

attaches our familiar codebook for sex to the SEX selector of the value labelled dimension of A, removing any previous value labels for SEX.

Any label or code may be removed by simply assigning NIL to the appropriate selection. Value labels may be removed in larger chunks: assigning NIL as a codebook for a level on the value labelled dimension causes it to have no value labels; changing the value labelled dimension removes all existing value labels, as noted above.

2.3 Function extension

The extension mechanism, briefly introduced in section 1.6, is a system facility that intercepts array arguments of greater dimensionality than a function is programmed to process. Such arguments are decomposed into "slices" each of the dimensionality required by the function. The function is then applied to each slice in turn and the results from each application are assembled into a single array value.

Such a decomposition can take place in several different ways. A [2,3]-array, for example, may be regarded as a two by three array of scalars, as two 3-vectors, or as three 2-vectors. The way the arguments are decomposed affects the number of times the function is applied, the arguments that it receives for each application, and the shape and labels of the result.

This section discusses the extension mechanism in some detail, outlining the factors that determine how an argument will be regarded. The extension of single argument functions is considered first, followed by multi-argument functions, the devices by which the user can control the decomposition, and finally, the facilities by which the user may embed the extension mechanism in functions that he himself writes.

2.3.1 Single-argument functions

The function INVERT is a good example of an extended single argument function. It expects a matrix argument and returns a new matrix which is the inverse of its argument. If it is given a three-dimensional array, that array will be split into matrices along its leading dimension, these will be inverted separately, and the results will be fitted back together again in an image of the original argument. In a sense, the leading the dimension is withheld (or *kept*) from the inversion operation, and INVERT only sees objects that it is prepared to handle. Suppose A is the [2,3,3]-array

```
  4     9    12          90     3     6
 54     7     2           1    32    56
  5     7    32           7     3   567
```

The two levels on the first dimension are represented by the two panels, and the rows and columns in each panel correspond to A's second and third dimension. If the expression

```
(INVERT A)
```

is evaluated, A will be broken into slices along its first dimension, corresponding to the panels above, each slice will be inverted, and the results pasted back together to form the array

```
-.020    .019    .006          .011   -.001   -.000
 .164   -.006   -.061         -.000    .032   -.003
-.033   -.002    .044         -.000   -.000    .002
```

each panel of which is the inverse of the corresponding panel of A above.

The general rules for a single-argument function F expecting an array of *e* dimensions and given an array of *g* dimensions are as follows:

> If *g* is less than or equal to *e*, then F is simply applied once to its argument, and the resulting value is the value of the (vacuous) extension.
>
> Otherwise, *g* is greater than *e*, and the first *g e* dimensions of the argument are considered to be in excess. The function will be invoked a number of times equal to the number of cells represented by the excess dimensions (i.e., the product of the numbers of levels for those dimensions). The argument seen on each invocation will be a slice of the argument formed by selecting all levels on the non-excess, trailing dimensions and one level for each of the leading dimensions.
>
> The result of the extension is constructed from the values returned by the repeated calls on the extended function. These values are combined into an array whose leading dimensions have the extents and labels of the argument's excess dimensions. The extents for the trailing dimensions come from the common shape of the objects produced by each call. Their labels are taken from the value returned on the last call. Thus the shape of the result is the first *g e* elements of the shape of the argument, concatenated with the common shape of the values. The elements of the array are obtained by placing the values as trailing subsections in the larger array.

As mentioned in section 1.6, the extension will fail if the value shape for one of the argument slices differs from the shape for other slices. This requirement is necessary so that the partial results can be formed into an IDL array, which must be rectangular.

For the INVERT example, the original argument is a [2,3,3]-array and the function expects a 2-array, so the two-level first dimension is in excess. The function will be called twice, receiving X@'(1 ALL ALL) the first time and X@'(2 ALL ALL) the second time. If A were a [4,2,3,3]-array, there would be two excess dimensions and the inversion would be done eight (= 4 x 2) times. The slices would be A@'(1 1 ALL ALL), A@'(1 2 ALL ALL), A@'(2 1 ALL ALL), etc. The final value would be another [4,2,3,3]-array.

2.3.2 Multiple argument functions

If F has multiple arguments, then both the dimensionality of the array given and the dimensionality expected of it must be considered for each argument. Let g_i be the dimensionality of the *i*th argument given to a function which expects its *i*th argument to have dimensionality e_i. The case of vacuous extension is the same as before: If, for each *i*, g_i is less than or equal to e_i, then all the arguments are passed directly to the function, it is invoked once, and the single value is returned.

The situation is a little more complicated if there are excess dimensions but the number of them is the same for all arguments, i.e., g_i e_i is the same positive number for all *i.* As in the single argument case, the leading dimensions are considered to be in excess. The left-most argument is taken to be the *controlling* argument of the extension, and to a certain degree the extension mechanism acts as if F were a function of only that argument. As above, the number of times that F is invoked is the product of the number of levels for the excess dimensions. It will receive a different slice of that argument on each invocation, and leading dimensions of the extension result will have the levels and labels of the controlling argument's excess dimensions. The other arguments are sliced up in parallel with the controlling argument, so the function will receive a different slice of each argument on each invocation. As before, the value of each invocation must have the same shape so that the final result is rectangular. There is an additional conformability requirement that the number of levels for the leading dimensions be the same for all arguments; otherwise, the parallel slicing is ill-defined.

As an example, suppose F is a three argument function expecting 3-arrays, matrices, and vectors. If it is applied to a [3,2,4,5,6]-array A, a [3,2,2,2]-array B, and a [3,2,7] array C, the decomposition will proceed in the following way: The number of excess dimensions for each argument is 2 (= 5 3 = 4 2 = 3 1), and the excess dimension levels have the same [3,2] extents. F will therefore be applied six times, receiving different argument slices on each application. The first time it will be given

```
A@'(1 1 ALL ALL ALL)
B@'(1 1 ALL ALL)
C@'(1 1 ALL)
```

the second time

```
A@'(1 2 ALL ALL ALL)
B@'(1 2 ALL ALL)
C@'(1 2 ALL)
```

the third time

```
A@'(2 1 ALL ALL ALL)
B@'(2 1 ALL ALL)
C@'(2 1 ALL)
```

and so on. The labels of A will be used for the leading dimensions of the result, since it is left-most and thus the controlling argument. Note that the leading subscripts for the selections are enumerated in parallel. The extension would fail if, for example, B were a [4,2,2,2]-array, as there would be no slices of A and C corresponding to the [4,--] slices of B.

In the most general situation, the number of excess dimensions is not the same for all arguments. The one with the greatest number of excess dimensions is taken as the controlling argument (the left-most one if there is a tie), and again, it determines the number of times the extended function will be called, the decomposition of arguments for each call, and the shape and labels of the result. The other arguments are "expanded" so that their excesses match the controlling argument's; the computation then proceeds as for the equal-excess case.

An argument is expanded by replication. Suppose that C in the example above were a [3,7]-matrix instead of a [3,2,7]-array. A is still the controlling argument, and it has a [3,2] excess. The extension will thus replicate C twice, treating it as the needed [3,2,7]-array. The same 7-vector will be given to F no matter what second-dimension subscript is involved in slicing A and B. Thus on one invocation F will receive A@'(1 1 ALL ALL ALL), B@'(1 1 ALL ALL), and C@'(1 ALL). That same slice of C will be given in the call involving A@'(1 2 ALL ALL ALL) and B@'(1 2 ALL ALL). The general requirement for conformability among arguments is that the left-most excess dimensions (which are themselves leading dimensions) must have the same extents for all arguments.

To take a concrete example, let A be the [4,3]-matrix introduced in section 2.1:

```
          Another Random Matrix
            Variable
        Subject      SEX       AGE     VOTE
              1        1        24        2
              2        3        31        1
              3        2        28        3
              4        1        25        2
```

The function DIFFERENCE expects both of its arguments to be scalars ($e_i = 0$) and returns their arithmetic difference. For the expression

```
(DIFFERENCE 50 A)
```

A is the controlling argument, since it has two excess dimensions and the scalar 50 has none. The scalar will be replicated to form the [4,3] array:

```
50    50    50
50    50    50
50    50    50
50    50    50
```

and the elementwise subtraction will be performed. The values are scalars with no dimensionality, so the result of the extension will simply be an image of A, complete with its labels:

```
          Difference of 50 and
          Another Random Matrix
            Variable
        Subject      SEX       AGE      VOTE
              1       49        26        48
              2       47        19        49
              3       48        22        47
              4       49        25        48
```

The extension mechanism would also work if the first argument were a 4-vector:

```
(DIFFERENCE '(2 4 6 8) A)
```

The vector has one excess dimension, so A is still the controlling argument. The vector is treated as the [4,3]-matrix

2	2	2
4	4	4
6	6	6
8	8	8

and the 12 subtractions are performed. The result, again taking labels from A, is

Difference of Array 15
and Another Random Matrix

Variable			
Subject	SEX	AGE	VOTE
1	1	22	0
2	1	27	3
3	4	22	3
4	7	17	6

On the other hand, the extension would fail if the first argument were a 3-vector: even though one dimension of A does have extent three, it is not the dimension being aligned with the vector so the conformability requirement would not be satisfied.

2.3.3 Affecting the decomposition (KEEP and LEAVE)

Whereas function extension automatically handles the case of subtracting each column of A from the same 4-vector, subtracting each row of A from the same 3-vector is less conveniently specified with this mechanism. The dimensions of A must be re-ordered so that the rows and columns are reversed, giving a [3,4]-matrix which the normal extension will be able to handle. Dimension permutations are computed by the function TRANSPOSE, described in Chapter 4. Thus

```
(DIFFERENCE '(1 3 5) (TRANSPOSE A))
```

produces the matrix

Difference of Array 17 and
Transpose of Another Random Matrix

Subject				
Variable	1	2	3	4
SEX	0	2	1	0
AGE	21	28	25	22
VOTE	3	4	2	3

Note that the desired subtractions have been performed but the result is tipped on its side, reflecting the shape of the *transposed* A, not the original. TRANSPOSE must also be applied to the output of the extension in order to get a more intuitive result.

Transpositions are a general way of affecting argument conformance and decomposition when the default behavior of the extension mechanism is inappropriate. By rearranging the leading dimensions, they alter which dimensions will be considered to be in excess and how they will be aligned. They do not change the number of excess dimensions; that is still determined by the argument expectations of the particular function involved.

There are occasions, however, when the arguments to a function must be decomposed into slices smaller than the function expects. This happens because an expectation defines the *maximum* number of dimensions that the function can handle, but many functions will produce reasonable results from smaller objects. Those results will never be computed for an over-sized object unless a function's permanent expectations can be overridden on a particular invocation.

As one example, the function ADJOIN, described in Chapter 4, produces vectors by joining its argument vectors together end to end:

(ADJOIN '(1 2) '(3 4))

is the 4-vector [1 2 3 4]. Though it expects vectors, ADJOIN accepts scalars and treats them as 1-vectors. This property provides great flexibility in the arrays that ADJOIN can produce. By the ordinary rules of extension, joining a 4-vector and the [4,3]-matrix A above gives a [4,7]-matrix with a copy of the vector appearing on the front of each row. Because the first dimension of A is the only one in excess, the 4-vector will be expanded to a [4,4]-matrix, and each call to ADJOIN will receive a row of that matrix and a row of A. Suppose that the extension mechanism could be told that on this invocation, ADJOIN's first argument is expected to be a scalar, so that the first dimension of the vector must also be withheld. Then both arguments would have a single excess dimension of extent 4, and no replication would be needed. Each element of the vector would be paired with the corresponding row of A, and the result would be a [4,4]-matrix with the vector added as a new column to the front of A.

The need for locally affecting a function's expectations is most clearly visible with functions that have *generic* arguments. These arguments are expected to be arrays, but there is no further constraint on their dimensionality. RPLUS is an example of a generic-argument function: it accepts an arbitrary array and produces the scalar sum of all the elements. Thus, if B is the matrix

1	3	4
2	7	5

the expression (RPLUS B) would evaluate to 22, the grand total of the cell values. This quantity can be used to determine the proportion of the total that is due to each cell:

(QUOTIENT B (RPLUS B))

QUOTIENT, like DIFFERENCE, is a scalar-to-scalar operation. Its arguments in this example are the matrix, and the scalar total. The matrix will be broken down into scalars, each one of which will be divided by the total, and the result will be an array with the same shape as B:

.045	.136	.182
.091	.318	.227

Suppose, however, that the cell proportions within columns are needed. In this case, each element of B must be divided not by the grand total, but by the total of all elements within its column. These numbers will result from collapsing just across the rows to produce a 3-vector of column totals (i.e., [3 10 9]). In other words, even though RPLUS will accept an arbitrary array, in certain situations it must behave as though it expected three 2-vectors (or two 3-vectors). The matrix B could then be sliced appropriately, the slices totaled, and the results pasted together to form the triple or pair of interest, which could then be given to the QUOTIENT function.

The behavior needed in examples like these is obtained by marking arrays so that certain dimensions will be withheld, or *kept*, no matter what the expectations might be of the function they are given to. The function KEEP provides this marking facility:

keep[a;dims...] where **a** is an array and **dims...** is an indefinite number of dimension specifications (either integers, dimension labels, or ALL) for **a**, produces a copy of **a** with **dims** added to (the front of) its kept dimensions. This will decompose in an extended function so that the kept dimensions will be preserved in the output.

If no dimensions are specified, KEEP returns a vector containing the indices of the kept dimensions of **a**.

Marking dimensions in this way causes the extension mechanism to treat an array as follows. First, the number of dimensions withheld will be the greater of the number determined by the argument-expectation discrepancy and the number of kept dimensions. Second, for the purposes of selecting the excess dimensions and aligning them across different arguments, the kept dimensions are considered to have been transposed to become the leading dimensions. In other words, all the kept dimensions will be withheld, and other dimensions may also be withheld in order to guarantee that the slices given to the extended function will not exceed its expectations. The "transposition" is carried out purely for alignment purposes, and the kept dimensions will appear in the result in exactly the same order that they appeared in the controlling argument, as before.

With this facility, the column-joining operation can be requested by

```
(ADJOIN (KEEP ’(1 2 3 4) 1) A)
```

The KEEP causes the first dimension to be withheld, so that ADJOIN behaves as if it expected a scalar first argument. KEEPing the generic argument of RPLUS yields the column proportions:

```
(QUOTIENT (KEEP B 2) (RPLUS (KEEP B 2)))
```

Since the second dimension is kept, the RPLUS in the denominator collapses across rows, giving [3 10 9]. The KEEP in the numerator moves the second dimension to the front, so that for purposes of argument conformance, the numerator is treated as a [3,2]-matrix. The 3 extent on the first excess dimension matches the 3 for the denominator excess. This is the only effect of the numerator KEEP, since both dimensions must be withheld anyway so that QUOTIENT receives only scalars. The row proportions for this array can be computed by the analogous expression:

```
(QUOTIENT (KEEP B 1) (RPLUS (KEEP B 1)))
```

It is occasionally necessary to undo the effect of keeping some dimensions of an array. The function LEAVE is the inverse of KEEP, producing a copy of an array without the marks that KEEP puts on.

leave[a;dims...] where **a** is an array and **dims...** is an indefinite number of dimension specifications (either integers, dimension labels, or ALL) for **a**, produces a copy of **a** with the specified dimensions not kept. LEAVE has no effect for dimensions that are specified in **dims** but not marked as kept in **a**.

2.3.4 User-written functions

The same mechanism that IDL uses to extend its own functions is available for user functions as well. This mechanism is embodied in the system functions EAPPLY and EAPPLY*, which apply a function to arguments decomposed according to the rules described above.

eapply[fn;expects;args] and eapply*[fn;expects;a_1,a_2,...] apply **fn** to the arguments **args** or **a_1,a_2,...** decomposed according to their keeps and the expected dimensionality as given in the list **expects**.

An **expects** entry of either SCALAR, VECTOR, MATRIX, or an integer indicates the expected dimensionality. The entry ARRAY indicates a generic argument whose decomposition will be controlled entirely by the actual argument’s keeps. The entry NIL indicates an argument that the extension mechanism will not examine at all.

If **fn** is a no-spread function, the expectations for its (arbitrarily many) trailing arguments can be specified by having the last **expects** entry be the atom ... in which case the preceding **expects** entry is used for all subsequent arguments.

The difference between EAPPLY and EAPPLY* is analogous to the difference between the Lisp functions APPLY and APPLY*. EAPPLY accepts a list of (already-evaluated) arguments for **fn**; EAPPLY* accepts a sequence of argument expressions which will be evaluated before the extension mechanism is invoked.

To simplify the task of defining new extended functions, IDL provides a new kind of function object, the ELAMBDA. The key-word ELAMBDA is used instead of LAMBDA in the function definition, and the expected dimensionality is specified with each argument. For example,

```
(ELAMBDA ((M MATRIX) (V VECTOR)) (ADJOIN V (MOMENTS M)))
```

is a function object that specifies that it expects its first argument to be a matrix and its second to be a vector. Such a function object may either be associated with some name, using DEFINEQ, e.g.,

```
[DEFINEQ (FOO (ELAMBDA ((M MATRIX) (V VECTOR)) (ADJOIN V (MOMENTS M]
```

or applied directly to a set of arguments. For example, for two arrays A1 and A2 of arbitrary dimensionality,

```
((ELAMBDA ((M MATRIX) (V VECTOR)) (ADJOIN V (MOMENTS M))) A1 A2)
```

is equivalent to

```
(EAPPLY* (FUNCTION (LAMBDA (M V) (ADJOIN V (MOMENTS M))))
         '(MATRIX VECTOR)
         A1 A2)
```

The technique of direct application of an ELAMBDA is an elegant way of forcing the repeated evaluation of some computation (see Chapter 8).

The function EXTEND provides a convenient way of extending some previously defined function. EXTEND operates by altering the function's body so that it calls EAPPLY* with the appropriate information.

extend[fn;expects] alters the body of **fn** to be eapply*[**fn;expects**;a_1,a_2,...]. As EXTEND can detect if a function has previously been extended, a function may be re-extended simply by calling EXTEND with a different **expects**. An **expects** of NIL clears the extension.

2.4 Printing

Lisp objects may be printed in both simple and complex (or *pretty*) modes, and there are two corresponding modes for IDL arrays. The simple representation of an array is called its Lisp "print-name". This is printed whenever an object is returned to the Lisp executive as the value of some expression. The print-names for IDL arrays are designed to uniquely identify the array (via a *serial number* assigned to each array as it is created) and to briefly summarize the properties that directly affect how subsequent operations will apply to it. These properties are its shape and its kept dimensions. Suppose A is a [4,3]-matrix with serial number 7. Its print-name is

```
[Array 7: 1=4 2=3]
```

The integers before the equal-signs are the dimension indices, and the numbers afterwards are the number of levels on the corresponding dimensions. If the second dimension of A is kept, then the print-name would be

```
[Array 7: 1=4 2=3; kept 2]
```

If the array has dimension labels, these are used instead of the dimension indices to provide a more informative description of the array. If A were a Subject by Variable matrix, its print-name would be

```
[Array 7: Subject=4 Variable=3; kept Variable]
```

If Lisp is asked to pretty-print an IDL array, then all the properties of the array are displayed, including the title, dimension and level labels, and the values in the cells of the array. The examples in preceding sections and subsequent chapters illustrate the format in which this information is presented. Printing of this sort will result from applying the Lisp function PRINTDEF to an array, or by invoking the IDL function PPA:

ppa[a;file] pretty-prints the array **a** on the primary output file, or **file** if specified. If **file** is not open, it is opened, the array is printed, and then **file** is closed. Value is **a**.

The Lisp primary output file is usually the user's terminal (denoted by the special file name T), so omitting a file specification will cause the array to be displayed directly to the user. IDL automatically maintains a "dribble" file (named IDL.TYPESCRIPT), a complete transcript of the user's commands and the system's responses throughout a session. This provides an enduring record of the arrays printed to the terminal and of the computational context in which those arrays are produced. The primary output file may be changed by the Lisp functions OUTFILE and OUTPUT.

The actual format in which the array appears may be controlled in a number of ways. Arrays may be printed with or without labels, and the precision and column-widths may be varied, depending on the values of the following global variables:

LABELPRINTFLAG

An integer which encodes how much of an array's labeling the PPA routine will print. Initially 4, it is encoded as follows

0	print no labels
1	print only the array title
2	print array title and dimension labels
3	print title, dimension, and selector labels
4	print title, dimension, selector, and value labels

When label printout is requested, missing labels will usually print as the corresponding integer index. An exception is that a totally unlabelled, small array (i.e., one that fits on a single panel of printout) will print without row and column indices, as they are both uninformative and apt to be confused for data values in this case. If there is no value label to correspond to a value found in an array, the actual score for which no label was found is printed under the prevailing rules for numeric printout. This is often a convenient way to find wild scores as the number stands out clearly in the value-labelled array.

linelength[n]

A Lisp function that indicates the number of character positions on a line. This determines the number of print positions that will be used before an array will be "folded" and subsequent columns printed as a separate section further down the page. If **n** is NIL, value is the current setting; otherwise, value is the previous setting.

PRECISION

A list of the form (L R) initially, (4 3). L and R specify, respectively, the number of digits to be printed on the left and right of the decimal point. The field width is always L+R+2, providing for a sign and the decimal point. Column selector labels and value labels will be truncated on the right so that they fit within the field.

ROWLABELWIDTH

An integer, initially 8, which specifies the number of columns to be used for printing the selector labels for rows.

3. Arithmetic and Mathematical Functions

Interlisp provides a large repertoire of arithmetic and mathematical functions. All of these are available to the IDL user, and many of them have been generalized to handle missing values and extended to expect scalars, so that they will operate elementwise on arbitrary IDL arrays. In addition, IDL defines some commonly used numeric constants and includes a set of arithmetic functions particularly suited to statistical applications.

3.1 Arithmetic operators

There are three sets of basic arithmetic operators in Lisp, those that operate on integers and produce integer values (with names like IPLUS, IMINUS, ITIMES), those that operate on floating point numbers and produce floating values (FPLUS, FMINUS, FTIMES, etc.), and those that operate on mixed integer and floating arguments and produce values whose type depends on their argument types (PLUS, MINUS, TIMES, etc.). In IDL, only the mixed arithmetic operators have been extended and may be used freely on IDL arrays. Though the integer and floating operators exist in the system, they will generate errors if applied to missing data items or multi-element arrays, and their use is discouraged. The CLISP operators +,*,/, have been bound to the mixed operators PLUS, TIMES, QUOTIENT, DIFFERENCE, and MINUS, so that complicated expressions may be written in abbreviated form. QUOTIENT has been modified so that if its divisor is zero, the result will be NIL.

The simple Lisp functions ABS, MAX, MIN and REMAINDER have also been extended. As well as being extended, MAX and MIN have both been modified so that they ignore NIL arguments. Thus, (MAX 1 NIL 2) is 1, not NIL.

The mixed arithmetic predicates have been modified in IDL so that they provide a reasonable result when applied to a missing data item. These functions have not been extended, however, since they return the truth values T and NIL, which cannot both be stored into an array. The predicates from Lisp are GREATERP, LESSP, EQP, and MINUSP. For convenience, the function PLUSP also exists in IDL and is T for any argument greater than zero. As mentioned in section 1.4, the missing data value compares EQP to NIL. As a somewhat arbitrary choice, necessitated by the binary nature of truth, it compares as if it is zero for the GREATERP and LESSP predicates, and is neither MINUSP nor PLUSP.

scalarp[s] is T if **s** is a scalar, NIL otherwise.

3.2 Constants

The following variables are initialized to some common mathematical constants. If changed by the user, their values are lost and must be reset.

E	is set to the base of the natural logarithms.
PI	is set to p.

3.3 Mathematical and trigonometric functions

The following scalar functions have been extended: SQRT (which returns NIL for negative arguments), LOG (which also returns NIL for negative arguments), ANTILOG, EXPT (with associated CLISP operator ^), and GCD. These are fully described in Section 13 of the Interlisp Reference Manual (Teitelman, 1978).

Interlisp's trigonometric functions (SIN, COS, TAN, ARCSIN, ARCCOS, and ARCTAN) have been extended to expect scalars. All these functions have an optional flag argument which if T indicates that all angle sizes are to be interpreted in radians. If the flag is NIL or omitted, arguments and values will be in degrees. Again, see Section 13 of the Interlisp Reference Manual.

3.4 Distribution functions

IDL provides several functions for evaluating the probability of values drawn from certain common statistical distributions.

fprob[f;dfnum;dfden] where **dfnum** and **dfden** are the degrees of freedom for the numerator and denominator, respectively, of the *F*-value **f**, returns the probability of such an *F*-value.

nprob[z] where **z** is a normal deviate, returns the probability of such a normal deviate.

tprob[x;df] where **df** is the degrees of freedom for *t*-value **x**, returns the probability of such a *t*-value.

At a future date, binomial and x^2 distribution functions may be added.

3.5 RAND and RANDN

Uniformly distributed random numbers can be obtained from the Interlisp RAND function, and random normal deviates from the function RANDN. Both are extended to expect scalars.

rand[lower;upper] returns a single pseudo-random number between **lower** and **upper** inclusive.

randn[mean;stdev] returns a single floating number, randomly sampled from the normal distribution with mean **mean** (default 0) and standard deviation **stdev** (default 1).

The Lisp function RANDSET may be used to initialize the random-number generator that both these functions depend on.

At a future date, generators for random deviates from other distributions may be added.

3.6 ROUND

The function ROUND rounds a number to a specified tolerance.

round[val;interval] for scalar **val** and **interval**, returns the nearest integral multiple of **interval** to **val**. **interval** defaults to one, i.e., rounding to the nearest integer.

Thus, (ROUND INCOMES 1000) will round a vector of dollar incomes to the nearest $1000. Note that rounding is to the *nearest* multiple, as opposed to truncation, so (ROUND 1.4 0.5) is 1.5, not 1.

ROUND is useful for converting a continuous measure into a GROUP classifier.

3.7 SAME

The Lisp comparison functions (EQP, etc.) return either T or NIL. Occasionally, it is useful to have a predicate function return an arithmetic value, so that the results of such a comparison can be assembled into an array. SAME provides this alternative to EQP.

same[a;b] for scalar **a** and **b**, returns one if **a** and **b** are EQP, else zero.

This is useful when applied to arrays rather than individual scalars, as the result can be manipulated to derive array properties. For example, reducing the result with TIMES or LOGAND will indicate whether two separate arrays have identical elements or which of their rows or columns are the same. Thus, (RTIMES (SAME X Y)) is one if X and Y are the same; (RTIMES (KEEP (SAME X Y) 2)) is a vector indicating which columns are the same.

3.8 TRANSLATE

TRANSLATE provides a table-driven translation facility that is particularly useful for recoding.

translate[s;table;default] where **s** is a scalar to be translated, **table** is a translation matrix, and **default** is an (optional) default value to be used if there is no translation for **s** in **table**. TRANSLATE provides three types of translation, depending on the number of columns in **table**:

One column: If **table** is an [N,1]-matrix, TRANSLATE returns the smallest row index *r* of **table** such that **table**@'(*r* 1) is **s**.

Two columns: If **table** is an [N,2]-matrix, TRANSLATE returns **table**@'(*r* 2) for the smallest *r* such that **table**@'(*r* 1) is **s**. That is, it returns the second element of the row whose first element matches **s**.

Three columns: If **table** is an [N,3]-matrix, TRANSLATE returns **table**@'(*r* 3) for the smallest *r* such that **table**@'(*r* 1) < **s** < **table**@'(*r* 2). That is, it returns the third element of the first row such that **s** is in the interval defined by its first two elements. NIL receives special treatment:

NIL	NIL	X	translates a NIL **s** to X
NIL	Y	X	Y not NIL, translates any **s** < Y to X
Y	NIL	X	Y not NIL, translates any **s** > Y to X

In all cases, if no match for **s** is found in the translation table, **default** is returned if it was given, otherwise **s**.

TRANSLATE is extended on all its arguments. Thus, one can translate an entire array, obtain a vector of one scalar translated according to several different translation tables, or both at once (producing a multi-dimensional array). For example, TRANSLATE is commonly used for recoding a variable, represented as a vector or column of a matrix. Because of extension, the output will be an image of **s** (the variable) with each value mapped into an index (one-column **table**), a scalar corresponding to the old value (two-column **table**), or a scalar corresponding to the interval in which the old value lies (three-column **table**). The latter case is particularly useful for categorizing continuous variables.

4. Array Functions

The functions described in this chapter are IDL's basic tools for manipulating arrays. None of these are specific to data analysis, but all of them are designed to make it easier to deal with the rectangular data aggregates commonly encountered in data analysis.

Unless stated otherwise, all the functions in this chapter are extended. Thus, if a function is described as expecting an array of a certain dimensionality, an argument of higher dimensionality will be broken into slices of that size before the function is applied.

4.1 ADJOIN

ADJOIN joins an indefinite number of vectors together end to end.

adjoin[vectors...] produces a vector of length equal to the sum of the lengths of the arguments which consists of the elements of the first argument, followed by the elements of the second, and so on.

With the use of appropriate keeps, ADJOIN can be used to add sections to higher dimensional objects. For example, to form two conformable matrices A and B into a 3-array with two planes on the third dimension ("laying A flat on top of B"), one might use the expression

```
(ADJOIN (KEEP A ALL) (KEEP B ALL))
```

This breaks A and B down into scalars, the ADJOIN assembles each pair into a 2-vector, and the extension mechanism assembles these results into an image of A and B.

4.2 DEAL

DEAL generates random permutation vectors.

deal[n] where **n** is a positive integer, returns a vector of length **n** consisting of the set of integers from 1 to **n** in a random order.

The name DEAL reflects the fact that applying the resulting vector as a selector to, say, a deck of cards shuffles the deck. DEAL vectors are useful for randomizing treatments in experimental design (see section 8.1). Note that (probably) different vectors are returned from separate calls to DEAL with the same argument.

As the length of its result varies with the value of its input, DEAL is not extended.

4.3 FORMAT and ELEMENTTYPE

format[a] returns the storage format (FULL or SYMMETRIC) of the array **a**.

elementtype[a] returns the type (INTEGER or FLOATING) of the elements of the array **a**.

4.4 GENVEC

GENVEC (for **gen**erate **vec**tor) produces a vector whose elements form a linear sequence, i.e., each element differs from its predecessor by a fixed amount. Such a sequence is completely determined by its first two elements and its last element.

genvec[initial;end] where **initial** is either a 2-vector or a scalar and **end** is a scalar, returns a vector generated as follows:

If **initial** is a 2-vector, its first element is the first element of the sequence, and the difference between its two elements is added to the *i*th element of the sequence to form the *i*+1st element, until **end** is reached.

If **initial** is a scalar, it is treated as the first element of a 2-vector whose second element is 1 if **end** is greater than **initial** and 1 otherwise. The sequence is then constructed as described above.

Thus, (GENVEC 4 7) generates the vector [4 5 6 7], and (GENVEC '(1 3) 9) yields [1 3 5 7 9]. Such vectors are particularly useful for selection. For example, if A is a vector of length N, then

A@(LIST (GENVEC N 1)) produces the reverse of A, and

A@(LIST (GENVEC '(1 3) N)) produces the vector containing every second element.

Note that for short sequences, it may be more convenient to simply type in the elements as a list.

As the length of its result varies with the value of its input, GENVEC is not extended.

4.5 IDLARRAYP

idlarrayp[a] is T if **a** is an IDL array, NIL otherwise.

4.6 INVERT

invert[m] returns the inverse of its argument matrix.

INVERT uses Gaussian elimination with total pivoting and is thus somewhat slow but numerically very stable. If the argument is not square, its square top left-hand corner is interpreted as a coefficient matrix and the rest of the matrix is taken as a set of value vectors. The inverse of the square corner appears in the result, along with the solutions to the set of linear equations corresponding to the value vectors.

4.7 MPROD

mprod[a;b] where **a** and **b** are conformable matrices or vectors, returns the usual matrix product of the two matrices.

The conformability requirement on the two arguments is that the second dimension of the first must have the same number of levels as the first dimension of the second. However, the vector case is treated specially so that a useful orientation of vector arguments is achieved. For example, an R-vector multiplied by an S-vector is treated as if it were an [R,1]-matrix by a [1,S]-matrix, resulting in a [R,S]-matrix. Consequently, the conventional dot-product of two equal-length vectors cannot be obtained with MPROD; the expression (RPLUS R*S) will produce that scalar quantity, since TIMES will be applied in an elementwise fashion.

4.8 ORDER

order[v;cfn] for **v** a vector and **cfn** an (optional) comparison function, returns the permutation vector which would, if applied as a selector, sort **v** into order according to **cfn**. ORDER does not actually sort **v**. If no comparison function is supplied, LESSP is used and the resulting permutation vector would sort the elements into ascending order.

For example, (ORDER '(4 6 3 9 100)) returns the vector [5 3 1 2 4]. To actually sort a vector A, one might use A@(LIST (ORDER A)).

4.9 RANK

rank[a] returns an image of the array **a** with each element being the rank of the corresponding element among the other elements of **a**. Although a similar result could be obtained using ORDER (specifically, order[order[a]]), RANK both corrects for ties by assigning the average rank to each member of a group of tied observations and treats cells containing NIL specially.

Cells containing NIL are ignored for the purpose of computing the ranks, so the ranks will range from 1 to the number of non-missing observations. Furthermore, the corresponding cells in the output array will also be NIL. That is, (RANK '(4 NIL 1 2)) is [3 NIL 1 2]. This is done because ranks are often used as data transformations, in which case the NIL should propagate. The arithmetic interpretation of NIL is, in any case, irrelevant in this context.

RANK accepts arrays of any dimension and constructs one set of ranks for the entire object. Thus, within-column ranks for a matrix M cannot be obtained simply by calling (RANK M). If the ranks are to be determined independently for each column, the larger structure must be explicitly broken down into vectors with KEEP, as in (RANK (KEEP M 2)).

RANK is often used in computing the statistics for non-parametric tests of significance (see Chapter 8).

4.10 REDUCE, RPLUS, RTIMES

reduce[a;f;startingval] where **a** is an array, **f** is a scalar function of two arguments, and **startingval** is a (optional) scalar, returns the value of applying f to the members of **a** in row-major order (i.e., for vector **a**, f[...f[f[startingval;a_1];a_2] ... a_n]). Thus, the first argument of **f** will be the result of previous applications, and the second argument will be the next element of **a.**

If **startingval** is specified and **a** has no elements, **startingval** is returned.

If **startingval** is not specified, then the initial invocation of **f** will be f[a_1;a_2] instead of f[startingval;a_1], provided **a** has at least two elements. Otherwise, $\mathbf{a}_1$ is returned if it exists, else NIL.

For convenience, the reductions of addition and multiplication have been given special function definitions:

(RPLUS A) is equivalent to (REDUCE A 'PLUS 0), and
(RTIMES A) is equivalent to (REDUCE A 'TIMES 1).

Thus (RPLUS A) will return the sum of the elements of A, 0 if A is empty, or NIL if A contains any missing values. (REDUCE A 'MAX) yields the maximum element in A, while (REDUCE A 'NCONC1 NIL) returns a Lisp list of the elements of A in row-major order.

4.11 RESHAPE

RESHAPE forms a new array of a given shape out of the elements of another.

reshape[a;newshape;newformat] where **a** is an array, **newshape** is an (optional) vector, and **newformat** is an (optional) format specification (either FULL or SYMMETRIC, default is FULL), returns an array of the shape and format given, whose elements are those of **a**. This is done by enumerating **a** in row-major order to fill the places in the result, which is also filled in row-major order. If **a** is exhausted before the newly constructed object is filled, it is repeated from the beginning in the same sequence.

For example, (RESHAPE 0 '(2 3)) produces a [2,3]-matrix of zeros. In the special case when **newshape** is NIL, **a** is "flattened" to a vector of the same length as there are elements in **a**.

4.12 SEEK

The SEEK function searches a vector for occurrences of specified value(s).

seek[sought;vec] where **sought** is either a scalar, a vector, or a function of one argument, and **vec** is a vector, returns a vector (possibly empty) of all indices *i* such that
vec@'(*i*) equals **sought** for scalar **sought**,
vec@'(*i*) is a member of **sought** for vector **sought**, or
apply[**sought**;**vec**@'(*i*)] is not NIL for function **sought**.

The indices are returned in the result in ascending order. For example

(SEEK '(1 3 7) '(6 5 4 3 2 1)) is [4 6] and
(SEEK 'PLUSP '(4 2 5 1)) is [2 3].

As the length of the result returned by SEEK varies with the values of its inputs, it is not extended.

4.13 SHAPE

shape[a] returns the shape of its argument as a vector. The shape of a scalar is the empty vector, a vector of no elements. (SHAPE (SHAPE A)) is always a 1-vector of the dimensionality of A.

4.14 SHIFT

shift[v;shift;fill] returns a vector formed by shifting vector **v shift** positions (positive **shift** shifts to the right; negative to left). The elements vacated from one end of **v** are filled by values shifted from the opposite end of **fill**.

If **fill** is a scalar, then that scalar will replace all vacated elements. If **fill** is omitted, then **v** itself is used, and SHIFT has the effect of rotating the elements of **v.**

If the absolute value of **shift** is greater than the length of **fill**, it is reused as many times as needed.

Examples:

(SHIFT '(1 2 3 4) 2 '(11 12 13)) is [12 13 1 2]
(SHIFT '(1 2 3 4) 2) is [3 4 1 2]
(SHIFT '(1 2 3 4) 2 0) is [3 4 0 0]

SHIFT provides a way of generating cumulative reductions for scalar functions. For example, the cumulative sum vector for a vector V is formed by

```
C _ (RPLUS (KEEP (SHIFT V (GENVEC (SHAPE V) 1 0) 0) 1) )
```

This constructs a matrix whose rows contain successively longer initial subsequences of V, padded on the left with zeros. PLUS-reduction along the rows of this matrix yields a vector of the cumulative sums. V can be recovered from C by

```
C   (SHIFT C 1 0)
```

which lines up each element in C with the cumulative sum of all levels before it. The subtraction yields the original values.

This technique can be used to convert between vectors containing cumulative and raw frequency counts.

4.15 TRANSPOSE

The TRANSPOSE function permutes the dimensions of an IDL array.

transpose[a;perm] where **a** is an *n*-array, and **perm** is an (optional) *n*-vector (for arbitrary *n*).
TRANSPOSE treats the vector **perm** as a mapping from the dimensions of **a** onto the dimensions of the result. Dimensions may be repeated or arbitrarily permuted in **perm**, provided only that, if the largest element of **perm** is *x* (which will be the dimensionality of the result), then all the integers between 1 and *x* appear in **perm**.

If **perm** is omitted, it is assumed to be (GENVEC *n* 1) and the dimensions of **a** are reversed. Thus (TRANSPOSE M) for matrix M produces the conventional transposition.

The result is the obvious dimension rearrangement when **perm** is a permutation vector of the integers from 1 to *n*. However, more complex rearrangements are possible. For example,

```
(TRANSPOSE A '(1 1))
```

with A an [*m*,*m*]-matrix will return the *m*-vector major diagonal of A.

As TRANSPOSE contains facilities for selecting the dimensions on which it is to operate, it is not extended.

5. Compression Functions

The distinction between compression and analysis functions is basic to IDL. Compression functions are used to transform a data array into a (typically smaller) array for further analysis. The compressed object is intended to be an information rich representation of the data which can serve as the base for many subsequent analyses. Data arrays can be compressed in different ways depending on the types of analysis desired. The different compression routines described here each preserve different aspects of the larger array.

Several of these functions take an optional argument described as a "weighting" vector. This is used when the data is a systematically biased sample of the target population. This bias can often be corrected by counting each subject as representing more or less than one observation. For each subject in the array to be compressed, the corresponding element of a weighting vector indicates the number (not necessarily integral) of observations that the subject is to represent. Thus, a weighting vector must have as many elements as there are subjects. The default weighting for a subject, if no weighting vector is supplied, is 1.0. Subjects with a non-positive or NIL weight are simply omitted.

5.1 COUNTS

counts[a] returns the sum of the values of the array **a**. It is similar to RPLUS except that it skips over NIL (rather than returning NIL if NIL is encountered, as RPLUS does).

Note that, since the output of GROUP is kept on the dimensions of grouping, COUNTS will operate *within* each cell of a GROUP classification to produce a contingency table. Thus, for a matrix A,

```
(COUNTS (GROUP A@’((VOTE SEX)) 1))
```

produces a cross tabulation of VOTE against SEX.

5.2 COVAR

COVAR computes a *covariation matrix*, a symmetric matrix of the mean-centered cross products of the columns of a matrix. A covariation matrix can easily be converted to traditional correlation and (variance-) covariance matrices. However, it contains more information than either of these and is thus more useful for subsequent analyses.

covar[a;wt] where **a** is a matrix, and **wt** is an (optional) weighting vector, produces a symmetric [c+1,c+1]-matrix whose first c rows and columns correspond to the c variables (columns) of **a**. The last row and column correspond to a Constant "variable" whose value is taken to be 1.0 for all rows of **a**. In the first c rows and columns, the ij off-diagonal elements contain the sum of cross-product deviations from the means of the ith and jth variables, while the diagonal element gives the sum of squared deviations from the corresponding variable mean. In the Constant row, the ith off-diagonal element is the mean of the ith variable, while the diagonal element is $-1/n$, for n the number of observations on which the matrix entries are based.

The result corresponds to a matrix of "raw" cross-products from which the Constant "variable" has been swept out (see section 6.6). COVAR does this as the result is constructed for reasons of numerical stability.

If there are any missing values, then each cross product will be based on all rows for which data exist (i.e., observations are deleted pairwise). The figures in the final result are adjusted to appear as if they are based on the smallest number that any individual entry is based on, and that *n* is used to compute the Constant diagonal cell. The function PAIRN is available to compute the actual pairwise *n*.

5.3 GROUP

GROUP produces cross-classifications of data values as defined by the values of other observations on the same individuals. For example, to determine whether the heights of young and old men and women differ, their measured heights are grouped by their age and sex. Summary statistics can then be computed separately for the groups and compared. Cross-classifications in IDL are represented in the dimensional structure of an array, usually in an array's leading dimensions. Consequently, the extension mechanism will cause certain analysis functions to apply within the cells of the classification without further specification.

group[attribs;values;dim] for an [*s*,*m*]-matrix **attribs** and an *n*-array **values** with extent *s* on dimension **dim** (optional, default is one), constructs a classification array of *m*+*n* dimensions. Each row of **attribs** is considered as a subscript describing a cell in an *m*-way classification, and GROUP places the slice of **values** corresponding to that row into that cell of the classification. The slice corresponding to the *i*th row of **attribs** is the plane formed by selecting from **values** the *i*th level of dimension **dim**. The *m* leading dimensions of the result are induced by the columns of **attribs**; the *n* trailing dimensions are the original dimensions of **values**.

If **values** is a number, it will be reshaped to a vector of length equal to the number of rows of **attribs** before the grouping is performed. If **values** is NIL, it is defaulted to one. This convention simplifies the construction of contingency tables: COUNTS of such a GROUP computes the number of cases in each cell of the classification.

If **attribs** is an *s*-vector, it is treated as a [*s*,1]-matrix.

The levels induced in the classification by each attribute (column of **attribs**) are determined as follows: If the column has a codebook, then only those values in the codebook form levels in the classification. Any other values are treated as "wild scores" and are omitted. Thus, if A is a matrix with columns SEX and HAIRCOLOR (coded blond, brown, black), then (GROUP A@'((SEX HAIRCOLOR)) ...) will produce a two by three classification with one cell for each possible combination of SEX and HAIRCOLOR.

If the column does not have a codebook, then each value actually present is used. This requires more computation than the value-labelled case, as a preliminary pass over the attributes is made to determine how many distinct values there are.

The resulting array has the classification dimensions kept, so that generic functions (MOMENTS, COUNTS, RPLUS, etc.) will automatically apply *within* the levels of the grouping unless these keeps are explicitly overridden.

Frequently, the attributes and values are both variables in the same data matrix. In such cases, the full power of the IDL selection mechanism, including label selectors, can be used to construct the appropriate GROUP arguments. Thus, for a matrix M, a three-way cross classification of the variables AGE and INCOME by the attribute variables SEX, EDUC, and VOTE will result from

```
(GROUP M@'((SEX EDUC VOTE)) M@'((AGE INCOME)) )
```

Since the shape of the result depends on the attribute data, GROUP is not extended.

5.4 MOMENTS

MOMENTS computes the first *m* moments of an array.

moments[a;wt;m] where **a** is an array, **wt** is an (optional) weighting vector, and **m** is an optional (defaulted to two) integer giving the number of moments to be taken, returns a vector containing the zero through **m**-th moments of the values in **a**, defined as follows:

M_0	= the number of non-missing observations, i.e., Swt.
M_1	= the mean
M_2	= the sample variance = $Swt(a\ M_1)^2/(M_0\ 1)$, and
M_i	= $Swt(a\ M_1)^i/M_0$ for $i > 3$.

The third moment M_3 is related to the skewness, and the fourth M_4 to the kurtosis by

Skew	= $M_3/M_2^{3/2}$
Kurtosis	= $M_4/M_2^2\ \ 3$

In other words, the skew and kurtosis are essentially the third and fourth moments, scaled by the variance. Higher moments are available from this routine but are not commonly used. Note that, while the variance is unbiased, subsequent entries of the result are biased estimates of the population parameters.

MOMENTS collapses an entire array down to a vector of moments, ignoring the dimensional structure. The dimensions often represent a cross-classification for which within-cell moments are desired. If some of the dimensions are kept, they will be withheld from MOMENTS, and within-cell results will be computed for the kept dimensions collapsing across all the unkept ones. This meshes nicely with GROUP, which keeps the leading classification dimensions:

```
(MOMENTS (GROUP M@’((SEX EDUC VOTE)) M@’(INCOME) ))
```

yields a table of moments appropriate for a 3-way analysis of variance of the variable INCOME in data matrix M.

5.5 PAIRN

pairn[a;wt] for **a** a matrix and **wt** an (optional) weighting vector, returns the pairwise *n* matrix (sum of the weights for rows having data on both of a pair of columns) for the columns of the matrix. If **a** has *c* columns, the result will be a symmetric [*c,c*]-matrix each cell of which gives the *n* that the corresponding entry of (COVAR **a wt**) would be based on.

5.6 POOL

POOL could also be thought of as an analysis function, in that it operates on a compression of the original data (a MOMENTS array). However, it acts to further compress the data, so it is treated here. POOL collapses a MOMENTS array into an array with fewer classifying factors (thus its name, as it corresponds to the "pooling" operation of the analysis of variance).

pool[mtable] where **mtable** is an *n*-array whose last dimension is interpreted as the Moment dimension of a MOMENTS table, returns a vector formed by collapsing all the observations represented by **mtable** into a single classification.

By itself, this is not very interesting. Its chief use is in combination with KEEPing to select the dimensions that are to be preserved. Thus, (POOL (KEEP M I J K)) collapses the MOMENTS table M, preserving the classifications which make up its I, J, and K dimensions. The result is exactly the same as MOMENTS would have produced from the original data with only these dimensions kept.

6. Analysis Functions

Analysis functions are those that perform statistical analyses of some sort, usually on a compressed object. This chapter gives a relatively formal description of their behavior; their use in data analysis is illustrated in Chapter 8.

6.1 ANOVA

ANOVA computes an analysis of variance table given a MOMENTS array, a specification of its random factors, and a design showing its crossing and nesting relations.

anova[mtable;random;nesting] where **mtable** is an array of moments, **random** is an (optional) specification of the factors to be considered random, and **nesting** is an (optional) specification of the nesting relationships between the factors of the analysis, returns a matrix which gives the summary table for the analysis of variance.

The array of moments is of the form produced by MOMENTS, with a last dimension containing either the N, mean and variance, and with leading dimension(s) forming the classification structure (or *factors*) for the analysis. If there is only one observation per cell, the moments dimension may have a single level containing just that observation: the N is defaulted to one, and there will be no within-cell error term.

The random argument is a list of factors, specified by dimension name or number, which will be assumed random in the construction of the *F*-tests. All other factors are assumed to be fixed. For example, in a typical repeated-measures design the subject factor is considered random, so a specification of (SUBJECTS) would be appropriate.

The nesting argument is a list of lists, each one corresponding to a single nested factor (dimension of the moments array). The first element of each list specifies the nested factor; subsequent elements indicate the factors inside which it is nested. As a factor may be denoted by either its dimension name or its number, a nesting specification of

```
((WARD CITY STATE) (CITY STATE))
```

indicates that the factor WARD is nested inside the factors CITY and STATE, and that CITY itself is nested within STATE.

In the table of moments, nested factors appear as if they were *crossed* with their nesting factors. That is, without the nesting specification, the first level on the WARD dimension would be taken to represent the same ward for each of the levels on the STATE dimension, and the ANOVA output would include WARD by CITY interaction effects. The nesting specification eliminates those sources of variation, causing them to be pooled into the appropriate WARD effects.

ANOVA returns a matrix laid out and labelled like a conventional analysis of variance summary table. The dimension labels of the moments table are used to generate appropriate labels for the rows (sources of variation) of the anova table. Statistics for the grand mean are given as the first row, while the within-cell error, if any, is found in the last row. The columns are labelled as SumSq, df, MS, F, and p. The F column is NIL for effects that cannot be tested directly; the EMS function provides expected-mean-square coefficients and may help in computing quasi-*F* values.

If the within-cell frequencies are not equal, ANOVA computes an unweighted-means approximation.

6.2 EMS (Expected Mean Squares)

EMS computes an array of coefficients that define the expected values of mean squares in an anova design with arbitrary crossing and nesting relations and arbitrary combinations of fixed or random factors. From these coefficients, the structurally appropriate *F* and quasi-*F* ratios for testing various null hypotheses may be determined.

ems[nlevels;random;nesting] where **nlevels** specifies the number of levels for each factor, **random** is an ANOVA specification of the random factors, and **nesting** is an ANOVA nesting specification, returns the matrix of effect coefficients for the described design.

If **nlevels** is a vector, it is interpreted as being the "shape" of the factor portion (all but the Moment dimension) of a moments array. The entry for each factor is the number of levels it has. In a nested design, this is the number of levels a nested factor has within all other factors; thus, if a design has wards nested within cities, and there are 20 wards and 4 cities, then there must be 5 wards in each city, and **nlevels** must be [5 4] if **nesting** is ((WARD CITY)).

If **nlevels** has more than one dimension, it is assumed to be a moments table, and EMS computes the number of levels directly from its shape.

EMS interprets **random** and **nesting** in exactly the same way as ANOVA.

EMS returns an [*n*,*n*]-matrix, where *n* is the number of lines in the corresponding ANOVA table, excluding the line for the grand mean and the line for within-cell error, if one exists in the design. The *i,j* entry in the EMS table is the coefficient for the *j*th source of variation in the expected-value for the mean square of the *i+1*th row of the ANOVA table. Consider as an example a simple 2 x 4 crossed design with factor 1 random and 2 fixed. The expression

```
(EMS '(2 4) '(1))
```

produces the table

	Coeff		
Source	Factor1	Factor2	1*2
Factor1	4	0	0
Factor2	0	2	1
1*2	0	0	1

This indicates that the expected mean square for Factor2 is $2s^2_2 + s^2_{1*2}$. In a design with a within-cell error term (i.e., more than one observation per cell), every expected mean square includes the term $+s^2_{error}$; this term does not appear in the EMS table. Thus, if a line in the EMS table has a nonzero entry on the diagonal and all other entries zero, its appropriate *F*-test denominator is the within-cell error line.

6.3 HIST (Histogram)

hist[v;file] prints onto the file **file** (primary output file if not given) a histogram display of the vector **v**, returning **v**. If **file** is not open for output, it is opened, the histogram is printed, and **file** is closed. The axis of the histogram will run down the page, and the plot unit (the number of cases represented by a single character) will be scaled with respect to the page width to allow maximum discrimination. Both positive and negative counts are allowed, and the axis of the histogram will be moved to accomodate the range encountered in the vector. To speed the printing of very large histograms, sequences of more than HISTRPTLINES (initially 5) lines with identical counts are suppressed and replaced by a sequence of three vertically aligned dots.

6.4 PLOT

plot[y;x;file] prints a plot of the vector **y** against the vector **x** (1 through length(**y**), if not given) to the file **file**, returning NIL. If **file** is not open for output, it is opened, the plot is printed, and **file** is closed. If **x** is given, the result will be an x-y (scatter) plot of elements of **y** against the corresponding elements of **x**. Otherwise, it will be a series graph. The axes of the plot will appear at the left and bottom sides of the plot, which will be scaled in terms of the page size to give maximum legibility. The vertical size of the plot will be chosen so that both axes are approximately the same length (rather than the same number of print positions). The scaling from length to print positions is controlled by the global variable PLOT.AXIS.RATIO (notionally the ratio of the lengths of vertical and horizontal print positions), initially .6.

6.5 NORM

norm[m] normalizes a matrix **m** by dividing each entry by the square root of the product of the diagonal elements in its row and column. It is commonly used to norm an inner product representation of a vector space, e.g., to turn covariation matrices into correlation matrices.

For non-square arrays, NORM selects out the top left hand square corner. From this, or from the whole array for square **m**, those row/columns which have negative diagonal entries are discarded. This is useful when applying NORM to matrices which have had a transformation applied to them which leaves some diagonals negative (e.g., SWEEP). Then, each element of the result is divided by the square root of the product of its diagonals, and the normed matrix is returned.

6.6 SWEEP

SWEEP performs successive orthogonalization of an inner product matrix (such as those produced by COVAR) using the SWP and RSW operators of Dempster (1969). SWEEP may be used to obtain most regression based statistics.

sweep[m;outvars;invars] where **m** is a matrix and **outvars** and **invars** are (optional) selectors for the second dimension of **m**, returns a matrix which is the result of having swept the columns of **m** selected by **outvars** *out* and the columns selected by **invars** *in* (in that order). (In the notation used by Dempster (1969), sweep[**m**;**outvars**;**invars**] may be described as RSW(SWP(**m**,**outvars**),**invars**)). Either or both **outvars** and **invars** may be NIL, in which case the corresponding operation is not carried out. Sweeping will be carried out in the sequence specified in the **outvars** and **invars** arguments unless **m** is non-symmetric. In this case, total pivoting is used to maximize numerical stability.

For a precise definition and discussion of the properties of the SWEEP algorithm, the reader is referred to Dempster (1969). The common statistical interpretations of SWEEP operations are discussed in Chapter 8 below, along with computational methods of extracting information from the results. Here we simply describe the interpretation of the result of sweeping a set of variables out of some covariation matrix. The *i,j* off-diagonal element of the result of such a sweep contains

for swept *i* and unswept *j*, the regression coefficient of variable *i* as a predictor for variable *j* in the regression equation including all the swept out variables as predictors.

for unswept *i* and *j*, the partial covariation between variables *i* and *j* controlling for all the swept out variables. The partial correlation is obtainable by scaling this quantity with respect to its current diagonals using NORM.

The *i*th diagonal element contains

for swept *i*, the negative reciprocal of the residual sum of squares of this variable regressed against the other swept variables. This quantity is closely related to the multicollinearity of this variable with the other swept variables.

for unswept *i*, the residual sum of squares (unaccounted for variation) in variable *i*. The R^2 for *i* is computed by dividing this quantity by its value before the sweep.

Sweeping a variable in to a matrix merely undoes the effect of previously having swept it out. Sweeping out all variables from a matrix gives the negative of its inverse.

7. Data Input and Output

The material in the preceding chapters assumed that the data to be analyzed already existed as an IDL array. This chapter considers how data may be entered into an array from an external representation, either from list structure residing in memory or from a sequence of characters on a file. Functions for converting an array back into an external representation are also described, so that, for example, objects created by IDL computations may be preserved from one session to the next.

7.1 To and from list structure

The easiest way of entering data is simply to type it in as list structure at the point in the IDL session when it is needed. Thus, typing

```
X _ '(4 2 6 9 3 11 2 8 31 0)
```

saves the ten observations as a list in the variable X. As all IDL functions will convert numeric lists to arrays, X may be used whenever a vector would be appropriate (see section 1.7). This method has the disadvantages that the conversion would be carried out every time X was used as an array, and the resulting arrays have no labels or titles. Thus, IDL provides special functions for explicitly constructing arrays from list structure that also includes labelling.

idlarray[data] returns an array constructed according to the description found in **data**. **data** is a list in the following format

(*{title} organization {keeps} {format} elements*)

The braces *{}* indicate that an item is optional and need not be specified. The *title*, if present, is simply a string enclosed in double-quotes. The *organization* specifies the shape and labelling of the array, and the optional *keeps* indicates its kept dimensions. The optional *format* is either SYMMETRIC or FULL (the default), and *elements* is a list of the values that will fill up the array in row-major order.

The *organization* is a list containing one sub-list for each dimension of the array. This is of the form

(*dim = nlevels {level-labels*$_1$ *level-labels*$_2$ *...}*)

dim is either the dimension number or the dimension label, *nlevels* is the number of levels for that dimension, and *level-labels*$_i$ specifies the level label and, for the value-labelled dimension, the value-labels for level *i*. If there are no value-labels, the specification may simply be the atomic level label, or NIL or the level index if the level is unlabelled. If there are value-labels, then the specification is a list consisting of the level label followed by the codebook for that level.

keeps, if present, is a list of the form (kept k_1 k_2 ...), where each k_i is either a dimension number of label specifying a kept dimension.

For example, the expression

```
(IDLARRAY '("Another random matrix"
        ((Subject = 4)
         (Variable  = 3 (SEX (1 MALE) (2 FEMALE)) AGE VOTE))
        (kept Variable)
        (1 24 2 3 31 1 2 28 3 1 25 2)))
```

produces the matrix

```
      Another Random Matrix
      Kept:  Variable
           Variable
  Subject        SEX      AGE     VOTE
        1       MALE       24        2
        2          3       31        1
        3     FEMALE       28        3
        4       MALE       25        2
```

The inverse of IDLARRAY is also provided:

listarray[a] produces a list-structure description of the array **a** in the form outlined above. This can be given back to IDLARRAY to reconstruct an array equivalent to the original.

Usually, data is collected and initially entered in the form of a subjects by variables matrix. These arrays can be described in the format above, but, for convenience, a more perspicuous list-structure representation is also allowed for such arrays. This representation groups the rows into individual lists and permits the row-labels to be specified next to the corresponding data values. The functions IDLMATRIX and LISTMATRIX process this special form of description.

idlmatrix[data] returns a matrix constructed according to the instructions and data found in **data**. **data** is a list each element of which is a list representing one row of the output. The list is a sequence of data values optionally preceded by the level label for that row. Before the row specifications there may be one list of the form

(TITLE *title-string dimension*$_1$*-label dimension*$_2$*-label*)

and another of the form

(LABELS *level*$_1$*-labels level*$_2$*-labels* ...)

These level labels are for levels on the second dimension of the result, and each one may be NIL (meaning no level label), a non-NIL literal atom which provides a level label but no codebook for the corresponding level, or a list of the form

(*level-label codepair*$_1$ *codepair*$_2$ ...)

For example, an array with two subjects, John, a 32 year old male, and Susan, a 27 year old female, could be created by

```
(IDLMATRIX '( (TITLE "An array with two subjects" Subjects Variables)
              (LABELS (Sex (1 Male)(2 Female)) Age)
              (John  1  32)
              (Susan 2  27) ))
```

The function LISTMATRIX is the inverse of IDLMATRIX. It produces the list structure corresponding to an IDL matrix:

listmatrix[m] returns a list structure representing the elements and labels of the IDL matrix **m**. This structure can be passed to IDLMATRIX to reconstruct a matrix equivalent to **m**.

List-structure descriptions can be written on files and read back in to form arrays, and this is one way of preserving arrays. However, the functions described below achieve the same effect without allocating intermediate list-structure and are thus more efficient. The main advantage of list structure representations is that they can be manipulated by the large number of Lisp list-processing functions, including the powerful editor (see Teitelman, 1978).

7.2 To and from files

Data can be stored on files in either the full-array format or the special matrix format. Arrays or matrices can be read into memory by reading the appropriate expression from the file to form the list structure, and then giving that structure to either IDLARRAY or IDLMATRIX. For example, suppose the file MYDATA contained only expressions representing the matrix form of an array, inserted by means of a text-editor familiar to the user. The data could be formed into an IDL matrix and bound to the variable D by means of the following expression:

```
D _ (IDLMATRIX (READFILE 'MYDATA))
```

The Lisp function READFILE opens a file, reads all the expressions from it and forms them into a list structure, closes the file, and returns that list structure. For this example, the file MYDATA must not have the outer parentheses enclosing the titles and rows of the matrix, since READFILE will itself insert those.

If the more general form of description is on the file, the function IDLARRAY may be used instead of IDLMATRIX to construct the array. However, it is more efficient to use the function READIDARRAY, for this constructs the array in one step, without building the intervening list structure:

readidlarray[file] constructs an IDL array from an IDLARRAY-type expression read from **file** (or the Lisp primary input file if **file** is NIL). If **file** is not an open file, it is opened, the expression is read, and then **file** is closed.

The information in arrays may be saved on a file by first converting it to list structure and then using any of the Lisp printing functions PRIN2, PRINT, PRINTDEF, etc. to print the structure on the open file. For the general array format, the function DUMPIDLARRAY may be used instead:

dumpidlarray[a;file] writes an IDLARRAY-type list expression on **file** (or the Lisp primary output file if **file** is NIL) from which an array equivalent to **a** can be constructed. If **file** is not an open file, it is opened, the expression is printed, and then **file** is closed.

The Lisp file package command IDLARRAYS can be used to dump and restore IDL arrays on LOADable files, via DUMPIDLARRAY and READIDLARRAY. This permits arrays to be saved on the same files as functions, variables, and other information, and to be restored by the simple Lisp function LOAD. The appropriate dumping and reading functions are automatically supplied. To include an array on a LOADable file SURVEYINFO, the Lisp variable SURVEYINFOCOMS must be a list of commands one of whose elements begins with IDLARRAYS and specifies the name of a variable whose value is the array to be dumped. Other commands might specify other objects to be saved. For example, if SURVEYINFOCOMS were bound to the command list

```
((VARS X Y) (IDLARRAYS SDATA) (FNS RECODE))
```

then the expression

```
(MAKEFILE 'SURVEYINFO)
```

would save on the file SURVEYINFO the values of the (non-array) variables X and Y, the array bound to SDATA, and the user function named RECODE. If in another IDL session the expression

```
(LOAD 'SURVEYINFO)
```

were evaluated, the variable values and function would be restored. The Interlisp Reference Manual (Teitelman, 1978) has a very extensive discussion of file commands, MAKEFILE, and LOAD.

Rather than saving particular arrays from one session to be used in another, it is also possible to preserve the complete state of the current session on a file. This file can be run at a later time to resume the suspended analysis session. The Lisp function SYSOUT moves the state of the current session to a file:

sysout[file] saves the state of the current IDL session on **file**. If **file** does not include an extension in its name, the extension SAV will be supplied by default.

A SYSOUT file may be resumed simply by running that file instead of IDL to start the session. For details, see Teitelman (1978).

8. Data Analysis with IDL

This chapter presents some simple ways to do standard analyses, some tricks that might not be apparent from the rest of the manual, and some clues to the effective use of IDL for advanced statistical work. Even a quick reading of this chapter will give the potential IDL user some ideas as to what the system can do. When the time comes that a specific analysis is needed, the appropriate section in this chapter can serve as a guide, adding a more practical flavor to the reference material in other chapters.

8.1 Ante data

Data analysis systems are not ordinarily used before the data are gathered, but there are two ways in which IDL can be of use with no real data in hand at all. The first is in the construction of experimental designs; the second in the generation of artificial data.

8.1.1 Experimental design

Experimental design often requires randomly assigning subjects into groups so that certain constraints are satisfied. With few subjects this can easily be done by hand, but with many subjects or complicated designs it is useful to have mechanical help. The basic IDL tool is DEAL, which returns a random permutation of the integers from one to its integer argument, in a vector of that length. This can be related to the design in two ways. The first indicates which subjects are in each condition. For example, assume a 2 by 2 factorial design (four conditions), with 13 subjects per condition. The problem is to assign 52 subjects (each represented by a subject number from 1 to 52) into one of four conditions. This can be done with

```
(RESHAPE (DEAL 52) '(2 2 13) )
```

which will distribute each of the randomly ordered subject numbers into a cell in a first variable by second variable by subject within condition array. This array can be printed out to provide the condition lists, or it can be saved and further manipulated. One obvious manipulation would be to sort the subject numbers within the conditions, to make it easier to find a given subject.

The second way of using DEAL produces a list in subject number order, giving for each subject the condition that he is in. Here we want to keep the subject numbers fixed and deal around the conditions, rather than vice versa. We generate a vector whose length is equal to the number of subjects and whose elements are the condition numbers, each condition number appearing as often as there are subjects to be assigned to that condition. The vector is randomly permuted by selecting with a DEAL vector:

```
(RESHAPE '(1 2 3 4) 52) @ (DEAL 52)
```

This relies on the cyclic repetition of RESHAPE to produce the desired condition vector.

Note that these two procedures do *not* in general give the same subject assignments (and most certainly do not if different DEAL vectors are used). It takes a little more trouble to generate both lists for the same design, and that problem is left for the interested reader to explore.

Some experimental situations require that treatments be counter-balanced according to a random Latin square. The SHIFT and DEAL functions combine to produce this kind of experimental plan:

```
(SHIFT (GENVEC 1 6) (GENVEC 1 6)) @ (LIST (DEAL 6) (DEAL 6))
```

Because of the extension for its second argument, the SHIFT produces a square each row of which contains the numbers 1 through 6 "rotated" one position from the previous row. This is a 6 by 6 Latin square whose rows and columns are then randomized by the DEAL selections.

8.1.2 Artificial data

The random number generators RAND and RANDN are the key to artificial data generation. The function RAND (from Lisp) generates a uniformly-distributed random number in the range given by its arguments. RANDN generates a random number from the normal distribution with a specified mean and standard deviation. Both functions allow the user to control the numbers produced (and so to generate the same sequence twice, if necessary) by setting the random number generators with the Lisp function RANDSET. Generating data with distributions other than these is non-trivial and expert advice should be sought.

The result of a random generator is a number and can be used anywhere that a number can be. One must be careful to make sure that the generator is called an appropriate number of times. Thus

```
(RESHAPE (RAND 1 5) 10)
```

does not generate a vector of ten random numbers between one and five, but rather a vector full of ten copies of the same random number between one and five, just like

```
(RESHAPE 6+1 10)
```

generates a 10-vector of sevens. To achieve the former result, the random generator must be called once for each new number desired. For example,

```
(EAPPLY* '(LAMBDA NIL (RAND 1 5)) '(SCALAR) (GENVEC 1 10))
```

uses EAPPLY* to evaluate (RAND 1 5) for each element of the vector 1 through 10. This produces an image of that 10-vector each element of which is a different number drawn randomly from the uniform distribution between one and five. Note that the elements of the GENVEC were not actually used in the computation at all! Their role was simply to force the extension mechanism to "pulse" RAND ten times. This technique is often useful to cause a computation to be repeated without using loops, especially when the results are to be saved in an array.

We can use a variant of this technique to obtain, from the function RANDN, a vector of ten random numbers from a specified normal distribution:

```
(RANDN (RESHAPE M 10) SD)
```

Here, M and SD are scalar variables bound to the desired mean and standard deviation. The RESHAPE produces a 10-vector each element of which is M, and the extension causes RANDN to be called once for each of these ten elements.

A common application of random numbers is random sampling, or making any decision on a stochastic basis. Suppose you want to sample from the elements of a vector with a sampling probability of one fifth. This could be done by testing the expression (RAND 1 5)=1 for each element of the vector, which will be true one fifth of the time in the limit. However, in a finite number of trials it will not be true *exactly* one fifth of the time but will vary around that. To extract exactly one fifth, randomly chosen, use as a selector one fifth of a DEAL vector of length N where N is the length of the vector being sampled (a DEAL vector of length N/5 will *not* do, as that selects the first fifth of the source vector, but in a random order). This method samples without replacement. To sample with replacement, select with a vector of length N/5 each element of which is set to (RAND 1 N).

Artificial data generated in these ways may be used directly as input to the compression and analysis routines (e.g., to test the robustness of various analyses), or may be used to increase the "noise" in existing data by adding some appropriately scaled random number. There are various other uses of random generators (e.g., simulations) that are beyond the scope of this chapter.

8.2 Data

Once the data have been collected, they can be modified and reorganized for subsequent analysis using IDL's array manipulation tools.

8.2.1 Data entry

There are two ways to input data to IDL. The simplest is just to type it in at the point in the session when it is needed. However, this is unwieldy for large amounts of data. In this case, the data may be prepared as a text file, using a text editor, and read into IDL using the functions described in Chapter 7.

8.2.2 Recoding

Usually the form in which data is most easily collected is not the most suitable for subsequent analysis. Sometimes individual data values must be adjusted to compensate for peculiarities of the collection procedure or to eliminate observations that are known to be invalid for idiosyncratic reasons. The simplest way of "purifying" the data matrix is to use the ASSIGN operator to change specific values indicated by a selector. For example,

```
X@'(101 SEX) _ 2
```

changes to 2 the value of the variable SEX for subject 101 in the matrix X.

Often, however, the same change must be applied to many different subjects, as when the raw observations must be transformed or combined in some systematic way. Such systematic recoding may be accomplished by means of the array and arithmetic operations using function extension. For example, the column of values representing the variable INCOME in a data array can be converted to a column matrix containing the logarithms of INCOME values via the expression

```
(LOG X@'((INCOME)))
```

The resulting matrix may be used independently of X in various analyses. If the transformed values are required for many subsequent analyses, it may be convenient to combine them with the old values in X. The new values can be assigned into X to replace a column of previous values by

```
X@'((INCOME) _ (LOG X@'((INCOME)))
```

or a new matrix can be constructed by adjoining the new values to X with

```
(ADJOIN X (LOG X@'((INCOME)))
```

In either case, some change in labelling is also usually desirable.

As another example, the variable INCOME can be converted to standard scores by first computing its mean and variance and then performing the appropriate arithmetic operations:

```
M _ (MOMENTS X@'((INCOME)))
(X@'((INCOME)   M@'(Mean)) / (SQRT M@'(Variance))
```

The function extension mechanism may be invoked explicitly for more elaborate data transformations. For example, a common technique in the analysis of questionnaire data is to combine answers to individual questions into scale scores, either by summation or averaging. The following expression will produce a subjects by variables matrix whose first column contains the sum of the values in each row of X and whose second column contains the average of those values:

```
([ELAMBDA ((R VECTOR))
          (ADJOIN (RPLUS R) (MOMENTS R)@'(Mean))]
      X)
```

The ELAMBDA will evaluate the ADJOIN form with R bound in turn to each row of X, yielding a different 2-vector each time. The output matrix, formed by combining these together, may be used independently, may replace some 2-column section of X, or may be ADJOINed to X, as described above.

8.2.3 Weighting

The compression routines MOMENTS, COVAR, and PAIRN have an optional argument providing for differential weighting of cases. This argument takes the form of a vector that specifies the weight to be given to each unit of analysis. Weighting vectors may be used either to correct bias in a sampling procedure or to allow the entry of data in frequency compressed form.

The first case arises when a single subject in a subjects by variables data matrix is used to represent several observations rather than just one. For example, if the sampling in a survey results in the relative under- or over-representation of various groups in the sample, each observation in the sample might be weighted by the reciprocal of its group's sampling fraction, thus allowing unbiased estimates of the population parameters to be obtained. Such a weighting vector is commonly an additional variable selected from the same subjects by variables matrix that is being compressed. However, it might be in a completely separate array or be computed from other information.

The other use of weighting vectors is when there are many observations each of which can be one of a small set of measured values. In this case, it may be more efficient simply to count the number of observations having each value rather than to represent each observation as a row of a subjects by variables data matrix. The observed vector of frequencies can then be used as a weighting vector to compute summary statistics on the measure variables.

8.2.4 Subsetting

Analyses may be confined to particular subsets of a data array by recoding, weighting, selection, or some combination of these.

The recoding technique relies on the fact that almost all compression and analysis routines ignore missing observations, so that transforming a data value to NIL effectively removes it from consideration. As for recoding in general, specific values may be recoded to NIL by assignment, or a systematic transformation may be done to eliminate invalid or wild observations. Thus, if MI is the mean of the variable INCOME and SDI is its standard deviation, all observations two or more standard deviations away from the mean can be omitted using the expression

```
([ELAMBDA ((IVAL SCALAR))
           (if (GREATERP (ABS IVAL MI) 2*SDI)
                then   NIL
             else IVAL)]
    X@'((INCOME)))
```

The *if-then-else* expression returns NIL if IVAL is outside the specified interval and IVAL otherwise. Alternatively, the cut-off points can be placed in a translation table:

```
(TRANSLATE X@'((INCOME) (LIST (LIST NIL MI 2*SDI NIL)
                              (LIST MI+2*SDI NIL NIL)))
```

This translates all extreme scores to NIL leaving other scores unchanged.

Subsetting can be achieved through weighting vectors because the compression functions ignore observations whose weights are not greater than zero. Thus, subjects can be excluded from analyses by giving them weights of zero or NIL. Several such vectors may be defined to indicate membership in subgroups which are of continuing interest. By specifying the group vector as the weighting vector for a given compression, only data from the members of that group will be included. This method of subsetting is only applicable to analyses that begin with the weight-accepting compression functions.

The most direct and powerful method of confining analyses to particular subsets of the data is to use selection. If the subjects to be included (or deleted) are known by number or label, then a selection on the subject dimension can be used to produce the desired subset. Thus,

```
A@(LIST (ADJOIN (GENVEC 1 50) (GENVEC 52 99)) '(SEX AGE EDUC) )
```

yields a sub-matrix of A containing the specified three variables for subjects 1 through 99 omitting number 51. This technique is the most general of all, since it it does not depend on special treatment of missing data or weighting vectors.

If several analyses are to be performed on the data for a particular group, it may be convenient to save the subject selector. Thus,

```
GRP1 _ (ADJOIN (GENVEC 1 50) (GENVEC 52 99))
A@(LIST GRP1 '(SEX AGE EDUC) )
```

produces the same result as the example above, but GRP1 is available to select the same set of subjects again.

Since subject selectors can be arbitrary vectors, they can be constructed using IDL's array functions, such as the ADJOIN and GENVEC above. The function SEEK is especially useful for locating all subjects whose data satisfies a given condition:

```
(SEEK 1 ([ELAMBDA ((VARS VECTOR))
                    (if (AND (LESSP VARS@'(AGE) 30)
                             (GREATERP VARS@'(INCOME) 5000))
                        then 1
                     else 0)]
         X) )
```

yields a selection vector containing the subject numbers for all subjects under 30 earning more than $5000. The ELAMBDA considers the vector of variables for each successive subject in X, and returns a vector containing 1 for every subject meeting the specified conditions. The SEEK returns a vector of the subject numbers corresponding to those 1's.

If subjects have been organized into a cross-classification structure (e.g., by GROUP), subsetting can be accomplished by selecting single levels of the classifying dimensions instead of multiple levels on the subject dimension. Thus, if B is a two way classification (by SEX and VOTE), then

```
(COVAR B@'(MALE DEMOCRAT ALL (INCOME AGE EDUC)))
```

will produce the three-variable covariation matrix for only those subjects in the MALE-DEMOCRAT cell of the cross-classification. If COVAR were applied to B without selection, the extension mechanism would cause the compression to be done separately for each of the groups, producing an array of within-cell covariations.

8.3 Correlational statistics

The basis for all continuous multivariate statistics in IDL is the covariation matrix. This is a symmetric matrix each row and column of which corresponds to a variable (column) of some data array, and whose elements contain the covariation between the two variables that are used to select it. Such matrices are produced from data arrays by the COVAR function. This function takes an array as argument, and computes the covariation matrix between the variables which form the columns of the data array. For example, if A is a matrix with four columns, labelled AGE, STATUS, INCOME and VOTEPCT,

```
C _ (COVAR A)
```

computes the covariation matrix for these variables and stores it in C. This produces a matrix such as

```
        Covariations of SURVEY DATA
                Variable
Variable         AGE         STATUS       INCOME      VOTEPCT      Constant
AGE            17.868
STATUS          1.733          0.241
INCOME          3.905          0.642        4.336
VOTEPCT       10.027          1.862        8.450       21.388
Constant       23.051          0.651       12.596       52.208         0.020
```

The Constant row holds the means of the variables selected by the other columns, and 1/N (where N is the number of subjects that the matrix is based on) in its diagonal (the right corner element). Each of the other cells contains the sum of the products of the deviations around the means for the variable pair specified by the selectors. The number of subjects on which each entry is based can be affected both by weighting and by missing data. However, all the available data are used to calculate each entry, and thus the N's for different entries will, in general, not be the same. The N appearing in the Constant diagonal is the minimum cell N, and all other entries are scaled so as to appear as if based on this N. The function PAIRN is available to produce the actual pairwise N's.

8.3.1 Covariance and correlation

A covariance matrix can be produced from a covariation matrix by eliminating the Constant row and column and dividing throughout by N 1. This is simply done, for a covariation matrix C, with the sequence

```
S _ (GENVEC 1 (SHAPE C)@'(1) 1)
C _ C@(LIST S S)/( 1/C@'(Constant Constant)  1)
```

which saves a selector for all but the last row or column in S, selects out the section of the matrix corresponding to variables other than Constant, and divides it by N 1, obtained from the Constant diagonal cell of the matrix.

A covariance matrix may be converted to a correlation matrix simply by dividing each element by the square root of the product of its diagonals (i.e. scaling the covariances in terms of the variances). Since a covariance matrix differs from a covariation matrix by only a factor of 1/(N 1), this procedure would also work for covariation matrices if the Constant row and column are removed. However, it is easier to use the function NORM, which normalizes a matrix by doing exactly this. Thus, we could obtain a correlation matrix from C with

```
(NORM C)
```

NORM will disregard rows with negative diagonal entries (as their square root is not defined), so this implicitly eliminates the last row of the matrix.

Rank-order correlations (Spearman's r) result from applying the same correlation procedure to variables that have been transformed by RANK to within-variable ranks.

Partial correlation coefficients are obtained from a covariation matrix using regression techniques and will be considered below.

8.3.2 Regression

The SWEEP function is the basis of all regression related statistics. SWEEP takes three arguments: the covariation matrix, an (optional) set of variables to be swept out, and an (optional) set of variables to be swept back in. Intuitively, sweeping a set of variables out of the covariation matrix corresponds to redistributing the variance associated with the swept variables so that it is removed from the unswept variables and divided up among the swept ones. The swept out variables are thus the independent variables of a regression against the unswept out (dependent) ones. Sweeping a set of variables back into the matrix distributes their variance back into the whole matrix. This provides a very flexible basis for all forms of regression analysis, including partial and multiple correlation.

8.3.2.1 Unstandardized regressions

Unstandardized regressions can be produced by the SWEEP function directly from a covariation matrix. For example, using the covariation matrix C from the example above, we can simultaneously produce multiple regressions for each of INCOME and VOTEPCT (the dependent variables) on AGE and STATUS (the independent variables) with

```
S _ (SWEEP C '(AGE STATUS))
```

which would produce

```
Sweep of Crossproducts of SURVEY DATA, Swept out: AGE Constant STATUS
          Variable
  Variable       AGE        STATUS       INCOME      VOTEPCT      Constant
  AGE           0.185
  STATUS         1.330       13.714
  INCOME        0.132          3.610        2.532
  VOTEPCT        0.622       12.199        3.047         4.911
  Constant       3.398       21.733       13.280        45.813        64.197
```

Notice that the sign of the diagonal elements indicates which variables have been swept out: the diagonal for a swept out (independent) variable is negative. Thus if SDIAG is the diagonal formed by

```
(TRANSPOSE S '(1 1))
```

the expression

```
IV _ (SEEK 'MINUSP SDIAG)
```

gives a selector for the independent variables in the regression and

```
DV _ (SEEK 'PLUSP SDIAG)
```

gives a selector for the dependent variables.

There are three types of information in a swept matrix: relations between the swept and the unswept variables, relations between the swept (independent) variables, and relations between the unswept (dependent) variables.

First, for each dependent variable, the regression coefficients for the independent variables in its regression equation are found in the intersection of its row and the columns corresponding to the independent variables (or vice versa, as the matrix is symmetric). For example, the 3-vectors of coefficients for the dependent variables INCOME and VOTEPCT can be selected by

```
S@(LIST 'INCOME IV)
S@(LIST 'VOTEPCT IV)
```

and their regression equations can be read off as:

```
INCOME is estimated as  0.132*AGE + 3.610*STATUS + 13.280
VOTEPCT is estimated as 0.622*AGE   12.199*STATUS + 45.813
```

Note that each regression equation has three terms, not just the two that were swept with the SWEEP command. This is because the original COVAR matrix was created with the Constant swept already (i.e., the entries are cross products of deviations from the variable means, not cross products of the raw scores). This is done purely for reasons of numerical stability, and the Constant term may be removed from a regression equation (by sweeping it in) in cases when a constant term is not wanted in the equation.

The great flexibility of this approach lies in the fact that other regressions may be computed from the output of the SWEEP operation. As it is itself a covariation matrix, albeit one with the variance redistributed in certain ways, it too can be swept to produce further regressions. For example, if we were dissatisfied with the INCOME regression shown above, and decided that the variable VOTEPCT should be added as an independent variable, we could simply compute

```
(SWEEP S 'VOTEPCT)
```

As this does not require the resweeping of AGE and STATUS, it is convenient to add or remove independent variables (by sweeping them out or in). The order of sweeping does not affect the result of the analysis.

The diagonal element for a dependent variable is the amount of variance left unaccounted for by the independent variables. Thus the residual sum of squares for INCOME is

```
SDIAG@'(INCOME)
```

This is more easily interpretable if it is scaled by its value *before* the sweep:

```
CDIAG _ (TRANSPOSE C '(1 1))
SDIAG@'(INCOME) / CDIAG@'(INCOME)
```

The last line gives the proportion of variance unaccounted for by the last sweep operation. That is just 1 R^2 where R is the multiple correlation between the predicted values from the regression equation and the observed values.

The next interesting set of questions is the relationship between the independent variables. If these are closely related to each other, then the unique contribution of each one to the overall prediction will be small. The diagonal element for an independent variable is the negative reciprocal of its unique variation in a different regression: that of this variable on all the other independent variables. This is a measure of "multicollinearity," or the interdependence of the independent variables in a regression. Like the residual sum of squares, it is more clearly interpretable if it is scaled by the variable's variance before the SWEEP took place, that is, by the corresponding element of CDIAG. This can be done simultaneously for all independent variables by

```
 1/(CDIAG@(LIST IV) * SDIAG@(LIST IV))
```

which produces a vector of 1 R^2s for the multicollinearity regression. The 1 R^2 figure gives the proportion of variance predicted by this variable which is not predicted by the other variables in the regression. If 1 R^2 for a predictor is 1.0, then that predictor is totally unrelated to all others and thus contributes maximally to explaining variation in the dependent variable. On the other hand, a 1 R^2 of 0.2 or less indicates that this independent variable has little effect of its own on any dependent variable; it is just a combination of other independent variables in the regression.

8.3.2.2 Standardized regression

If the SWEEP operations are applied to a NORMalized matrix rather than an unNORMalized one, the result is a standardized regression. The layout of the matrix is the same, but some of the interpretations are easier. The same entries in the matrix are the regression coefficients, but now they are standardized coefficients, also called b or *path* coefficients (when they are used to label lines of influence between variables in a path model). As the initial variance of each variable has been set to 1.0 by the NORMalization, the diagonal entries do not need to be scaled by the pre-SWEEP entries, as was previously the case. Now, the diagonal element for a dependent variable is the proportion of variance left unexplained by the regression, and the diagonal of an independent variable is now the negative reciprocal of the proportion of variance unique to that variable in the regression.

8.3.2.3 Regression residuals

The analysis of a regression model is not complete with the computation of the regression coefficients. Examination of the residuals from the regression often helps one to detect significant departures from the linear, additive model assumed by regression, and then to improve the model's fit to the data. Residuals can be computed by using the regression coefficients to calculate the actual predicted values of the regression. Consider our last example, where we derived a regression equation for INCOME. First, we compute a vector of predicted values for the dependent variable:

```
P _ (MPROD A@'((AGE STATUS)) S@'((AGE STATUS) INCOME))
   + S@'(Constant INCOME)
```

This expression extracts the matrix of raw scores corresponding to the independent variables AGE and STATUS, multiplies them by the corresponding coefficients, adds them up, and finally adds the Constant coefficient. (Recall that A was the data matrix from which the covariation matrix was originally produced.) The residuals are then produced by subtracting the predicted scores from the real ones, i.e.,

```
R _ A@'(INCOME)   P
```

In cases where either the residuals or the predicted scores have continuing interest, they can be added to the data array as a new variable.

The vector of residuals can be plotted against the predictor from the regression (here P), or against any other variable with the PLOT function. For example,

```
(PLOT R P)
```

plots the residuals generated above against the vector of values predicted by the regression. Plots of residuals against the regression predictor or independent variables often show departures from the form of relationship assumed by the regression model; they can also show the appropriate transformations or additions to the model to correct the problem. For instance, if a residual plot shows an upward- or downward-facing "U" shape, one or more independent variables may have a quadratic relationship with the dependent variable; the addition of a quadratic term in those variables to the equation may improve the model's fit. Anscombe and Tukey (1963) or Kruskal (1968) contain extensive discussions of the issues involved in plots and transformations of variables.

8.3.2.4 Partial correlations

Information on the relations between the dependent variables, in the form of partial correlations, is also available from the output of SWEEP. Partial correlations are simply ordinary correlations between variables that have had the effect of some common variation removed. Thus, NORMing a covariation matrix from which some set of variables has been swept out gives the partial correlations among the unswept variables, controlling for those swept. This can provide partial correlations controlling for more than one variable.

In a similar fashion, one can define a partial multiple correlation. Assume that there are two sets of variables, X and Y, and one is interested in the multiple correlation of X with some dependent variable Z after controlling for Y. The first step is to remove the variance associated with the set Y by sweeping those variables out of the matrix. This leaves a certain amount of variance common to X and the dependent variable Z, which can be measured by SWEEPing out X and looking at the proportional change in the variance of Z. This is an exact parallel to the sum of squares in ordinary regression. There the post-SWEEP variance of Z is compared with the total variance of Z; here it is compared with its variance after the removal of the variance associated with the set Y.

8.4 Classification based statistics

Classification based statistics are computed from data grouped according to the values of other observations on the same individual. Classifications in IDL are represented in the dimensional structure of an array, usually in its leading dimensions. Thus, the basic data structure for comparing the heights of men and women is a [2,*n*]-matrix of which SEX is the leading dimension and each row of which contains the heights of a set of either men or women. Summary statistics can then be computed for the two levels of the first dimension. Multi-way classifications are represented by more than one classifying dimension. The usual mechanism for constructing such classified data arrays is GROUP.

GROUP takes a matrix of classifiers or attributes and an array of data values, and arranges the data into classification cells as follows. Each column of the attributes matrix produces a new leading dimension in the output array. Thus, a single column (one variable) classifier induces a one-way classification, a two column classifier groups into a two-way table, and so on. The output is formed by slicing the data array along one dimension (usually the first) and grouping together all slices whose corresponding attribute rows are the same. The data values appear in the cell in classifier space "addressed" by that row. The trailing dimensions of the output are the original dimensions of the values array.

Computation of summary statistics for the cells of a classification is facilitated by the fact that the dimensions of classification are kept in the output of GROUP, so generic functions (such as MOMENTS and COUNTS) will automatically be applied *within* those cells.

If an attribute variable has a large number of different values relative to the number of observations, e.g., if it is a continuous measure, then it might be necessary first to recode it into intervals using TRANSLATE or ROUND. For example,

```
(ROUND A@'((TIME)) 20)
```

could be used to scale a column of response times into 20-second wide intervals.

8.4.1 Frequency statistics

Frequency or contingency tables are usually produced by GROUPing a weighting vector (often a vector of 1's) by an attributes matrix and then obtaining the sum of the weights within groups by COUNTS, which ignores any NILs in the grouped array.

For example, if a matrix A contained variables labelled SEX and VOTE, then

```
TAB _ (COUNTS (GROUP A@'((SEX VOTE)) 1) )
```

could be used to request the two-way cross-tabulation

```
Counts of Group of 1 by Selected Variables from SURVEY DATA
          VOTE
SEX          DEMOCRAT     REPUBLIC      OTHER
FEMALE            317          181         27
MALE              351          256         34
```

It is often informative just to look at a raw frequency table, to detect such things as underpopulated categories of some variable (they could be collapsed with other categories) and to notice, in the case when the input array was value labelled, the number of subjects omitted because of "wild scores" (i.e., values which were not assigned a value-label).

Tables can be scaled in different ways to make the trends they show more clearly visible. For instance, a two-way table could be displayed as a set of row proportions or as a set of column proportions. These are obtained in IDL by a combination of PLUS-reduction and KEEP:

```
(KEEP TAB 'VOTE) / (RPLUS (KEEP TAB 'VOTE))
```

yields the column proportions for TAB. The denominator is a vector with one element for each column in TAB giving the sum across rows for that column:

```
VOTE
        DEMOCRAT    REPUBLIC     OTHER
             668         437        61
```

The KEEP in the numerator expression indicates that the columns are to be withheld in the QUOTIENT extension. This means that each element of TAB will be divided by the sum corresponding to its column. The row proportions result simply by specifying that rows are to be kept. In either case, multiplying by 100 converts the proportions to percentages:

```
100 * (KEEP TAB 'SEX) / (RPLUS (KEEP TAB 'SEX))
```

Expressions of the same general form will produce proportions or percentages for margins in higher-dimensional arrays: the keeps indicating the desired marginals must appear in both the numerator and denominator to achieve proper alignment for the extension. Thus, for a 4-array A

```
(RPLUS (KEEP A 1 3 4))
```

gives the 1, 3, 4-plane of the marginal totals formed by summing across the second dimension, and

```
100 * (KEEP A 1 3 4) / (RPLUS (KEEP A 1 3 4))
```

is the corresponding percentaged array.

x^2 is the most widely-used test of significance for contingency tables. It allows one to specify an array of "expected values" against which the values in the observed array are tested for a significant departure. By varying the expected values, many possible null hypotheses can be tested, not just the usual one of no relationship between the variables (Goodman, 1972). The x^2 statistic is a summation of terms called "x deviates", one from each cell of the table. A x deviate is the squared difference of the observed and expected frequencies scaled by the expected frequency. Often, examination of the table of x deviates, or even the raw deviates (observed minus expected for each cell), is helpful. One can detect patterns of positive and negative deviation in the various regions of the table, or can note which cells contribute large amounts to x^2 by showing large x deviates.

The task of computing a x^2 for a table thus involves two steps: generating the expected values and calculating the x deviates. As an example, consider the hypothesis of equal counts in all cells of some two dimensional array OBS. The expected value table can be obtained by

```
EXP _ (RESHAPE (RPLUS OBS)/(RTIMES (SHAPE OBS)) (SHAPE OBS) )
```

(RTIMES (SHAPE OBS)) is the number of cells in OBS so (RPLUS OBS)/(RTIMES (SHAPE OBS)) will be the count per cell if the total count of OBS is evenly distributed over its cells. Reshaping this number to the shape of OBS gives us the array that we would have under the hypothesis of equal distribution. Alternatively,

```
(MOMENTS OBS)@'(Mean)
```

could be used to compute the average cell count. With the observed array OBS and the expected array EXP, we can compute x^2. The most straightforward method is simply to write

```
(OBS EXP)^2 / EXP
```

which subtracts the two arrays, squares the result, and divides it by the array of expected values, producing an array of x deviates. A correction for continuity is usually applied if (REDUCE EXP 'MIN) is less than 5. The appropriate formula is then

```
((ABS OBS EXP) 1)^2 / EXP
```

The total x^2 may be found by PLUS-reducing this deviate array, and its probability may be obtained from the x^2 distribution.

The usual expected value array for bivariate independence in a two dimensional table, which takes into account the observed marginals of the variables but no association between them, is computed using matrix product. If OBS is a matrix, then

```
R _ (RPLUS (KEEP OBS 1))
C _ (RPLUS (KEEP OBS 2))
EXP _ (MPROD R C) / (RPLUS OBS)
```

will compute the cell-wise expected values. First, the row and column totals are calculated. Their matrix product is then divided by the grand total. x^2 can then be computed using the techniques described above.

8.4.2 Measure statistics

The previous section considered comparisons between groups when the only information available was the frequency of occurrence of events or subjects with certain attributes. We now turn to analyses based on measures of various other properties. The first thing to examine about a set of measure values is the frequency with which each value appears in the set. The frequency distribution can be produced by GROUPing the number 1 using a vector of measured values as the classifier, and counting the result with COUNTS. If the values are in a multi-dimensional array instead of a vector, the dimensional structure must first be eliminated, by RESHAPEing the array down to a vector. Thus,

```
(COUNTS (GROUP (RESHAPE X) 1))
```

will produce a vector of frequency counts for the various values in X. A weighting vector of the appropriate length can be used instead of 1 to take sampling biases into account. Applying the function HIST to the vector produced by COUNTS will generate a graphical display of the distribution.

After examining the frequency distribution, it is common to analyze the differences in measured values between groups defined by some classification design. The appropriate test statistic and approach for this problem depend on the nature of the classification, the level of measurement of the measure involved, and assumptions about its underlying distribution.

8.4.2.1 Non-parametric methods

The non-parametric methods impose the weakest requirements on the measure in that it need not be interval-scaled and its distribution need not be normal. If the criterion variable is only ordinal scaled, then one of the rank-order statistics should be used. The function RANK computes the set of ranks on which many statistics are based. It returns an image of its arbitrary array argument with each cell containing the rank of the corresponding observation within the set of values found in the array.

The Kruskal-Wallis one-way analysis of variance by ranks is the non-parametric equivalent to the ordinary one-way analysis of variance. It is appropriate when there is a single measure for subjects in a set of independent groups. It requires only ordered data and makes no assumptions about the distribution of the underlying variable. The Kruskal-Wallis *H* statistic is obtained by assigning to each subject the rank of his score within the set of scores for all subjects pooled together. The MOMENTS of the ranked scores are computed and submitted to ANOVA. The test statistic is not the *F* that ANOVA produces, but a ratio of quantities easily obtained from the ANOVA. This ratio may be referred to the x^2 distribution.

Assume that A is a vector (or [n,1]-matrix) of grouping attributes for a set of scores in the vector S. Then the ranks over all scores grouped by A are obtained by

```
R _ (GROUP A (RANK S))
```

Alternatively, if G is a matrix of already grouped scores, the ranks within the pooled groups are

```
R _ (KEEP (RANK (LEAVE G ALL)) 1)
```

In either case, the first dimension of R is kept, so that the desired ANOVA table is simply

```
AN _ (ANOVA (MOMENTS R))
```

The Kruskal-Wallis *H* statistic is defined by

$$H = SS_{groups}/MS_{total}$$

where MS_{total} is obtained by pooling the group and error lines in the table. Thus

```
H _ AN@'(2 SumSq) / (RPLUS AN@'((2 3) SumSq))/(RPLUS AN@'((2 3) df))
```

When the number of subjects in each group is large, this statistic is distributed as x^2 with K 1 degrees of freedom, for K the number of groups. The degrees of freedom is either

```
(SHAPE R)@'(1)   1    or    AN@'(2 df)
```

In the special case where there are only two groups, the Kruskal-Wallis test is equivalent to the better known Mann-Whitney *U* test. The normal approximation for the *U* test is simply (SQRT H).

Other non-parametric tests are appropriate when there are two or more observations on the same individual. To determine whether two paired observations differ, the sign-test and Wilcoxon signed-ranks test are used. The sign-test requires only ordinal measurement. Subjects are classified into groups depending on the sign of the difference of the two observations, and the numbers of subjects with positive and negative signs are compared. In effect, a one-way contingency table is constructed, using the sign of the difference as a classifier. If V1 and V2 are the two observation vectors and S is the translation table

0	0	NIL
NIL	0	1
0	NIL	1

then

```
(COUNTS (GROUP (TRANSLATE V1 V2 S) 1)))
```

produces a vector of counts showing the number of positive and negative difference signs. This may be referred to the binomial distribution to determine its probability level. In this example, the matrix S translates tied scores to NIL, negative differences to 1, and positive differences to 1.

The Wilcoxon signed-ranks test assumes ordered metric measurement (i.e., assumes that it makes sense to rank a set of differences between scores), which is stronger than ordinal-level measurement. Subjects are classified by the sign of the difference just as for the sign-test. The classified value is not a subject's weight, however, but the rank of the absolute value of the difference between the two observations for that subject:

```
G _ (GROUP (TRANSLATE V1 V2 S) (RANK (ABS V1 V2))))
```

The test statistic is either the sum of the positive ranks or the sum of the negative ranks, whichever is less. The expression

```
(REDUCE (COUNTS G) 'MIN)
```

provides the appropriate number. If the number of subjects (the length of V1) is less than 25, the probability of this statistic may be determined from published tables (e.g. Siegel, 1956). Otherwise, a normal approximation may be used.

8.4.2.2 Parametric methods (ANOVA)

The basis for analysis of group differences on an interval scaled measure is a MOMENTS table containing the zero through second moments of the observations within each group. The function ANOVA performs an analysis of variance on such a table, providing both *F* and *t* tests.

Consider an experiment in which subjects are asked to decide whether visual patterns are the same. The patterns are presented under two lighting conditions in four different colors, and subjects' response times are recorded. A single observation for each of 40 subjects is entered in a subjects by variables matrix A, with variables labelled LIGHT, COLOR, and TIME. The following expression computes the table of moments needed for determining the effect of the LIGHT and COLOR conditions on TIME scores:

```
M _ (MOMENTS (GROUP A@'((LIGHT COLOR)) A@'(TIME)))
```

This causes the TIME values to be cross-classified by LIGHT and COLOR. The MOMENTS are then computed within each level of the classification (MOMENTS only operates over the unkept dimensions), and the result is saved as M. M is a [2,4,3]-array containing the within-cell N, mean, and variance for each cell in the classification:

```
              Moments of Group of Variable TIME from Pattern
              Experiment by Selected Variables from Pattern
              Experiment keeping LIGHT and COLOR

     LIGHT = LOW
              Moment
      COLOR              N            Mean      Variance
         RED         5.000           6.052          .380
        BLUE         5.000           5.802          .397
       GREEN         5.000           5.976          .373
      ORANGE         5.000           6.520          .391

     LIGHT = HIGH
              Moment
      COLOR              N            Mean      Variance
         RED         5.000           6.454          .379
        BLUE         5.000           6.200          .382
       GREEN         5.000           6.398          .393
      ORANGE         5.000           6.656          .386
```

The expression

```
(ANOVA M)
```

applies ANOVA to test the significance of the differences between the means in this table. This yields the two-way analysis of variance

```
                Anova of Moments of Group of Variable TIME from
                Pattern Experiment by Selected Variables from Pattern
                Experiment keeping LIGHT and COLOR

          Column
       Source    SumSq        df          MS           F          p
       Gnd-mean  1516.469     1.000       1516.469     3936.210   .000
          LIGHT  2.911        1.000       2.911        7.555      .010
          COLOR  1.800        3.000       .600         1.557      .219
            L*C  .139         3.000       .046         .120       .948
          Error  12.328       32.000      .385         NIL        NIL
```

In this simple design, the error term is the appropriate denominator for testing all the effects, and the main effect for LIGHT is the only one significant beyond the .05 level. Other denominators would be used for more complicated designs. Thus, if the two LIGHT conditions are considered as representatives of a much larger set of illumination levels and if the results of this experiment are to be generalized to that larger set, then LIGHT should be specified as a random factor in the design:

```
        (ANOVA M '(LIGHT))
```

The only change between this and the preceding analysis is that the mean-square for the LIGHT by COLOR interaction becomes the correct denominator for testing the COLOR main effect, and the probability level for that effect drops from .219 to .03.

GROUP, MOMENTS, and ANOVA can be composed to analyze a wide range of experimental designs. For hierarchical nesting of one factor inside another, the GROUPing must be done in two stages to produce what looks like a moments table for a cross-classification. Suppose, for example, that there is an additional factor in the pattern discrimination experiment: the red and blue stimuli are bigger than the green and orange ones. The factor COLOR is thus nested within SIZE. The moments table must have an additional SIZE dimension with two levels (large and small). The effective number of levels on the COLOR dimension is reduced from four to two, because there are only two colors within each size.

The first step is to GROUP the subjects by the non-superordinate factors in the hierarchy, LIGHT and COLOR, just as above. Then, however, the levels on the COLOR dimension are grouped by an attributes vector placing each of the colors in one of the SIZE levels:

```
        G _ (GROUP '(1 1 2 2)
                      (GROUP A@'((LIGHT COLOR)) A@'(TIME))
                      'COLOR)
```

The attribute list classifies the first two colors into level one on the new size dimension and the second two colors into level two. The result is then submitted to MOMENTS to obtain the desired [2,2,2,3] moments table. Alternatively, MOMENTS can be applied to the first grouping, giving the array M as above. The second GROUP is then performed on the moments instead of the raw data:

```
        G _ (GROUP '(1 1 2 2) M 'COLOR)
```

Either way, the result is given to ANOVA, with a third argument specifying that COLOR is nested within the size dimension:

```
        (ANOVA G NIL '((COLOR 1)))
```

The size dimension is indicated by number here, since we have not taken the trouble to add the SIZE label to G. The nesting specification suppresses the COLOR by SIZE interaction effects that would otherwise be included as sources of variation. They are pooled into the appropriate COLOR effects instead.

Repeated-measures designs, i.e., those in which observations for more than one condition are recorded from a single subject, require no special mechanism in IDL. The nesting and crossing relationships between the within-subject factors are handled simply by applying GROUP to the variable dimension of the subjects by variables matrix. The between-subject factors are treated in a similar way, the only difference being that the subject dimension itself is considered to be a factor in the design nested within all the between-subject factors. The arguments to ANOVA must specify this implicit nesting relationship and should further indicate that subject is a random factor.

Suppose COLOR and SIZE are within-subject, rather than between-subject, factors, with COLOR still nested within SIZE. That means that four different response times are recorded for each subject, one for each of the four presentation colors, so the data matrix has five variables (including the LIGHT attribute column). The within-subject classification is accomplished by grouping the variables into color categories and then the colors into sizes:

```
C _ (GROUP '(1 2 3 4) A@'((2 3 4 5)) 'VARIABLE)
SC _ (GROUP '(1 1 2 2) C 1)
```

The first line explodes the four response time variables into a new dimension with four levels, yielding the [4,*n*,1]-array C (where *n* is the number of subjects). The second line groups the levels on the first (COLOR) dimension into the appropriate size classes. Again, dimension numbers are used instead of explicitly labelling the results at each step. A third GROUP takes care of the between-subject factor:

```
LSC _ (GROUP A@'((LIGHT)) SC 'SUBJECT)
```

The desired moments table may be constructed by

```
M _ (MOMENTS (KEEP LSC 'SUBJECT))
```

where the KEEP is needed to retain SUBJECT as a factor in the result. Finally, the analysis of variance is computed by

```
(ANOVA M '(SUBJECT) '((SUBJECT LIGHT) (3 2)))
```

The second and third dimensions of L correspond to color and size respectively, so the list (3 2) correctly specifies the nesting relationship.

These examples illustrate how complex experimental designs may be analyzed. The same functions provide the much simpler comparisons for which a *t* test is normally utilized. The *t* test for independent samples is equivalent to a one-way analysis of variance. A [2,3]-matrix of moments for the two groups is constructed and given to ANOVA. The *F* statistic for the group effect is the square of the desired *t* value, and the p column of the table gives its two-tailed probability. If the data for the two groups are in separate vectors V1 and V2, moments can be computed separately and combined together by LIST:

```
(ANOVA (LIST (MOMENTS V1) (MOMENTS V2)))
```

The matched-pairs *t* test is equivalent to a repeated-measures analysis of variance. As above, the two within-subject variables are exploded into a separate dimension, and subjects are included as a random factor in the design. The *F* statistic for the two variables is the square of the desired *t* value.

The array operators may be used to examine effects in an analysis of variance design more directly, by successive subtraction of common additive factors from the array of means. Initially, the grand mean is subtracted out, leaving an overall mean of zero. Each cell then contains the effect for that cell of the design. If M is a moments array for some design, this can be accomplished by

```
E _ M@'(Mean)                   to compute the array of means
G _ (MOMENTS E)@'(Mean)         to compute the grand mean, and
E _ E   G                       to compute the array of cell effects
```

Here, G is computed as the unweighted average of the cell means. The grand mean using the cell N's as weights may be computed with POOL by

```
G _ (POOL M)@'(Mean)
```

This elimination procedure can be carried out for higher order effects, and in this way, successive effects can be removed from the array. Assume the three factors in the design are labelled A, B, C, and that the AB interaction and the B and C main effects are significant. The AB effects matrix and the vectors of A, B, and C effects might be constructed. These can then be subtracted from E in turn, and the resultant patterns examined for clues as to the direction and locations of effects. When the results look "random", i.e., all numbers about the same size and no consistent patterns of + or   signs, then it is time to stop. The first steps of such a process might be

```
A _ (RPLUS (KEEP E 'A))
B _ (RPLUS (KEEP E 'B))
C _ (RPLUS (KEEP E 'C))
AB _ (RPLUS (KEEP E 'A 'B))
(PPA AB   B)
X _ (KEEP E 'A 'B )   AB
(PPA (KEEP X 'C)   C)
```

which calculates the A, B, C and AB effects, prints the AB effect with the significant B effect subtracted out, and then prints the array with both the AB and C effects removed. If this last array shows interesting residuals, the ABC interaction could be examined, even if it is known not to be significant.

The EMS function provides the coefficients for the variance components included in the expected value of the mean squares for every row of an analysis of variance. It may be useful for theoretical explorations, for constructing quasi-*F* values when no legitimate test exists for an effect, and for determining how to remove effects from a model by pooling. An example, for a simple 2 x 4 crossed design with the first factor fixed and the other random, is

```
(EMS '(2 4) '(1))
```

This produces

```
          Coeff
Source      Factor1      Factor2        1*2
Factor1           4            0          0
Factor2           0            2          1
1*2               0            0          1
```

The proper denominator for testing the effect of a particular source of variation *i* is determined as follows: If row *i* is all zero except for the diagonal element and if the design has a a within-cell error term, the error is the appropriate denominator. Otherwise, if a row below the *i*th row is identical to the *i*th row in all respects except that its *i*th element is zero, that effect is the correct denominator for testing row *i*. According to these rules, the error term is the correct test for Factor1 and the 1*2 interaction, while the 1*2 interaction is the proper denominator for the Factor2 test.

If an appropriate row does not exist in the EMS table, it may be possible to add or subtract two or more rows to form an estimate of the denominator mean-square. Principles for pooling in this way and for correcting the degrees of freedom of the resulting quasi-*F* statistic are found in Winer (1971).

Appendix B

The IDL Library

Although the user of IDL is encouraged to use the flexibility of the system to construct his own analysis tools, the IDL system provides a library which contains some useful combinations of IDL operators as functions. Often the real benefit of these functions is simply that they provide more detailed labelling than the user immersed in a specific problem would be willing to provide or that they make use of operator combinations which, although simple, would require some amount of thought to derive.

As they are expressed as IDL programs, the user is encouraged to study the text of these definitions as indication of how the IDL primitives may be combined together to produce new functionality. The descriptions here are correspondingly brief.

The Library may be LOADed from <IDL>LIBRARY (source code for perusal) and <IDL>LIBRARY.COM (compiled code for unreflective execution). This listing is for convenience only, and the contents of the Library are subject to change without notice.

apool[atable;lines]
: Pools the specified **lines** of the ANOVA table **atable**, producing a vector of the SS, df, and MS.

freq[a;wt]
: Computes frequency distribution of the values in an arbitrary array **a**, weighted by **wt**. Not extended, because there might be different numbers of values for different slices.

freqhist[a;file;wt]
: Produces frequency histograms of the values of **a**, weighted by **wt**, for each level of its kept dimensions. Returns **file**.

friedman[m]
: Computes the Friedman 2-way analysis-of-variance by ranks. Columns are the repeated measure. The sign test is essentially the special case where there are only two columns. Produces the Chi-square/r statistic, which may be looked up in Segal, or referred to the chi-square distribution with C-1 degrees of freedom, for C the number of columns.

hmean[a]
: Computes the harmonic mean of the non-NIL elements of **a**.

kruskal-wallis[values;attribs]
: Computes the Kruskal-Wallis analysis-of-variance by ranks. Produces the H statistic, which may be looked up in Segal, or referred to the chi-square distribution with K-1 degrees of freedom, for K the number of groups. **values** can be a grouped array, with the treatment dimension the only kept dimension, or it can be a vector of values, with the classification indicated by **attribs**.

mann-whitney[values;attribs]
: Essentially the special case of the Kruskal-Wallis analysis-of-variance by ranks where there are only two groups.

nfroms[m]
: Returns the N on which a swept cross-product matrix **m** is based, no matter what the pattern of sweeping has been, by sweeping in all the variables that are swept out. Doesn’t work if anything has been swept out or in twice.

pct[a;keeps]

Returns an image of **a** with elements indicating the percentage that the corresponding element **a** is of the margin formed by collapsing across all dimensions not in **keeps**. For example, for a matrix, **keeps**=1 gives row percentages; **keeps**=2 gives column percentages.

regress[pre;post;dv;n]

Produces a more-or-less standard regression table for the dependent variables in the selector **dv**, given a pre-swept matrix **pre** and a **post**-swept matrix **post**. If **dv**=NIL, then tables for all unswept variables in **post** are constructed. If **n** is not NIL, then it is the total number of observations on which the regression should be based. Otherwise, the **n** is computed using NFROMS. The user should provide the **n** if the constant is not present in the array, e.g., because it was NORMed.

rmse[x;y]

Computes the root-mean-square-error of the conformable arrays **x** and **y**.

signtest[v1;v2]

Produces a frequency table to which chi-square can be applied for the sign-test.

wilcoxen[v1;v2]

Produces the Wilcoxon statistic to be looked up in tables (for N<25), or computes the correct probability using ANOVA.

yhat[m;sw;dv]

Computes predicted values for variables in **m**, given the swept matrix **sw** and the dependent variable selector **dv**. If **dv** is NIL, then predictors for all unswept variables in **sw** are computed.

zscore[a;mom]

Computes the image of **a** under the zscore mapping. **mom**, if given, is the MOMENTS table for **a**. Otherwise, zscore will compute them for itself.

Appendix C

Global variables and their initial values

The following global variables are defined when IDL is started. Some of them are designed for the user to change, so as to affect the behavior of various IDL functions. Others are just defined to provide the user with an easy way of specifying some value as input to a function. The latter include ALL and the mathematical constants, and cannot be meaningfully changed by the user.

Variable	Initial value	Section
ALL	ALL	2.1.1
E	2.718282	3.2
HISTRPTLINES	5	6.3
LABELPRINTFLAG	4	2.4
PI	3.141593	3.2
PLOT.AXIS.RATIO	0.6	6.4
PRECISION	(4 3)	2.4
ROWLABELWIDTH	8	2.4

Appendix D

Summary list of IDL functions

This appendix lists the functions available in the IDL system, giving their arguments and a brief description of their behavior. The functions are described in the notation used in the body of the manual. The annotation "no-spread" means that the function will accept any number of arguments.

adjoin[vector...]

> No-spread. Produces a new vector formed by joining its arguments together end to end.

anova[mtable;random;nesting]

> Produces a matrix containing a summary table for the analysis of variance of moments array **mtable**. **random**, if given, specifies the random factors (dimensions of **mtable**) and is used for constructing appropriate *F*-tests. **nesting** is an optional specification of the nesting relations between the independent variables. It is a list of lists, each of the form (nested-factor nesting-factor$_1$ nesting-factor$_2$ ...).

assign[target;source]

> The IDL assignment operator. **target** must be a selection on some array, and that array will be side-effected. The infix operator _ translates to *assign* if an array selection expression appears as its left-hand operand.

at[a;sltr]

> The IDL selection operator. Produces an array as output which is a window onto the section of **a** described by **sltr**. **sltr** may be a list of length no greater than the number of dimensions of **a** to specify a selection of **a**'s values, or a label or code selector. The infix operator @ is associated with *at*.

code[lev;val]

> Constructs a code selector to be given to *at*. If **lev** is NIL, the selector can be used to reference the value-labelled dimension of the array to which it is applied by *at*. Otherwise, if **val** is NIL, the selector will reference the complete codebook associated with level **lev** on the array's value-labelled dimension. If **val** is a non-NIL literal atom, it will reference the value associated with it on level **lev**. Finally, if **val** is a scalar, it will reference the label associated with that value.

copy[x]

> The Lisp *copy* function generalized to copy IDL arrays (which may be the value of **x** or may appear at arbitrary levels in the list-structure **x**). Assigning into a *copy* of an array will not affect the original.

counts[a]

> A generic function that returns the sum of the elements of **a** just like *rplus*, except that it skips over NIL's. Often used in conjunction with *group* to produce contingency tables from data arrays.

covar[a;wt]

> Builds the covariation matrix (sums of mean centered cross products) for the columns of matrix **a**, using **wt** as a weighting vector. Returns a symmetric matrix, whose last row represents a Constant variable (with value always 1) that has been swept out, so that the last row has the means of the columns of **a**, and 1/n in the last cell.

deal[n]

Returns a vector containing a random permutation of the integers from 1 to **n**.

dumpidlarray[a;file]

Writes a symbolic expression on **file** (primary output file if **file** is NIL) from which *readidlarray* can reconstruct an array equivalent to **a**. If **file** is not an open file, it is opened, the expression is printed, and then **file** is closed.

eapply[fn;expects;args]

Extending form of *apply*. Applies **fn** to each section of the decomposition of **args** according to **expects**, a list which gives, for each argument of **fn**, the dimensionality expected of that argument.

eapply*[fn;expects;args...]

No-spread. Extending form of *apply**. Applies **fn** to each section of the decomposition of **args** according to **expects**.

elementtype[a]

Returns the type (INTEGER or FLOATING) of the elements of the array **a**.

ems[nlevels;random;nesting]

Produces an expected mean squares coefficient table for an analysis of variance design with factor levels given by **nlevels**, the factors in **random** considered random, and the nesting relationships given by **nesting**. **nlevels** can be the moments table itself, or just the shape of its classification space.

extend[fn;expects]

Modifies the function **fn** so that it will automatically extend across array arguments according to **expects**.

format[a]

Returns the format (either FULL or SYMMETRIC) of array **a**.

fprob[f;dfnum;dfden]

Returns the probability of *F*-value **f**, with degrees of freedom **dfnum**, **dfden**.

genvec[initial;end]

Generates a vector of the numbers from **initial** to **end**. If **initial** is a two element vector, it is taken as the first two terms of the series (i.e., it specifies an increment). Thus, (GENVEC '(3 5) 11) generates the odd integers from 3 to 11.

group[attribs;values;dim]

Constructs an array of m+n dimensions, where m is the number of columns of the matrix **attribs**, and n the dimensionality of **values**. **dim** indicates which dimension of **values** is to be nested within the classification induced by **attribs**. The values in each row of **attribs** are used as a subscript for an m-dimensional array whose extents are given by the number of distinct values found in the corresponding column of **attribs**. The output is formed by grouping all **dim** dimension planes of **values** corresponding to equal rows of **attribs** in the m-space location addressed by that row. In effect, *group* places the **dim** dimension planes of **values** within the cells of the classification design represented by **attribs**. If **values** is an integer, it is interpreted as a constant vector of that integer with length the number of rows of **attribs**. If NIL, it is considered to be one, which produces an object from which *counts* will compute a contingency table.

hist[v;file]

Prints a histogram for **v**, a vector, to the primary output file, or to **file** if given.

Suppresses multiple identical lines whenever their number exceeds the value of HISTRPTLINES. If **file** is not an open file, it is opened, the histogram is printed, and then **file** is closed.

idlarray[data]
: Produces an IDL array of arbitrary dimensionality from a list-structure description of its properties. The list **data** begins with an optional title followed by a representation of its shape and labelling, an optional format, and a list of values to be filled into the array in row-major order.

idlarrayp[a]
: Returns T if **a** is an IDL array, otherwise returns NIL.

idlmatrix[data]
: Produces a two-dimensional IDL array from **data**, a list structure describing its labels and values. **data** is a list each element of which is a list representing one row of the output matrix. The row specifications may be optionally preceded by one list of the form (TITLES title-string dimension1-label dimension2-label), and another of the form (LABELS level1-labels level2-labels ...). The level labels are for levels on the second dimension of the output. Each row-list is a sequence of data values optionally preceded by the labels for that first-dimension level. The labels for a non-codebook level consist simply of a literal atom; a codebook is specified in the format (level-label codepair$_1$ codepair$_2$ ...).

invert[m]
: Produces the matrix inverse of **m**. Equivalent to (SWEEP M ALL).

keep[a;dims...]
: No-spread. Produces a copy of **a** with **dims** added to its kept dimensions. If **dims** is the literal ALL, then all the dimensions of **a** will be kept. If no **dims** are specified, a vector of the currently kept dimensions is returned.

label[dim;lev]
: Constructs a label selector to be given to *at* that will reference the label for dimension **dim** (if **lev** is NIL) or the label for level **lev** on dimension **dim**.

leave[a;dims...]
: No-spread. Produces a new array from **a** with **dims** removed from its kept dimensions. If the first of **dims** is the literal ALL, then all of its kept dimensions will be eliminated.

listarray[a]
: Converts IDL array **a** into a list structure from which *idlarray* can reconstruct the original.

listmatrix[m]
: Converts IDL matrix **m** into a list structure from which *idlmatrix* can reconstruct the original.

mprod[a;b]
: Matrix product of **a** and **b**. If they are vectors, *mprod* coerces them in the obvious way.

moments[a;wt;m]
: Returns a vector of length **m**+1 containing the number of non-NIL values (moment 0) and the first **m** moments of array **a**, using **wt** as a weighting array. The first moment is the mean, the second is the variance, the third is closely related to the skew, etc.

If **m** is NIL, it is defaulted to two and the N, mean, and variance are computed.

norm[m]
Norms matrix **m** by dividing each entry by the square root of the product of its basis (diagonal) elements.

nprob[z]
Returns the one-tailed probability of a normal deviate of size **z**.

order[v;cfn]
Returns a permutation vector which will order vector **v** by comparison function **cfn**. If **cfn** is not specified, *lessp* is used.

pairn[a;wt]
Returns a matrix of the pairwise N corresponding to a covariation matrix on **a**, using **wt** as a weighting vector.

plot[y;x;file]
Plots the vector **y** against the vector **x** (one through length **y**, if not given) on the primary output file, or **file** if specified. The axes are scaled so that they are approximately the same size. This scaling is controlled by the global variable PLOT.AXIS.RATIO. If **file** is not an open file, it is opened, the plot is printed, and then **file** is closed.

plusp[x]
A predicate which is T if the scalar **x** is greater than zero.

pool[mtable]
Collapses **mtable** into a 3-vector of the pooled cell N, mean, and variance. Effectively removes factors from a moments array.

ppa[a;file]
Prints the array **a** on the primary output file, or **file** if specified. If **file** is not an open file, it is opened, the array is printed, and then **file** is closed. The precision of numbers in the table is determined by the global variable PRECISION, a list specifying the number of digits to appear to the right and left of the decimal point. The initial setting is (4 3) so that numbers are printed in the format rrrr.lll. The global variable ROWLABELWIDTH is an integer determining the number of columns to leave for the labels on rows (initially 8).

randn[mean;stdev]
Returns a single number randomly sampled from the normal distribution with mean **mean** (default 0) and standard deviation **stdev** (default 1).

rank[a]
Ranks the array **a** in ascending order. Result will be integer unless floating is necessary to resolve ties. Cells containing NIL are ignored in computing the ranks, and the corresponding cells in the value will be NIL.

readidlarray[file]
Constructs an IDL array from an expression read from **file** (primary input file if **file** is NIL). If **file** is not an open file, it is opened, the expression is read, and then **file** is closed. *dumpidlarray* produces an expression of the appropriate form.

reduce[a;fn;startval]
Applies function **fn** left associatively to the elements of **a**. If **startval** is given, **fn** is

first applied to **startval** and **a**$_1$, otherwise to **a**$_1$ and **a**$_2$. Thus, (REDUCE ARY 'NCONC1 NIL) returns a list of the elements of ARY. *reduce* distinguishes the situation where **startval** is specified as NIL from the case where it is not specified at all, so (REDUCE ARY 'NCONC1) will ignore **a**$_1$ as it is not a list.

reshape[a;newshape;newformat]
Reshapes **a** to an array of shape **newshape** and format **newformat** (optional, default FULL). The elements of **a** are put into the new array in rowmajor order. If **newshape** is NIL, **a** is simply flattened into a vector.

round[val;interval]
Rounds **val** to the nearest integral multiple of **interval**. Thus, (ROUND 1.45 .5)=1 and (ROUND 785 100)= 800.

rplus[a]
Equivalent to reduce[**a**;quote[plus];0].

rtimes[a]
Equivalent to reduce[**a**;quote[times];1].

same[a;b]
Returns one if eqp[**a**;**b**], zero otherwise.

scalarp[s]
Returns T if **s** is a scalar, otherwise NIL.

seek[sought;vec]
sought can be a vector or a one argument function. Returns the (ordered) vector of indices of elements of **vec** which are found in, or which satisfy, **sought**.

shape[a]
Returns a vector giving the shape of array **a**.

shift[v;shift;fill]
Returns the vector formed by shifting vector **v** **shift** places (positive shifts right; negative shifts left) shifting in elements from the opposite end of **fill** to replace moved elements. **fill** can be a scalar or a vector. If **fill** is omitted, **v** is "rotated", i.e., the element shifted out one end is shifted in at the other.

sweep[m;outvars;invars]
Sweeps out variance components of matrix **m** corresponding to **outvars** (a selector for the second dimension of **m**); then sweeps in components corresponding to **invars**.

title[]
Returns a selector that can be given to *at* to reference the title of an array.

tprob[x;df]
Returns the probability of a *t*-value of **x** with degrees of freedom **df**.

translate[s;table;default]
Translates the scalar **s** according to the matrix **table**. If **table** has one column (or is a vector), the index of (the first) **s** in **table** is returned. If it has two columns, the second element of the row whose first element is **s** is returned. If three columns, the third element of the row such that **s** is between the first and second elements is returned. In the three-column case, a NIL in the first two columns only matches a NIL **s**; otherwise, a NIL in column 1 is interpreted as # while a NIL in column 2 is

interpreted as +#. If no match is found, **default** is returned if it is specified; otherwise the value is **s** itself.

transpose[a;perm]
: Transposes **a** by mapping each of its dimensions onto the dimension of the output given by the corresponding element of **perm**. Two dimensions that are mapped onto the same dimension of the output are represented by their joint diagonal. If **perm** is not given, the dimensions are reversed.

In addition to the above, the mixed-arithmetic operators *plus*, *times*, etc., and most of the other arithmetic functions in Interlisp have been extended to apply element-wise across arrays. The infix operators +,*, ,/ have been mapped onto the mixed-arithmetic operators instead of the integer operators as in standard Interlisp.

IDL implements a new kind of Lisp function, an ELAMBDA, to simplify the task of defining extended functions. The key-word ELAMBDA is used instead of LAMBDA in the function definition, and the size of the expected object can be associated with each argument name. For example,

```
[DEFINEQ (FOO (ELAMBDA ((M MATRIX) (V VECTOR)) (ADJOIN M (KEEP V 1]
```

which adds V as a new column to M, is equivalent to

```
[DEFINEQ (FOO (LAMBDA (M V) (EAPPLY* '(LAMBDA (M V) (ADJOIN M (KEEP V 1)))
                                     '(MATRIX VECTOR)
                                     M V].
```

The IDL system defines the file package command IDLARRAYS for dumping IDL arrays via *makefile* onto ordinary Lisp files in *load*-able format. (The Lisp ARRAYS command only dumps regular Lisp arrays.)

IDL automatically opens a Lisp DRIBBLE file at the beginning of each session, which maintains a transcript of all interactions with the system. This file is named IDL.TYPESCRIPT on the login directory. It is a temporary file, and will disappear when the user logs off Tenex; it must be explicitly renamed or copied to another file if the contents are to be preserved.

Finally, the facilities of <LISPUSERS>SHOW are included in IDL. Thus, the command SHOW may be given to the Interlisp executive to cause the value of the last expression typed in to be pretty-printed. This augmentation to the Lisp system greatly improves the convenience of IDL interactions. For more details, see <LISPUSERS>SHOW.TTY.

Appendix E

Technical considerations

This appendix presents information that is only of concern to the sophisticated user, and which would disrupt the flow of other parts of the manual were it not separated out.

IDL-Lisp interface

As mentioned in Chapter 3, IDL redefines the Lisp arithmetic and comparison functions in order to substitute functions with the appropriate extensions. However, the Lisp arithmetic and comparison functions are still available as functions of the form fn.LISP for some renamed function fn. The functions whose definitions have been changed are: PLUS, TIMES, DIFFERENCE, QUOTIENT, SQRT, EXPT, LOG, SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN, ARCTAN2, EQP, GREATERP, LESSP, REMAINDER, MAX, MIN, MINUS, MINUSP, ABS, GCD.

A possible difficulty that faces the user who intermixes IDL and Lisp functions in a program is that IDL scalars may not be Lisp numbers. That is, if V is a vector, V@'(3) may be neither FIXP nor FLOATP, but an object of type ARRAYFRAME. These objects will be handled correctly by all Lisp's arithmetic functions (even those that have not been redefined), but FIXP, FLOATP, SMALLP, and NUMBERP will not be true of them. SCALARP will be true of them, however. The equivalent of NUMBERP is (AND (SCALARP X) (NOT (EQP X NIL))). Note also that what appears to be a small integer might actually be a pointer to a large number box, so that EQ tests should be used cautiously.

Writing efficient functions in IDL

The efficiency gained by following these principles will ordinarily be negligible for very small objects (under, say, a hundred elements), but will grow to significant levels as larger arrays are processed.

1. Use EAPPLY and ELAMBDA to avoid generating intermediate results. Consider the example, discussed in section 8.4.1, of computing x^2 for two arrays O and E by using the expression ((O E)^2)/E. This generates the array O E and then (O E)^2, before producing the result. If instead, the formula is expressed as an ELAMBDA function which expects scalars, and this is then applied to the arrays O and E, no unnecessary results will be computed.
2. GROUP will run much faster if the attributes argument has value labels than if it does not, as an unlabelled classifier requires a preliminary pass to determine the actual values that occur.

Complete EAPPLY specifications

EAPPLY provides a mechanism for "splitting up" arrays that are too big (i.e., of too great dimensionality) for a function to handle; calling the function once for each split piece; and "pasting together" the returned values from the function calls into a result array. Its exact operation is quite complex, and some notation has to be introduced to make a precise specification possible.

Say an extended function has *N* arguments, numbered from left to right as 1, ..., *i*, ..., *N*. For each argument *i*, call the number of dimensions the function expects in that position a_i, and call the number of dimensions of the argument actually supplied, b_i. If a_i is NIL (i.e., the function will accept any argument), EAPPLY will ignore this argument and pass it to the extended function, without change, on every call. If a_i is ARRAY (i.e., the decomposition is to be controlled entirely by the keeps), then set a_i equal to b_i. If the number of kept dimensions on the *i*th argument exceeds b_i-a_i (i.e., the user is forcing the argument to be broken down into smaller pieces than the function expects), then set a_i to b_i minus the number of dimensions kept. The effective dimension discrepancy, d_i, is b_i-a_i. If this is non-positive, for any argument, that argument will be passed to the extended function without change on every call. The remainder of this discussion applies only to arguments that have positive discrepancy.

First, the shape vector of each argument with positive d_i is reordered so that any kept dimensions come first, in the order in which they were kept, followed by any unkept dimensions, in their natural order. Thus, for a 4-dimensional object A with shape [6 2 4 3], (KEEP A 4 1 3) results in the reordered shape vector [3 6 4 2].

The leftmost argument of greatest d_i is called the *controlling* argument. Let it be the *cth* argument. All the other arguments must match it in their "excess shapes". That is, for each argument *i*, the first d_i of its reordered subscripts must have the same extents as the first d_i of the controlling argument.

The operation now proceeds as follows. The controlling argument is enumerated in terms of sub-arrays of size a_c. The enumeration is of the reordered set of subscripts in row-major order. For example, if the controlling argument has shape [3 4 2] and the third and first dimensions are kept, its reordered shape vector is [2 3 4]. If the extended function expects vectors (i.e., a_c is 1), then the function will be called 3*2 times, each time with a 4-vector arg@[-,-,ALL] in the controlling argument position.

The subscripts of the other arguments are enumerated in parrallel with the subscripts of the controlling argument as follows. For the *i*th argument, the d_i leading subscripts in its reordered list are matched with the first d_i in the controlling argument's reordered subscript list (hence the conformability requirement stated above). For example, if the controlling argument is as above, and another argument, *i*, for which a vector is expected, has reordered shape [2 6], then d_i is 1. Note that the first extent, 2, matches that for the controlling argument. Each time the function is called, a 6-vector will be supplied by EAPPLY for the *i*th argument position; it will be matched with the controlling argument as follows:

Control arg	I*th* arg
[1,ALL,1]	[1,ALL]
[2,ALL,1]	[1,ALL]
[3,ALL,1]	[1,ALL]
[1,ALL,2]	[2,ALL]
[2,ALL,2]	[2,ALL]
[3,ALL,2]	[2,ALL]

The results of these applications of the extended function to the split-down arguments are put back together into an array that is an image of the decomposition of the controlling argument into pieces of dimensionality a_c. That is, the first d_c of its reordered subscripts are kept, but are sorted back into their original order to define the first d_c dimensions of the shape of the result. The last dimensions of the result shape are taken from the shape of the function's returned value. So if the function in our example returned a [8 4]-matrix, the result of EAPPLY would be an array of shape [3 2 8 4]: the d_c leading dimensions of the controlling argument in their original order [3 2], followed by [8 4].

Appendix F

Annotated Protocol

This protocol gives an annotated transcript of a data analysis session using IDL. The transcript and its annotations are intended to illustrate some basic techniques of data analysis and the use of IDL operations to accomplish them.

The annotations are set off from the transcript in a small font. The system may be assumed to have typed everything else, except lines beginning with a _ and the first line. These are the user's commands to the system, given in response to the _, which is the Interlisp prompt character. The use of the male pronoun in referring to the user in the annotations is neither conventional nor discriminatory. This particular user is a man.

The user begins at the TENEX executive (which prompts with @). He intends to analyze ratings of wines he collected at a recent wine tasting. Having placed these data in the file TASTING.DATA with an ordinary text editor, he uses the TENEX SEE command to verify that the file is in a suitable format:

```
@SEE TASTING.DATA

(TITLES "The Definitive Wine Tasting" Person Wine)
(LABELS Canyon Heights L'Effete Pallide)
(Ron -2 4 0 4)
(Jeff 2 -1 -4 3)
(Susan 5 4 5 5)
(Henri -10 -9 9 10)
(Kathy 5 -2 3 6)
(Joanne 5 4 -4 3)
(Bob -6 5 6 -3)
(Beau 0 4 2 4)
(Fred -1 1 2 5)
(Janet 4 -2 4 -5)
```

The file is in the format expected by the IDL function IDLMATRIX: The TITLES list contains the title string for the data matrix and the labels (Person and Wine) for its two dimensions, the LABELS list defines the levels on the Wine dimension, and the remaining lines give the level labels for the Person dimension and the data values. With the data in order, he starts IDL to begin the session. IDL prints out a message identifying the version of the system, greets him, and leaves him at the Interlisp executive (the _ prompt). He immediately types an expression that sets the variable TD to the array described in the file.

```
@IDL

INTERACTIVE DATA-ANALYSIS LANGUAGE 18-DEC-78 ...

Good afternoon.

_(TD _ (IDLMATRIX (READFILE 'TASTING.DATA]
[Array 1: Person=10 Wine=4]
```

The Lisp value of this instruction is the array itself, which appears as the square-bracketed expression. An IDL array is assigned a serial number as it is created, and this number serves as a unique identifier for it. The printed representation (or "print-name") of an array includes its serial number and an indication of its shape. For this array, it indicates that TD has two dimensions labelled Person and Wine, and that Person has 10 levels and Wine has 4. The values of the array are not displayed, so the user requests that the whole array be SHOWn to him.

```
_SHOW

    The Definitive Wine Tasting

        Wine
Person      Canyon   Heights  L'Effete   Pallide
     Ron        -2         4        0         4
    Jeff         2        -1       -4         3
   Susan         5         4        5         5
   Henri       -10        -9        9        10
   Kathy         5        -2        3         6
  Joanne         5         4       -4         3
     Bob        -6         5        6        -3
    Beau         0         4        2         4
    Fred        -1         1        2         5
   Janet         4        -2        4        -5

[Array 1: Person=10 Wine=4]
```

The array TD is printed using SHOW, a command which prints the value of the previous command. TD is a simple data matrix, which purports to report on the results of a wine tasting at which 10 people (named Ron, Jeff, ... , Janet) tasted four wines (named Canyon, ... , Pallide) and rated them on a scale from -10 to 10. Each row of the matrix contains, for one person, the score that they gave to each of the four wines.

The user's first question is "How well did people like these wines?". This is answered by looking at the mean (average) of all the ratings. The MOMENTS operator computes the first two moments (the first is the mean and the second the variance, plus the zeroth moment which is the count of the number of observations) of all the values in TD and the user SHOWs it.

```
_(MOMENTS TD)
[Array 2: Moment=3]
_SHOW

    Moments of The Definitive Wine
    Tasting
Moment
          N      Mean  Variance
    40.000     1.625    20.189

[Array 2: Moment=3]
```

The mean is 1.625, somewhat positive, but the variance is large, suggesting that there were differences, either by people, by wines, or by both.

In order to explore these, the user decides to look first at Wines. He recomputes the MOMENTS, KEEPing out Wines. The result will have the MOMENTS done separately for each wine (i.e., down the columns of the original matrix), giving a matrix where each wine is represented by three moments, rather than ten people. The user instructs the system to print (PPA for Pretty Print Array) the value immediately.

```
_(PPA (MOMENTS (KEEP TD 'Wine]
    Moments of The Definitive Wine
    Tasting keeping Wine

        Moment
Wine             N      Mean  Variance
  Canyon    10.000      .200    26.178
 Heights    10.000      .800    19.289
```

```
L’Effete    10.000     2.300    17.122
 Pallide    10.000     3.200    18.622

[Array 6: Wine=4 Moment=3]
```

The MOMENTS for the Wines reveal that the French wines were slightly better liked. But is this difference significant, or is it just the slight difference you might expect by chance? The user passes the value of the MOMENTS to the ANalysis Of VAriance to find out. He is able to specify that ANOVA is to work on the output of MOMENTS by using the name IT, which is always bound to the last value computed.

```
_(ANOVA IT]
[Array 7: Source=3 2=5]
_SHOW

    Anova of Moments of The Definitive Wine Tasting
    keeping Wine

        Column
Source       SumSq         df         MS          F          p
Gnd-mean   105.625      1.000    105.625      5.202       .029
    Wine    56.475      3.000     18.825       .927       .438
   Error   730.900     36.000     20.303        NIL        NIL

[Array 7: Source=3 2=5]
```

ANOVA is a technique that contrasts the size of differences between groups (here between the various Wines) with the amount of variation in the scores. Such a contrast provides much useful information to the experienced data analyst. Our user notes that the probability that the Grand Mean (the average of the averages) is zero is .029, very unlikely. He was right in his inference that the ratings were significantly positive. However, the differences between the wines are not significantly different from zero (and, therefore, they are not significantly different from each other) as the probability of them being just random perturbations is .438, which is quite high. [Most data analysts consider probability levels greater than 0.05 (1 in 20) too high to permit inference of an effect other than random variation.]

Reflecting on these results, the user realizes that this particular analysis is inappropriate for the data at hand. There are multiple observations (or "repeated measures") for each Person, and this means that some of the "Error" variation might be due to systematic differences between people (i.e., some people might simply like wine more than others). A more elaborate form of the analysis of variance is required so that the inter-wine variation may be tested independently of the inter-person variation. In the correct analysis, both Person and Wine must be treated as experimental factors so that their separate effects and their interaction may be examined. This is accomplished by letting the input to ANOVA be a MOMENTS table constructed with both dimensions of TD kept:

```
_(ANOVA (MOMENTS (KEEP TD ALL)) ’Person)
[Array 10: Source=4 Column=5]
_SHOW

    Anova of Moments of The Definitive Wine Tasting
    keeping Person and Wine and with Person considered
    random

        Column
Source       SumSq         df         MS          F          p
Gnd-mean   105.625      1.000    105.625     11.300       .008
  Person    84.125      9.000      9.347        NIL        NIL
    Wine    56.475      3.000     18.825       .786       .512
     P*W   646.775     27.000     23.955        NIL        NIL

[Array 10: Source=4 Column=5]
```

The second argument (Person) to ANOVA indicates that Person is to be considered a "random" factor. That is, the user considers the ten people who participated in the wine tasting as a small random sample drawn from the set of people in general. He wants to make inferences from the ratings of the particular sample of ten to the ratings that would likely be obtained from the larger population. The analysis of variance procedure selects an appropriate test for each effect of such designs, if one exists. In this case, there is no test for the Person variation, so the probability value is NIL.

The results of the corrected analysis are essentially the same as the first: The Grand Mean is slightly more significant than it was before, while the Wine effect is slightly less significant.

Despite the lack of statistical significance, the direction of the wine differences bothers the user. His impression of the evening was that the wines were thought of as being very similar. He did not expect to see a trend, even if slight, towards the French. Following these thoughts, he decides to explore differences between the people. He computes another set of MOMENTS, this time KEEPing out just Person.

```
_(MOMENTS (KEEP TD ’Person))
[Array 11: Person=10 Moment=3]
_SHOW

    Moments of The Definitive Wine
    Tasting keeping Person

        Moment
Person           N      Mean  Variance
     Ron     4.000     1.500     9.000
    Jeff     4.000     0.000    10.000
   Susan     4.000     4.750      .250
   Henri     4.000      .000   120.667
   Kathy     4.000     3.000    12.667
  Joanne     4.000     2.000    16.667
     Bob     4.000      .500    35.000
    Beau     4.000     2.500     3.667
    Fred     4.000     1.750     6.250
   Janet     4.000      .250    20.250

[Array 11: Person=10 Moment=3]
```

The data does show strong differences between the various Persons. Susan was the most positive; Henri and Jeff the most negative although, interestingly enough, noone actually averaged below the midpoint of the scale. More interesting than the means, however, are the individual variances. Susan, in addition to being positive, had almost no variance (that is, she liked all the wines about the same, as a glance back at the original data shows). Henri, on the other hand, had very different reactions (much more so than anyone else) to the different wines.

The extreme differences between the amount of variance in Henri’s ratings and the amount everywhere else suggests that one or more of Henri’s ratings may be unreliable - maybe he was given a dirty glass, or something. A very useful technique for detecting such unreliable scores is to look for *outliers*, or values that are clearly more extreme than all the others. This type of screening can be used to detect values that are the results of other influences than the one the investigator had in mind; subjects falling asleep during reaction time experiments, for example.

In this case, a simple but effective method, is to plot a frequency distribution of the scores. GROUP is used to classify a default vector of all ones according to the values found in TD. The values are RESHAPEd (to a vector) initially, otherwise the GROUP would *cross* classify by the columns (i.e., find all the rows with score x on wine 1, y on wine 2, etc.). Here, the user is interested in putting all instances of the same score together, no matter which column of the array they came from. Then, COUNTS is used to count the number of observations with each value

```
_(COUNTS (GROUP (RESHAPE TD]
[Array 18: Value=21]
_SHOW

    Counts of Group of 1 by Values of The Definitive Wine
    Tasting keeping Value
Value
      -10=       -9=       -8=       -7=       -6=       -5=
         1         1         0         0         1         1

Value
       -4=       -3=       -2=       -1=        0=        1=
         2         1         3         2         2         1

Value
        2=        3=        4=        5=        6=        7=
         3         3         8         7         2         0

Value
        8=        9=       10=
         0         1         1

[Array 18: Value=21]
```

and HIST displays the result as a HISTogram (or bar graph).

```
_(HIST IT]

          Histogram of Counts of Group of 1 by Values of The
          Definitive Wine Tasting keeping Value

   Value
    -10=  |*
     -9=  |*
     -6=  |*
     -5=  |*
     -4=  |**
     -3=  |*
     -2=  |***
     -1=  |**
      0=  |**
      1=  |*
      2=  |***
      3=  |***
      4=  |********
      5=  |*******
      6=  |**
      9=  |*
     10=  |*

          Each * indicates 1.  The | is at 0.
[Array 18: Value=21]
```

The HIST shows the form of the distribution clearly. The most common reaction was quite positive (a score of 4). However, there were some negative observations that, in the absence of any equally extreme positive observations, pulled the average down. [The asymmetry around the mean could also have been detected from the higher moments, in particular, from the third moment or *skew*.] This asymmetry immediately explains the discrepancy between the user's recollections and the average: his recollections reflected the *common* reaction, rather than the average.

The HIST also shows four outliers, scores that lie outside the range of all the others. These four scores are, furthermore, at the limits of the scale, so they are doubly suspicious. Going back to the data, the user finds that all four belong to the same individual (Henri), which explains his high variance, and persuades the user to delete him from the data matrix. Whatever was determining Henri's scores, it clearly had little to do with what was determining everyone else's (and the direction of the differences even raises a dark suspicion in the user's mind).

To delete Henri, the user simply creates a new data matrix formed by *selecting* all the people (rows) except Henri, and all the variables. This data matrix, NTD, will form the basis for the subsequent analyses.

```
_(NTD _ TD@'((1 2 3 5 6 7 8 9 10) ALL]
[Array 19: Person=9 Wine=4]
_SHOW

    Selected Persons from The Definitive Wine Tasting

        Wine
Person      Canyon   Heights  L'Effete   Pallide
     Ron        -2         4         0         4
    Jeff         2        -1        -4         3
   Susan         5         4         5         5
   Kathy         5        -2         3         6
  Joanne         5         4        -4         3
     Bob        -6         5         6        -3
    Beau         0         4         2         4
    Fred        -1         1         2         5
   Janet         4        -2         4        -5

[Array 19: Person=9 Wine=4]
```

. . . no Henri, redoing the MOMENTS by Wine . . .

```
_(MOMENTS (KEEP NTD 2]
[Array 35: Wine=4 Moment=3]
_SHOW

    Moments of Selection of The
    Definitive Wine Tasting

        Moment
Wine              N      Mean  Variance
  Canyon      9.000     1.333    15.000
 Heights      9.000     1.889     8.361
L'Effete      9.000     1.556    13.028
 Pallide      9.000     2.444    14.528

[Array 35: Wine=4 Moment=3]
```

. . . and this time, the results show no differences for country of origin. Another way to reduce the impact of outliers is to recode the data. Here, the problem is the non-comparability of the scores from one person to another. A recode which solves this problem is to use the ranks of the values within people, rather than the values themselves. This provides a scaling of the scores to common units. The RANK operator, applied within the (kept) Person dimension (the rows) produces the ranks.

```
_(RANK (KEEP TD 'Person]
[Array 49: Person=10 Wine=4]
```

```
_SHOW

    Rank of The Definitive Wine Tasting keeping
    Person

        Wine
Person      Canyon   Heights  L’Effete   Pallide
     Ron     1.000     3.500     2.000     3.500
    Jeff     3.000     2.000     1.000     4.000
   Susan     3.000     1.000     3.000     3.000
   Henri     1.000     2.000     3.000     4.000
   Kathy     3.000     1.000     2.000     4.000
  Joanne     4.000     3.000     1.000     2.000
     Bob     1.000     3.000     4.000     2.000
    Beau     1.000     3.500     2.000     3.500
    Fred     1.000     2.000     3.000     4.000
   Janet     3.500     2.000     3.500     1.000

[Array 49: Person=10 Wine=4]
```

In this case, they are floating numbers, as fractional ranks had to be introduced to represent ties.

```
_(MOMENTS (KEEP IT ’Wine]
[Array 53: Wine=4 Moment=3]
_SHOW

    Moments of Rank of The Definitive
    Wine Tasting keeping Person
    keeping Wine

        Moment
Wine             N      Mean  Variance
  Canyon    10.000     2.150     1.558
 Heights    10.000     2.300      .844
L’Effete    10.000     2.450     1.025
 Pallide    10.000     3.100     1.156

[Array 53: Wine=4 Moment=3]
```

The MOMENTS of the RANKs shows similar results to those obtained by eliminating the outliers.

Another way of approaching this type of data is to look at the relationship between the ratings produced by various people. Do they agree with each other, and who agrees with whom? The global level of agreement is given by the variance, and could be tested by an ANOVA of the MOMENTS KEEPing Person. The second question, however, requires a different form of analysis, an analysis of the covariations among the ratings. The covariation operator, COVAR, keeps the columns of its (matrix) argument as the objects between which the covariations are recorded, and computes those covariations across the rows. Usually, this preserves the observations and suppresses the subjects of a subjects by observations matrix. Here, the user is interested in the covariation between Persons, so he TRANSPOSEs the data array as he applies COVAR. He also asks for the covariations to be scaled (NORMalized) by their variances, so that each covariation appears in the same units. These scaled covariations, which range from 1.0 to -1.0, are the *correlation coefficients*, very useful measure-independent indices of covariation.

```
_(PPA (NORM (COVAR (TRANSPOSE TD]

    Norm of Covariations of Transpose of The Definitive Wine Tasting
```

```
        Person
Person         Ron       Jeff      Susan      Henri      Kathy    Joanne
     Ron     1.000
    Jeff      .141      1.000
   Susan     -.556       .211      1.000
   Henri      .243      -.163       .546      1.000
   Kathy     -.375       .533       .937       .469      1.000
  Joanne      .163       .800      -.327      -.684      -.023     1.000
     Bob      .319      -.891      -.507       .200      -.728     -.649
    Beau      .986      0.000      -.522       .349      -.391     -.000
    Fred      .689       .169       .200       .838       .300     -.261
   Janet     -.926      -.492       .333      -.243       .062     -.381


        Person
Person         Bob       Beau       Fred      Janet
     Bob     1.000
    Beau      .441      1.000
    Fred      .101       .731      1.000
   Janet      .056      -.870      -.733      1.000

[Array 56: Person=10 Person=10]
```

The correlation matrix, printed in two planes so as to fit on the page, shows the correlations between people across wines. As the covariation between *a* and *b* is the same as that between *b* and *a*, the matrix is symmetric.

The user's interest in this data is very practical. If he (Ron) has to miss the next wine tasting, whose opinion should he seek to best predict what his reactions would have been had he been there? Scanning the correlations in the column (or the row) under his name, he finds that Beau is an excellent choice, as he and Beau agree very closely. Joanne or Jeff on the other hand, are very poor sources of information. Strangely enough, if Beau also misses the next tasting, the next best source of information is not Fred, with a correlation of .689, but Janet! The fact that Janet and Ron disagree almost completely makes her a very useful source of information - all Ron has to do is to take her preferences and reverse them and he will have an excellent predictor of the choice he would have made!

Having finished with his immediate investigation of the tasting, the user decides to load in some information he has about the people that were there, to see if any of their characteristics (other than nationality!) predict their preferences. This file, unlike the last one, is defined to set the variable PATTRIB to the data array it contains. Thus, he merely has to LOAD it and PATTRIB will be set automatically.

```
_LOAD(TCLASS.DATA]
FILE CREATED 12-FEB-78 20:42:52
TCLASSCOMS
<IDL>TCLASS.DATA;2
_(PPA PATTRIB)

    People attributes

        Variable
Person         Sex Experienc       Age
     Ron      Male    Expert        31
    Jeff      Male      Some        38
   Susan    Female      None        31
   Henri      Male      Some        23
   Kathy    Female      Some        26
  Joanne    Female      None        32
```

```
     Bob      Male    Expert        42
    Beau      Male      Some        29
    Fred      Male      None        27
   Janet    Female    Expert        33

[Array 57: Person=10 Variable=3]
```

The matrix of attributes has a row for each person with values for each of three attributes: the person's Sex, previous wine-drinking Experience, and Age. Age is like the wine-ratings the user has been dealing with in that the magnitude of the numerical scores is meaningful. This is not true for Sex, where the variable indicates simply that a person falls into either the Male or Female category. IDL permits labels to be associated with the numerical values of such categorization variables, and those labels are printed instead of the underlying numerical scores. The mapping of values into labels is defined in a *codebook* associated with a variable; codebooks can be examined by means of the CODE selector:

```
_(PATTRIB@(CODE 'Sex))
((1 Male) (2 Female))
```

A codebook is a list of pairs that maps values into labels and vice versa. A different codebook can be associated with each variable (or more generally, with each level on a single dimension, called the value-labelled dimension). Besides allowing arrays to be displayed in a more intuitive format, value-labels and codebooks are used in certain kinds of operations, the most notable of which is the GROUP operation. GROUP was used above to group the wine ratings so that COUNTS could produce a frequency distribution. More generally, GROUP will impose a multi-dimensional structure defined by a categorization matrix on a second array. For example, the following statement groups the wine-tasting data into categories defined by the Sex and Experience of the people:

```
_(PCTD _ (GROUP PATTRIB@'((1 2)) TD]
[Array 62: Sex=2 Experience=3 Person=3 Wine=4; kept Sex Experience]
```

The attributes for GROUP are the first two columns of the PATTRIB matrix (the selector for the first dimension is defaulted to ALL). GROUP considers each row of PATTRIB to be the address in a Sex by Experience space into which the corresponding row of TD is to be placed (both arguments must have the same number of rows). Since Ron has Sex = Male and Experience = Expert, his data is placed in the Male-Expert cell of the classification space. The array resulting from the GROUP has 4 dimensions, 2 resulting from the attributes, and the original 2 from TD. The variable (column) labels of the attributes become the labels for the first 2 dimensions. The number of levels and the level labels for those dimensions are determined from the variables' codebooks. The print-name of the resulting array indicates not only the shape, but also the fact that the dimensions resulting from the attributes are implicitly kept. This means that subsequent operations will apply within the levels of those dimensions.

The grouped array has 72 (= 2 x 3 x 3 x 4) cells, whereas there were only 40 observations in the data. The reason is that the GROUP allocates space for the largest number of rows found in any of the conditions; the other conditions, into which fewer rows fall, will be filled with missing data values (NIL). The user issues a SHOW command to display the array, this time giving an argument that specifies the event on Interlisp's history list whose value he wishes to see:

```
_SHOW GROUP

    Group of The Definitive Wine Tasting by
    Selected Variables from People attributes

    Kept:  Sex Experience

Sex = Male
Experience = None
        Wine
Person      Canyon   Heights  L'Effete   Pallide
       1        -1         1         2         5
       2       NIL       NIL       NIL       NIL
       3       NIL       NIL       NIL       NIL


Sex = Male
Experience = Some
        Wine
Person      Canyon   Heights  L'Effete   Pallide
       1         2        -1        -4         3
       2       -10        -9         9        10
       3         0         4         2         4


Sex = Male
Experience = Expert
        Wine
Person      Canyon   Heights  L'Effete   Pallide
       1        -2         4         0         4
       2        -6         5         6        -3
       3       NIL       NIL       NIL       NIL


Sex = Female
Experience = None
        Wine
Person      Canyon   Heights  L'Effete   Pallide
       1         5         4         5         5
       2         5         4        -4         3
       3       NIL       NIL       NIL       NIL


Sex = Female
Experience = Some
        Wine
Person      Canyon   Heights  L'Effete   Pallide
       1         5        -2         3         6
       2       NIL       NIL       NIL       NIL
       3       NIL       NIL       NIL       NIL


Sex = Female
Experience = Expert
        Wine
Person      Canyon   Heights  L'Effete   Pallide
       1         4        -2         4        -5
       2       NIL       NIL       NIL       NIL
       3       NIL       NIL       NIL       NIL

[Array 62: Sex=2 Experience=3 Person=3 Wine=4; kept Sex Experience]
```

Up to now, the user has worked with only vectors or matrices, which print as simple 1 or 2-dimensional panels on the page. Higher-dimensional arrays print as a sequence of 2-dimension panels. Each panel is prefaced with a description of the levels on the leading dimensions that the panel represents. The panel itself has the labels for the last two dimensions. For the classified array, there are six panels, corresponding to the 2 x 3 grouping he imposed on the TD matrix.

Having grouped his tasting data, he can now ask how wine ratings varied as a function of the Sex and Experience of the taster. Perhaps Females liked certain wines but not others, or maybe some wines were more attractive to palates with more Experience. The user explores this by taking the MOMENTS of the grouped data, KEEPing the Sex and Experience dimensions. Earlier he used the function KEEP to mark which dimensions of an array he wanted to be preserved. That explicit marking is not necessary here, since the output of GROUP is automatically marked so that the classification dimensions will be kept, on the theory that the user wouldn't want his carefully constructed group structure to be completely ignored by the very next operation. This being the case, he obtains a Sex by Experience MOMENTS table by requesting:

```
_(MOMENTS PCTD]
[Array 65: Sex=2 Experience=3 Moment=3]
_SHOW

   Moments of Group of The Definitive
   Wine Tasting by Selected Variables
   from People attributes keeping Sex
   and Experience

Sex = Male
        Moment
Experience
                 N      Mean  Variance
    None     4.000     1.750     6.250
    Some    12.000      .833    38.152
  Expert     8.000     1.000    19.143


Sex = Female
        Moment
Experience
                 N      Mean  Variance
    None     8.000     3.375     9.411
    Some     4.000     3.000    12.667
  Expert     4.000      .250    20.250

[Array 65: Sex=2 Experience=3 Moment=3]
```

There is no clear pattern in the Means, other than that there is quite a bit of variation across the different classifiers. The user can get a better assessment of the situation by calling ANOVA on this table to produce a 2-way analysis of variance:

```
_(PPA (ANOVA IT]
    Anova of Moments of Group of The Definitive Wine
    Tasting by Selected Variables from People attributes
    keeping Sex and Experience

    Harmonic mean of cell N's:    5.538

        Column
Source       SumSq        df        MS         F         p
Gnd-mean    96.194     1.000    96.194     4.437      .043
     Sex     8.540     1.000     8.540      .394      .534
Experien    21.561     2.000    10.780      .497      .613
     S*E    13.330     2.000     6.665      .307      .737
   Error   737.042    34.000    21.678       NIL       NIL

[Array 66: Source=5 2=5]
```

The relatively high p values confirm the intuition that there are no strong effects in this data. If there had been a significant main-effect for Experience, say, the user might have then computed the MOMENTs for the Experience grouping alone to see where the differences were. To do this, he would need to eliminate one of the two implicitly kept dimensions of PCTD. The function LEAVE, described in Section 2.3, is provided for just this purpose.

GROUP, in conjunction with MOMENTS and ANOVA, is a solid foundation for examining the group differences on a measure or set of measures. The user is also interested in the relationship between a person's age and wine-ratings. He wonders, for example, whether older people give consistently higher ratings than younger people. He could answer this question by placing the people in a small number of age-groups, say, Young (below 25), Middle (between 25 and 34), and Old (above 35), and then applying the techniques he used above. But combining people of different ages into a single group throws away some of the information present in the data, so the user tries a more powerful analysis. He decides to find out how well wine-ratings can be predicted as a linear function of Age, using a linear regression technique. Initially, he cares more about ratings in general than about ratings on particular wines, so he collapses across the Wine dimension to obtain an average rating for each person. There are many ways of doing this. Here, the user takes the MOMENTS across Person, and selects the mean out of the result.

```
_(AVRATING _ (MOMENTS (KEEP TD ’Person))@’(Mean]
[Array 71: Person=10]
_SHOW

    Mean of The Definitive Wine Tasting keeping Person
Person
       Ron      Jeff     Susan     Henri     Kathy    Joanne
     1.500     0.000     4.750      .000     3.000     2.000

Person
       Bob      Beau      Fred     Janet
      .500     2.500     1.750      .250

[Array 71: Person=10]
```

The function COVAR, which is the starting point for regressions in IDL, produces a covariation matrix for a single matrix of data. SInce the user wants to include Age and and average rating in a single regression, he must first combine them into a single matrix. He does this with the function ADJOIN, which in its simplest usage joins a set of vectors together end to end. If higher-dimensional objects are given as arguments, IDL's function extension mechanism is invoked to slice the larger objects down to the vectors that ADJOIN knows how to deal with.

In the present situation, the user wants to think of PATTRIB as a collection of variable-vectors, one for each person. In other words, he wants the Person dimension kept in the result of the joining. Since Person is the first dimension, the default rule for function extension will extend along Variable, just what the user wants, without further specification. AVRATING is a little more complicated. It is a vector already, so the default rule would not break it down any further. Thus, (ADJOIN PATTRIB AVRATING) would simply add the average-rating vector as a series of 10 new variables at the end of each row in PATTRIB. This is clearly not what the user intends. The desired effect is to pair the first element of AVRATING with the first row of PATTRIB, the second element with the second row, and so on. In other words, the user wants to think of AVRATING as a sequence of 10 scalars, not a vector of length 10. He can get the desired affect by KEEPing the first (and only) dimension of AVRATING:

```
_(PVARS _ (ADJOIN  PATTRIB (KEEP AVRATING 1]
[Array 86: Person=10 Variable=4]
_SHOW

    Adjoin of People attributes and Mean of The
    Definitive Wine Tasting keeping Person
    keeping Person

        Variable
Person         Sex Experienc       Age         4
     Ron      Male    Expert    31.000     1.500
    Jeff      Male      Some    38.000     0.000
   Susan    Female      None    31.000     4.750
   Henri      Male      Some    23.000      .000
   Kathy    Female      Some    26.000     3.000
  Joanne    Female      None    32.000     2.000
     Bob      Male    Expert    42.000      .500
    Beau      Male      Some    29.000     2.500
    Fred      Male      None    27.000     1.750
   Janet    Female    Expert    33.000      .250

[Array 86: Person=10 Variable=4]
```

The average ratings appear as a fourth variable in the new matrix, bound to PVARS. The title and labels for PVARS are not quite appropriate, however, and the user decides to change them. He first adds a variable name for the fourth column, and then he adds a more descriptive title, using IDL's LABEL and TITLE selectors.

```
_(PVARS@(LABEL ’Variable 4) _  ’Avrating]
Avrating
_(PVARS@(TITLE) _  "Attributes + Average wine rating"]
"Attributes + Average wine rating"
_(PPA PVARS)

    Attributes + Average wine rating

        Variable
Person         Sex Experienc       Age  Avrating
     Ron      Male    Expert    31.000     1.500
    Jeff      Male      Some    38.000     0.000
   Susan    Female      None    31.000     4.750
   Henri      Male      Some    23.000      .000
   Kathy    Female      Some    26.000     3.000
  Joanne    Female      None    32.000     2.000
     Bob      Male    Expert    42.000      .500
    Beau      Male      Some    29.000     2.500
    Fred      Male      None    27.000     1.750
   Janet    Female    Expert    33.000      .250

[Array 86: Person=10 Variable=4]
```

Having gone to the trouble of constructing this matrix, the user decides to save it on a file so that it can be loaded in for future analyses. He does this with Interlisp's ordinary file-making function, MAKEFILE. First he specifies that the file is to contain the matrix bound to PVARS, by setting the variable NEWVARCOMS to a IDLARRAYS file command. With that done, he makes the file named NEWVAR.

```
_SETQQ(NEWVARCOMS ((IDLARRAYS PVARS]
((IDLARRAYS PVARS))
_MAKEFILE(NEWVAR]
<KAPLAN>NEWVAR.;1
```

He is now ready to examine the covariation structure of PVARS. He applies COVAR to a selection from PVARS, in order to exclude the variable Sex, for which this type of analysis is inappropriate. He includes the categorized variable Experience, because there is some notion of ordering in its groupings.

```
_(C _ (COVAR PVARS@’((Experience Age Avrating]
[Array 88: Variable=4 Variable=4]
_SHOW
```

```
    Covariations of Selected Variables from
    Attributes + Average wine rating

        Variable
Variable Experienc       Age  Avrating  Constant
Experien     6.000
     Age    16.000   283.600
Avrating    -6.250   -22.250    21.031
Constant     2.000    31.200     1.625     -.100

[Array 88: Variable=4 Variable=4]
```

The covariation matrix entries are expressed in raw score terms. It is easier to gauge the size of the relations when they are scaled with respect to the units of measurement and the amount of variance. This is done, as before, by the NORM operator.

```
_(PPA (NORM IT]
    Norm of Covariations of Selected
  Variables from Attributes + Average
    wine rating

        Variable
Variable Experienc       Age  Avrating
Experien     1.000
     Age      .388     1.000
Avrating     -.556     -.288     1.000

[Array 89: Variable=3 Variable=3]
```

This correlation matrix shows some interesting patterns, but before exploring them, the user remembers that he has requested the sequence COVAR of NORM to produce correlation matrices before, and realizes that he is likely to request it again. To make it more convenient, he packages them together into a single operation, with Interlisp's standard function-defining function, DEFINEQ. He associates the name CORR with the new operation, and tests it by replicating the matrix he computed above.

```
_DEFINEQ((CORR (M)(NORM (COVAR M]
(CORR)

_(PPA (CORR PVARS@'((Experience Age Avrating]
    Norm of Covariations of Selected
  Variables from Attributes + Average
    wine rating

        Variable
Variable Experienc       Age  Avrating
Experien     1.000
     Age      .388     1.000
Avrating     -.556     -.288     1.000

[Array 92: Variable=3 Variable=3]
```

The correlation matrix shows a fairly high negative correlation (-.556) between Experience and Avrating, suggesting that these particular wines did not appeal to sophisticated drinkers. On the other hand, Experience is correlated positively with Age, which seems reasonable, and Age is also negatively correlated with Avrating. Perhaps the high Experience/Avrating correlation is an artifact of the relationship that both those variables have to Age. To determine whether this is the case, the user wants to remove (or SWEEP out) the variation associated with Age from the matrix.

```
_(SWEEP C '(Age]
[Array 93: Variable=4 Variable=4]
_SHOW

    Sweep of Crossproducts of Selected Variables
    from Attributes + Average wine rating, Swept
    out: Age Constant

        Variable
Variable Experienc       Age  Avrating  Constant
Experien     5.097
     Age      .056     -.004
Avrating    -4.995     -.078    19.286
Constant      .240      .110     4.073    -3.532

[Array 93: Variable=4 Variable=4]
```

The swept matrix gives the covariation space with the variance on Age removed. Many useful pieces of information can be read off this matrix. For example, NORMing it will give the *partial correlations* between the (two) variables whose variance remains.

```
_(PPA (NORM IT]
    Norm of Sweep of Crossproducts
    of Selected Variables from
    Attributes + Average wine rating,
    Swept out: Age Constant

        Variable
Variable Experienc  Avrating
Experien     1.000
Avrating     -.504     1.000

[Array 94: Variable=2 Variable=2]
```

The correlation between Experience and Avrating is reduced (specifically, by about 16% of the joint variance) but a significant relationship still exists. Therefore, the user decides that he will get a better prediction for Avrating if he also removes Experience. He retrieves the output of the previous SWEEP using the Interlisp VALUEOF function, and SWEEPs out Experience. This has the same effect as if they had both been swept out together.

```
_(S _ (SWEEP (VALUEOF SWEEP) 'Experience]
[Array 95: Variable=4 Variable=4]
_SHOW

    Sweep of Crossproducts of Selected Variables
  from Attributes + Average wine rating, Swept
    out: Age Constant Experience

        Variable
Variable Experienc       Age  Avrating  Constant
Experien     -.196
     Age      .011     -.004
Avrating     -.980     -.023    14.391
Constant      .047      .107     4.308    -3.544

[Array 95: Variable=4 Variable=4]
```

The new matrix describes the variable space with both Age and Experience removed. As the user is interested in the regression (linear prediction) of the average rating using these two variables, he needs to extract the regression coefficients for Avrating on Age and Experience. There are three of these, coefficients for Age, Experience, and the Constant term, and they are found in the Avrating row (or column) of the matrix, where it meets the corresponding column (or row). For example, the regression coefficient for Age is -.023. The coefficients could be read out of the matrix and typed in to form a linear equation, but it is generally preferable to extract the coefficients directly from the matrix, as they are represented there with far greater precision than appears in the display.

The extraction and the computation of the predicted values could be done by selecting the coefficients (e.g. (S@'(AVRATING Age) for Age) and using them with selections on PVARS in a linear equation involving the operators * and +. This works because the arithmetic operators in IDL have been extended so that they will apply to corresponding elements of the various selections to produce a vector result. However, the user realizes that the desired result is equivalent to a matrix product that can be computed in a single step, by means of the MPROD operator. The arguments to MPROD are the matrix of independent variables, formed by joining the Constant 1 to the end of each row in PVARS, and the vector of coefficients selected from the Avrating column of the swept matrix S:

```
_[PV _ (MPROD (ADJOIN PVARS@'((Age Experience)) 1)
              S@'((Age Experience Constant)(Avrating]
[Array 159: Person=10 Variable=1]
_SHOW

    Matrix product of Adjoin of Selected Variables
    from Attributes + Average wine rating and 1
    and Selection of Sweep of Crossproducts
    of Selected Variables from Attributes + Average
    wine rating, Swept out: Age Constant Experience

        Variable
Person    Avrating
     Ron      .650
    Jeff     1.467
   Susan     2.610
   Henri     1.815
   Kathy     1.746
  Joanne     2.586
     Bob      .395
    Beau     1.676
    Fred     2.702
   Janet      .603

[Array 159: Person=10 Variable=1]
```

This regression accounts for ~30% (1-the ratio of the Avrating diagonals before and after the SWEEP) of the variance on Avrating. The rest of the variance may be either random (i.e. accounted for by unmeasured variables) or a reflection of a non-linear relationship between these variables. To find out which, the user decides to examine the residuals (the differences between the predicted and actual values) to see if there is a pattern to them. They can be specified simply by typing in the definition of "residual" and, since eyeballing numbers is of limited utility in detecting trends, the user will immediately plot them against something which will detect the trend he seeks. In this case, as he is worried about possible non-linear relationships between his variables, he plots the residuals against the predicted values. If there are, say, quadratic effects, the plot of differences will show a quadratic form.

```
_(PLOT AVRATING-PV PV)

                Plot of Variable Avrating from Difference of Mean of The
                Definitive Wine Tasting keeping Person and Matrix
                product of Adjoin of Selected Variables from Attributes
                + Average wine rating and 1 and Selection from Sweep of
                Crossproducts of Selected Variables from Attributes +
                Average wine rating, Swept out: Age Constant Experience
                and Variable Avrating from Matrix product of Adjoin of
                Selected Variables from Attributes + Average wine rating
                and 1 and Selection from Sweep of Crossproducts of
                Selected Variables from Attributes + Average wine rating
                Swept out: Age Constant Experience

2.2    |                                             *
       |
       |
       |
       |
       |                            *
       |
       |      *                    *
       |
       |
.2     | *
       |
       |
       |     *
       |                                             *
       |
       |                                               *
       |
       |                      *
       |
-1.8   |                             *
        :-----------------------:----------------------:
        .35                   1.55                   2.7

                X-axis plotted in increments of .05
                Y-axis plotted in increments of .2
NIL
```

In this case, the residuals show no pronounced trends with respect to the predicted values (although the discrete Experience classification is clearly visible in the clustering), so the user decides that the remaining variance is due to other variables which he has not observed. He therefore terminates his analysis session to go gather data on them.

Glossary of terms

Argument--a value that is passed to a function at the time the function is invoked. In the expression (F A), A is an argument of F.

Apply--when a function is invoked, it is said to be applied to its arguments.

Array--a rectangular aggregate of data items each one of which is either *integer*, *floating*, or a *missing data code*. Possibly labelled. See *shape*; see Chapter 1.

Assignment--the saving or storing of a value into a variable or array, from which it can be retrieved later by writing the variable name or array selection in an expression. Written with the operator _, as in var _ expr or array@selector _ expr.

Associative--of an operator or function, the property that (F (F A B) C) has the same effect as (F A (F B C)). PLUS is associative; DIFFERENCE is not.

Atom--a Lisp data object represented by a string of characters not including such special characters as (,), space, and ". The numbers form one subclass of atoms, while the symbols used as identifiers of Lisp variables and functions and as IDL labels form another subclass. See *literal atom*.

Binary--of an operator or function, having exactly two operands or arguments.

Bind--to cause a value to be associated with a variable; see *assignment*.

Break--a stoppage of the evaluation of an expression or a function due to an error. A message about the error is typed and the user is prompted with : rather than _.

CLISP--for *C*onversational *Lisp*, a facility in Interlisp that permits certain common operations to be specified in an abbreviated form by means of special characters. For example, A+B may be used for (PLUS A B) and 'A may be used for (QUOTE A).

Code--a numeric value associated with some value label in an array. For example, codes for a column with selector label SEX might be 1 for value-label FEMALE and 2 for value-label MALE.

Codebook--the set of codes and associated labels used to code a variable in numeric form. A codebook is represented as a *list* of pairs, so that a typical codebook for the variable SEX might be ((1 MALE) (2 FEMALE)).

Commutative--of an operator or function, the property that (F A B) has the same effect as (F B A). PLUS is commutative; DIFFERENCE is not.

Compression--describes a function designed to compress a data array (e.g., a subjects by variables matrix) to some smaller array for subsequent analysis.

Concatenate--put together "end to end". Given two strings "THIS" and "THAT", for instance, their concatenation is the longer string "THISTHAT". Can also be applied to vectors with the ADJOIN function.

Controlling argument--in a call on an extended function, the argument which has the greatest discrepancy in number of dimensions between the actual argument and the function's "expects". If two or more arguments have equal discrepancies, then the leftmost of them.

Data type--the kind of object that a value represents. Some examples are FIXP (integer), FLOATP (floating), STRING; see these terms for definitions.

Default--of an argument to a function, the value that will be assumed if none is explicitly given when the function is used.

Dimensions--of an array, the number of subscripts that are required to select a single element from it; also, the length of its shape vector. See *shape*, *subscript*.

Dimension label--a label for a dimension of an IDL array.

Dribble file--a transcript of an Interlisp session, showing what the user types and how the system responds. The Lisp function DRIBBLE turns the transcript mechanism on and off. IDL automatically creates such a transcript on a file named IDL.TYPESCRIPT.

Element--a single item in an aggregate; e.g., a scalar can be an element of an array.

Evaluate--to compute the value of an expression. Lists are evaluated by function application and literal atoms evaluate to their current bindings.

Expects--the list argument to EAPPLY, EAPPLY* and EXTEND that indicates the dimensionality of the value that the extended function will accept in each particular argument position.

Expression--a description of a computation. See *S-expression*.

Extend--to modify a function ("the extended function") so that the extension mechanism will intercept and decompose arguments with more dimensions than the function is programmed to handle.

Floating--(FLOATP)--a Lisp data type, representing numeric values from about 10^{-37} to about 10^{37} with about seven digits of precision. See *integer* (FIXP).

Function--a set of instructions that indicates how to compute a value from one or more input values, called *arguments*. A function is *applied* to a particular set of arguments in the *expression* (FN A_1 A_2 ...) where FN denotes the function, and A_1, A_2 ... are the arguments. FN may be an atom with which the instructions to be evaluated have previously been associated, or it may be a LAMBDA or ELAMBDA form enclosing them.

Generic--an argument to a function that is expected to be an array of arbitrary dimensionality. Also, a function with such an argument.

Identifier--the atomic name of a *variable* or *function*; so called because it is used to identify the variable or function in expressions.

Index--*subscript* (which see).

Initialize--to assign a value to a variable for the first time; give it a value that will be used or changed later in some sequence of operations.

Inner product--the standard matrix product of two 2-dimensional arrays.

Integer--(FIXP)--a Lisp data type representing numeric quantities within the approximate range plus or minus one billion, without fractions (i.e., whole numbers only).

Inverse--of a matrix, that matrix which when multiplied by the given matrix gives the identity matrix.

Invert--of a matrix, to compute its inverse.

Labels--of arrays. There are four types, a title, and dimension, selector, and value labels.

List--a Lisp data type, written as a sequence of elements enclosed in parentheses. Also, the Lisp function LIST, which constructs a list object from the elements resulting from the evaluation of its arguments. See *S-expression*.

Literal atom--an atom which is not a number.

Matrix--an array with exactly two dimensions. An [n,m]-matrix is a matrix with shape [n m].

Missing data code--a special value that is stored as an element of an array and signals that the data value that should occupy that position is missing, not available, not meaningful, etc. In a sociological survey, for example, the column for spouse's occupation would contain a missing data code for any unmarried respondents. In IDL, NIL is the missing data code.

No-spread--a property of Lisp functions that have an indefinite number of arguments. ADJOIN is a no-spread function.

NIL--a special Lisp object. In IDL, used to represent missing data in an array.

Operator--one or more characters (listed in the Interlisp Reference Manual) giving a convenient way of invoking a particular function, called the function associated with the operator. Thus, + is the operator for the PLUS function in Lisp.

Permutation--a reordering of some set without repetition or deletion. Thus, if the set is [1 2 3 4], then [4 2 3 1] and [1 2 4 3] are permutations but [1 2 3] is not.

Quote--to suppress evaluation of an expression. Usually, the expression (MOMENTS A) appearing as an argument to a function would cause that function to receive an array of moments. If the expression is quoted (i.e., (QUOTE (MOMENTS A))), the function will receive the expression itself. This might be interpretable, for example, as a selector for an array with levels labelled MOMENTS and A. The CLISP operator ' is associated with the Lisp function QUOTE.

Rebind--to assign a new value to a variable; *bind*, *assign*; stresses the fact that the former value of the variable is lost.

Return--a function that computes a value is said to return that value; supply a value for further computation.

Row-major--the order in which most IDL functions process the elements of an array; elements are enumerated with the last subscript varying fastest, so that the [1,1,...,1,1] element is considered before the [1,1,...,1,2] and [1,1,...,2,1] elements, etc.

S-expression or symbolic expression--a description of a computation. A *list* expression is evaluated by *applying* its first element as a function to the *arguments* described by the remaining elements and a *literal atom* evaluates to its current binding.

Scalar--a number (FIXP or FLOATP) or NIL; the set of objects that can be stored as elements of an IDL array.

Selector--a set of subscripts, one for each dimension of the selected array, that serve to pick a scalar or sub-array out of it. See *shape*, *dimension*, *scalar*; also Chapter 1.

Selector label--a label associated with one subscript value of one dimension of an IDL array. It can be used in a selector to specify the associated row or column.

Shape--the vector showing the "size" (subscript boundaries) of an array along each of its dimensions. Thus, the shape of an array with two rows and three columns is [2 3]. See section 2.1.

String--a Lisp data type consisting of a number of characters put together in a row. Typed by the user in quotation marks, as "HELLO", a five-character string of the characters H, E, etc.

Subscript--an integer or selector label, or several of these objects, specifying one or more values of the corresponding dimension of an array, used in selection. For example, the first element in the second row of a two-dimensional array may be selected with the subscripts [2 1] (second row, first column). See *shape*, *dimension*, also Section 2.1.

Sweep--to numerically remove variance components from a covariation matrix. See the description of the SWEEP function.

Transpose--to rearrange the dimensions of an array. The simplest case is the "ordinary" transpose of a matrix, where the (i,j) element of the original array becomes the (j,i) element of the transposed array. Generalized beyond this for IDL use. See the description of the TRANSPOSE function.

Unary--of an operator, having exactly one operand. See *binary*.

User function--a function (set of instructions) written by the user himself in the Lisp and IDL language, as opposed to a system function. Such a function is added to the IDL environment by the Lisp functions DEFINE, DEFINEQ, PUTD, perhaps when a user file is LOADed.

Value--the result of evaluating an expression; the "answer", the quantity produced by some computation. This corresponds with common usage in simple cases, as "the value of 4-3 is 1," but is extended to cover the value of a function call (the result computed by the function) and the value of a variable (the value most recently assigned to the variable)

Value label--one that labels one possible value of a cell of an array; the corresponding value is called its code. For instance, the values used to represent observations of a variable called SEX could have the two value labels MALE and FEMALE. See *code*.

Variable--a name with which has been associated some value. Also, an observation which has been made on many subjects - often represented as a column of a data matrix.

Vector--an array with exactly one dimension. An n-vector is a vector with n elements.

Weight--to allow different subjects to make unequal contributions in the computation of some statistic from a sample of subjects. Usually specified as a column in the data array that contains the desired weight value for each subject. Weights of zero and one are useful for omitting some subjects entirely from the analysis.

References

Anscombe, F. J. and Tukey, J. W.
Examination and analysis of residuals. *Technometrics*, 5, 1963, 141-160.

Blalock, H. M.
Social statistics. New York: McGraw-Hill, 1972.

Dempster, A. P.
Elements of continuous multivariate analysis. Reading: Addison Wesley, 1969.

Goodman, L.
A general model for the analysis of surveys. *American Journal of Sociology*, 77, 1972, 1035-1086.

Green, B. F. and Tukey, J. W.
Complex analyses of variance. *Psychometrika*, 25, 1960, 127-152.

Kerlinger, F. N. and Pedhazur, E. J.
Multiple regression analysis in behavioral research. New York: Holt Rinehart, 1973.

Kruskal, W. H.
Transformations on data. *International encyclopedia of the social sciences*, 15, 1968, 182-193.

Siegel, S.
Non-parametric statistics. New York: McGraw-Hill, 1956.

Snedecor G. W. and Cochran, W. G.
Statistical methods. Ames: Iowa State University Press, 1967.

Teitelman, W.
Interlisp Reference Manual. Palo Alto: Xerox Palo Alto Research Center, 1978.

Winer, B. J.
Statistical principles in experimental design. New York: McGraw-Hill, 1971.