Technical Report

Santa Teresa
Laboratory
San Jose, CA

IBM

TR

THE APL2 NAME ASSOCIATION FACILITY: Understanding the APL-FORTRAN Connection

May 1986 TR 03.286

by Harlan Crowder

May 1986
TR 03.286

The APL2 Name Association Facility

Understanding the APL-FORTRAN Connection

by

Harlan Crowder

General Products Division
Santa Teresa Laboratory
San Jose, California

The illustrations in this report were created using the Graphical Data Display Manager, Interactive Chart Utility running with APL2. Text and graphics were integrated using the Document Composition Facility. The report was set in Helvetica type and produced on the IBM 4250 printer.

The APL2 Name Association Facility

Understanding the APL-FORTRAN Connection

The APL2 Name Association facility allows APL2 application programs to call external functions written in other programming languages, including S/370 Assembler Language, FORTRAN, and, in the VM/CMS environment, REXX. External functions allow APL2 applications to benefit from increased execution performance using components written in compiled code, use of existing compiled code subroutine libraries, and use of special hardware such as the IBM 3090 Vector Facility.

This report describes the APL2 Name Association facility, and emphasizes the combined use of APL2 and FORTRAN in application programs. We explain how the APL-FORTRAN connection is established in the APL environment, and describe some programming techniques for fully exploiting the external function concept. We describe in detail some examples that illustrate the hybrid APL-FORTRAN programming technique.

Reprints of this report may be obtained by writing the author at the following address:

IBM Corporation
Department M30
555 Bailey Avenue
San Jose, CA 95150
USA

The APL2 Name Association facility allows APL2 application programs to call external functions written in other programming languages. In previous implementations of APL, application programs could use only subroutines coded in the APL language; this restriction limited the flexibility and range of many APL application systems. With the APL2 Name Association facility, APL applications can use subroutines written in a variety of programming languages, including S/370 Assembler Language, FORTRAN, and, when operating APL2 in the VM/CMS environment, the REXX programming language.

APL application designers and programmers can use the Name Association mechanism with advantage in several ways:

- Execution performance of computationally intensive APL subroutines may be improved by recoding them in FORTRAN or Assembler Language.
- Existing FORTRAN subroutine library routines may be used, without modification, in APL applications.
- APL applications can use the facilities of REXX in the VM/CMS environment for high-level programming language capabilities, string processing, and access to host system information.
- Using VS FORTRAN Version 2, APL applications can take advantage of the high-performance processing power of the IBM 3090 Vector Facility.

The APL-FORTRAN Advantage

In this report, we concentrate on the use of APL in conjunction with FORTRAN.[1] APL and FORTRAN complement each other in several ways, and the Name Association facility allows application designers and programmers to exploit the strengths of each language.

APL is both a concise language and a notation for describing computer algorithms and procedures at a high level. With APL, you describe briefly, and without extraneous detail, the operations that a program performs. With APL, you describe *what* to do, not *how* to do it. From the brief program description, the APL system decides the actual underlying computational algorithms and procedures to execute.

The ability to briefly describe procedures, while subordinating extraneous detail, is the main strength of APL. Thus, APL becomes a very productive tool for implementing application programs, allowing the building of solutions in a small fraction of the time and effort required by other programming tools.

FORTRAN is a high-level programming language with a long history as an efficient and effective computational tool. For computationally intensive problems where efficiency is essential, it is the language of choice. The wide-spread use of FORTRAN in scientific, engineering, academic, and analytical business applications attests to its continued value as an essential programming element. The recent support of the IBM 3090 Vector Facility by the VS FORTRAN Version 2 Program Product further establishes the supremacy of FORTRAN as the Programming Power Tool.

The main advantage of FORTRAN is algorithmic flexibility; with FORTRAN, you specify both what you want to do *and* how you want to do it. This ability to code algorithms and procedures 'close to the machine', but yet still program in a high-level language, gives FORTRAN its great power.

The APL2 Name Association facility offers the high productivity of APL combined with the algorithmic flexibility and high processing power of FORTRAN. The purpose of this report is to explain various facets of the APL-FORTRAN connection so application designers and programmers can use these concepts to maximum advantage.

[1] Execution of FORTRAN routines by APL2 requires IBM VS FORTRAN Release 4 or later. For more information, see [IBM86A].

Organization of This Report

Part 1 is a description of the APL2 Name Association facility, presented from a programmer's point-of-view. We explain the concept of names and objects in APL and show how the Name Association facility extends and enhances this concept. We give a brief overview of the mechanics involved in using the facility, and outline some programming tips and considerations for effective use of the APL-FORTRAN connection.

Part 2 describes and demonstrates two sample applications that use APL and FORTRAN in combination for problem solving.

Part 3 shows how APL2 can be used to measure the computational efficiency of FORTRAN subroutines.

The Appendix describes and lists the APL2 function *CHARTX*, an APL2 data interface to the Graphical Data Display Manager/Interactive Chart Utility.

Acknowledgement

Special thanks to Betty C. Faith and John R. Ehrman; their critical reading and extensive suggestions improved both the style and content of this report.

Note:

This report contains several performance comparisons of sample APL and FORTRAN programs. These comparisons are intended to show the relative performance when running simple routines and are not intended to be a general representation of APL2 or VS FORTRAN performance.

Part 1: The APL2 Name Association Facility

APL Names and Objects

APL programs and applications operate in an environment called the APL *workspace*. A workspace contains all the APL components associated with and necessary for the proper operation of an application; these components are called APL *objects*. For purposes of our discussion, APL workspaces contain three types of APL objects: *arrays*, which are structured collections of alphanumeric data; *functions*, which are programs that manipulate and perform computations on arrays, usually creating new arrays in the process; and *operators*, which process functions to create new functions for application to arrays.

An APL object in a workspace has a *name*, the referent that the object is given when it is created, and by which it is referenced when used in an application program. For example, if the value of the array *A* is the length-4 numeric list 3 5 7 9, then the value of *A* can be displayed by the following expression in an APL interactive session:[2]

```
      A
3 5 7 9
```

The array *A* can also be referenced by name in a computational expression:

```
      2 × A
6 10 14 18
```

Likewise, APL functions are invoked by using their names in expressions. *Primitive functions* in APL are denoted by preassigned symbols, and using the symbol invokes the function. In the previous example, the symbol '×' denoted the function *multiply*. APL *defined functions* are programs, defined as ordered sequences of array operations; defined functions use other functions, both primitive and defined. For example, the defined function *AVG* computes the numeric mean of a list of numbers; *AVG* has the following definition:

```
    ∇
[0]   Z←AVG X
[1]   Z←(+/X)÷⍴X
    ∇
```

The function is invoked by writing its name, followed by its argument, a list of numbers:

```
      AVG 3 9 7 11 14
8.8
      2 × AVG 3 9 7 11 14
17.6
```

In an analogous manner, APL has both *primitive* and *defined operators*. For example, the primitive operator *reduction*, denoted by the symbol '/', can be used to modify the action of primitive functions such as *add* (+) and *multiply* (×):

```
      +/5 3 2 4 1
15
      ×/5 3 2 4 1
120
```

Defined operators, like defined functions, are constructed from ordered sequences of array operations using primitive and defined functions, and possibly primitive and defined operators. The construction and use of defined operators will be addressed later in this report.

In summary, the names of defined objects in APL applications -- arrays, functions, and operators -- are selected by the application designer, and reference data structures and programs. Of particular interest in this report are names that reference defined functions. In

[2] In this report, the usual convention is used for showing APL expressions and their results: the expression is indented six spaces from the left page margin and the result immediately follows displayed at the left margin

previous APL implementations, function names used in applications could only reference programs written in the APL language. In APL2, function names can reference programs written in other programming languages.

External Functions and Name Association

The *Name Association* facility of APL2 allows APL2 application programs to access and use subroutines written in other programming languages. The rules that govern how such routines are used in APL expressions are the same as for routines coded in APL. Of particular interest in this report is the use by APL programs of subroutines written in FORTRAN, but the principles and concepts are general to a wide range of programming languages.

In simple terms, the Name Association facility provides a mechanism for informing the APL2 system that it is to take special action when it encounters the name of a particular function in an APL expression. In particular, the facility allows the system to associate a function name in the APL workspace with an *external function*, a subroutine coded in FORTRAN. After the association has been established, the external function is invoked when the function name is encountered in an APL expression. The details of how the name association is established are explained below. The remainder of this section demonstrates a simple example that shows the Name Association facility in action.

Computing Standard Deviations

An elementary numerical procedure in descriptive statistics is the computation of the *standard deviation* of a set of data points that usually result from some empirical process. A precise definition of the standard deviation algorithm is the APL function *SDA*:

```
    ∇
[0]   Z←SDA X
[1]   Z←(+/X)÷⍴X
[2]   Z←((+/(X-Z)*2)÷⍴X)*.5
    ∇
```

SDA can be used to perform the standard deviation calculation in APL:

```
      SDA 5 11 3 24 8
7.414
```

The subroutine *SDF* is the FORTRAN version of the standard deviation algorithm:

```
      SUBROUTINE SDF(S,N,X)
COMPUTE STANDARD DEVIATION 'S' OF 'N' NUMBERS 'X'
      REAL*8 X(N),S,A
      INTEGER*4 N
      A=0.
      DO 10 I=1,N
   10 A=A+X(I)
      A=A/N
      S=0.
      DO 20 I=1,N
   20 S=S+(X(I)-A)**2
      S=DSQRT(S/N)
      RETURN
      END
```

The following sequence shows how *SDF* is invoked from APL2:

```
      3 11 ⎕NA 'SDF'          ⍝ statement 1
1
      V←5 11 3 24 8           ⍝ statement 2
      N←⍴V                    ⍝ statement 3
      SDF (0 N V)             ⍝ statement 4
7.414
```

Statement 1 uses the APL2 Name Association system function ⎕NA to make the FORTRAN subroutine *SDF* known to APL2; the rules for using ⎕NA will be described in the next section. Statement 2 is an APL assignment expression; it assigns a numeric list to the array named *V*. Statement 3 assigns the number of elements in *V* to the array *N*; in this example, *N* is assigned the value 5. Statement 4 applies *SDF* to its argument structure to compute the standard deviation of the elements of *V*; the computed value is returned as the result of the expression.

The general array argument to *SDF* in statement 4 is a vector with three items, corresponding to the three arguments expected by the FORTRAN program. The first item is a place holder for the result to be computed by *SDF*; the second item is the length of the list that follows; and the third item is the list itself.

The computation using *SDF* could be simplified by using all constants in the function's calling structure; the following statement illustrates the simplified form and gives the same result:

```
      SDF (0 5 (5 11 3 24 8))
7.414
```

Name Association: Mechanics

The APL2 Name Association facility allows access to FORTRAN subroutines; the facility does not support access to FORTRAN function subprograms. Thus, when we use the term *function*, it is in the APL sense.

To establish the association of a FORTRAN subroutine with the name of an APL function in an application workspace, the following questions must be addressed:

1. What is the APL language construct that accomplishes the name-subroutine association?
2. Since the actual FORTRAN-derived executable code is outside the APL environment, how does the system know where to find it?
3. How are the array-structured arguments to an APL associated external function mapped to the argument list required by a FORTRAN subroutine?
4. How are the explicit results returned by an APL function derived from the FORTRAN subroutine's argument list?
5. What happens if errors, such as program checks and exceptions, occur in external routines?

This section gives a brief overview of these and other aspects of the APL2 Name Association facility. For a more comprehensive treatment of these topics, see [IBM85A].

***Author's note:** Some of the material presented in this section assumes some familiarity with systems programming jargon, such as load modules and linkage conventions. If the reader finds himself or herself in unfamiliar territory, please skip ahead to the next section. This material is presented for readers that are interested in such matters, and is not required to understand the following sections of this report.*

The Name Association System Function

The Name Association system function ⎕*NA* is the APL2 language construct used to associate external subroutines with names in APL application programs. In the previous section, the external function *SDF* was made known to the APL2 system by issuing the statement

```
      3 11 ⎕NA 'SDF'
1
```

The left argument to ⎕*NA*, the numeric list `3 11`, indicates that an external function is being associated using associated processor 11, the compiled code external function interface.[3] The right argument to ⎕*NA*, the character string `'SDF'`, is the name of the function to be associated. Note that invoking ⎕*NA* gives a numeric result. The value `1` indicates a successful association. A value of `0` would indicate an unsuccessful association; causes for unsuccessful association will be addressed below.

After a successful association between an APL function name and an external routine, the associated name can be used in APL language expressions exactly as though the name referenced a defined function coded in APL.

NICKNAME Files

A NICKNAME file is the mechanism used by ⎕*NA* to make the system-related connection between the APL name and the external routine code. The NICKNAME file provides the following information:

[3] APL2 also provides associated processor 10, the REXX interface. In the VM/CMS environment, this allows APL2 programs to access and use routines coded in the REXX programming language [IBM83A]. For more information on the use of REXX in APL2, see [IBM85A]

- The load module library where the APL2 system can find the actual FORTRAN-derived executable subroutine code.
- The name of the member in the library; this name can be different from the name used in the APL workspace to invoke the routine.
- The linkage convention used to transfer and return control when the subroutine is called. In this report, only the FORTRAN convention is discussed, but the Name Association facility allows other conventions; see [IBM85A].
- The name of a FORTRAN execution environment that is to gain control of execution should an error occur in the called subroutine.
- A description of the FORTRAN subroutine argument list structure that allows APL2 to build a proper argument sequence when the routine is invoked.

A NICKNAME file can contain entries for a collection of routines to be associated with APL names. Normally, one file will contain all external routine entries related to a specific APL application. Each entry in a file will consist of a sequence of *tag-operand* statements of the form

```
:tag.value
```

Recalling the FORTRAN subroutine *SDF* introduced in the previous section:

```
      SUBROUTINE SDF(S,N,X)
COMPUTE STANDARD DEVIATION 'S' OF 'N' NUMBERS 'X'
      REAL*8 X(N),S,A
      INTEGER*4 N
      A=0.
      DO 10 I=1,N
   10 A=A+X(I)
      A=A/N
      S=0.
      DO 20 I=1,N
   20 S=S+(X(I)-A)**2
      S=DSQRT(S/N)
      RETURN
      END
```

SDF might have the following NICKNAME entry:

```
:nick.SDF          :load.APLDEMO
                   :memb.SDF
                   :link.FORTRAN
                   :init.FORTMAIN
                   :rarg. (G0 1 3) ( < E8 0) (1 I4 *) (E8 1 *)
```

The tag-operand statements in NICKNAME file entries have the following interpretations:

:nick.name - specifies the name of an external function. When making the association, this name must be specified in the right argument of ⎕*NA*. This tag is used to create the link between the name specified with ⎕*NA* and the descriptive information that follows.

:load.library - the name of the load library into which the external function code has been link-edited. In CMS, the library is the name of the LOADLIB file; in TSO, it is the DD name.

:memb.name - the member name of the external function routine in the specified load library.

:link.FORTRAN - the linkage convention used when calling the external function. In this report, we only consider FORTRAN linkage, but other conventions are available.

:init.name - names a FORTRAN main program execution environment to be associated with this routine; this tag is optional. An execution environment is required if the external function uses FORTRAN services such as input/output. Also, execution environments invoke FORTRAN error handling procedures if programming errors should occur in external function routines.

:rarg.pattern - specifies a description for the FORTRAN argument list. (See below.)

The treatment given here to NICKNAME files is cursory and intended to give only a brief introduction. For a comprehensive treatment, see [IBM85A].

Argument Patterns

Argument patterns provide a mechanism for describing the expected arguments for external functions. When an external name is encountered during the execution of an APL expression, APL compares the actual arguments against the pattern provided in the NICKNAME file entry. If possible, APL converts the actual arguments to match the pattern in order that the external routine receives its argument data in the expected and predictable form. If conversion is not possible, APL issues appropriate error messages.

Argument patterns describe both the type and structure of array arguments. Type definitions include G0 for general arrays, I4 for fullword integers, C1 for byte characters, and E8 for double-precision floating point. The numbers following type specifications indicate the rank and shape of array arguments.

The following are examples of how argument patterns describe the APL array structures that can be passed as argument lists to FORTRAN routines:

A matrix, `5 6⍴⍳30`
```
:rarg. I4 2 5 6
```

A nested array, `(2 3⍴⍳6) 'ABCD'`
```
:rarg. G0 1 2 (I4 2 2 3) (C1 1 4)
```

A vector of 3 character strings, any length,
```
'GREETINGS' 'FROM' 'CALIFORNIA'
:rarg. G0 1 3 (C1 1 *) (C1 1 *) (C1 1 *)
```

Fullword integer matrix, any shape
```
:rarg. I4 2 * *
```

Floating point matrix, any shape, containing 100 numbers
```
:rarg. 100 E8 2 * *
```

A single fullword integer, any shape
```
:rarg. 1 I4 *
```

For a more detailed treatment of argument patterns, see [IBM85A].

When Errors Occur

Errors in using the Name Association facility can occur in three main areas:

1. Name Association failures
2. APL errors during external function execution
3. Internal errors in external routines

We briefly discuss these errors and their cause; for a more comprehensive treatment, see [IBM85A].

Name Association failures can occur for several reasons:

- Incorrect arguments to ⎕*NA* - Malformed APL names in the right argument, or incorrect name class or processor numbers in the left argument to ⎕*NA*
- Errors in the NICKNAME file - Invalid or illegal entries in the NICKNAME file.
- Functions cannot be located - Either the function does not have an entry in the NICKNAME file, or the member does not exist in the indicated load library.
- System-related errors - These include insufficient freespace for proper operation of the Name Association processor, or the unavailability of the processor itself.

APL errors during external function execution occur for several reasons.

- APL errors like *RANK ERROR*, *LENGTH ERROR*, or *DOMAIN ERROR* often indicate a mismatch between the external function arguments and the corresponding argument patterns in the NICKNAME file.
- A *VALENCE ERROR* indicates that the external function is called with an incorrect number of arguments or that the external function is unavailable. External functions are unavailable if an error has previously occurred when trying to locate, load, associate, or use the function.
- A *SYSTEM ERROR* usually indicates that the external function has terminated abnormally.

- A *SYSTEM LIMIT* usually means that an attempt has been made to activate an execution environment that is already active.

Internal errors in external routines occur when a FORTRAN program causes an error; these include program checks and ABENDS. If a FORTRAN execution environment is active when a program check occurs, it will normally be handled by the FORTRAN error recovery procedures. If an ABEND occurs, or if any internal error occurs and no execution environment is active, Processor 11 will handle the error and issue a *SYSTEM ERROR*.

Name Association: Programming Considerations

This section provides some programming tips and insights into using the APL2 Name Association facility in conjunction with FORTRAN subroutines.

Forming FORTRAN Argument Lists

The argument lists for FORTRAN subroutines are usually required to be more comprehensive than argument lists for functionally equivalent APL routines. For example, FORTRAN routines require the passing of extents for array arguments, and passing of pre-allocated storage for results and work areas. Because APL routines can functionally determine the extents (i.e., the *shapes*) of their array arguments, and can dynamically allocate space for array results and work areas, such information is not normally passed from the calling routine to an APL function.

Consider again the functions for computing standard deviations introduced previously: *SDA*, coded in APL, and *SDF*, coded in FORTRAN. *SDA* requires the list of numbers for which the standard deviation is to be computed:

```
      SDA 5 11 3 24 8
7.414
```

SDF requires a 3-item array argument, corresponding to the three parameters of the FORTRAN subroutine: a place-holder for the result, the extent of the list, and the list itself:

```
      SDF 0 5 (5 11 3 24 8)
7.414
```

There are several ways to offer the simple APL2 argument structure to FORTRAN routines. One way is to define a companion APL function for a FORTRAN routine that constructs the FORTRAN routine's argument list from essential information. For example, consider the function *SD_ARG* that, given a list for the standard deviation computation, constructs the argument list appropriate for *SDF*. *SD_ARG* has the following definition and use:

```
    ∇
[0]   Z←SD_ARG V
[1]   Z←0(ρV)V
    ∇

      SD_ARG 5 11 3 24 8
 0  5  5 11 3 24 8
```

SD_ARG can be used to simplify the use of *SDF*:

```
      SDF SD_ARG 5 11 3 24 8
7.414
```

Another way to simplify the argument construction process is to use an APL *cover function* that constructs the appropriate FORTRAN argument structure and then invokes the FORTRAN routine. For example, the function *SD* is such a cover function for *SDF*:

```
    ∇
[0]   Z←SD X
[1]   Z←SDF 0(ρX)X
    ∇

      SD 5 11 3 24 8
7.414
```

As explained below, the cover function technique leads naturally to the use of APL programming control structures with FORTRAN external functions.

Using APL Operators With FORTRAN Routines

A powerful control structure in APL is the primitive operator *each*, denoted by the symbol '¨'. *Each* allows the application of a function to each item of an array. For example, *each* can be used in conjunction with the *shape* function, denoted by the symbol 'ρ'. One use of *shape* is to compute the length of a numeric or character list:

```
      ρ 10 4 17 8 13 9 11
7
      ρ 'CATFAT'
6
```

Used with *each*, the *shape* function can be used to determine the length of each item of a list of lists:

```
      ρ¨ (4 9 8) (22 44) (¯5 6 7 ¯11)
 3  2  4
      ρ¨ 'GREETINGS' 'FROM' 'CALIFORNIA'
 9  4  10
```

Using *each* in conjunction with the function *SD* allows the standard deviation computation of each item in a list of numeric lists; the control structure is APL, but the actual computation is done by FORTRAN:

```
      SD¨ (5 11 3 24 8) (17 3 8 11) (3 4 3 6)
7.414 5.068 1.225
```

Another example is the FORTRAN program named *GCDF* that uses Euclid's algorithm for computing the greatest common divisor of two integer numbers:

```
      SUBROUTINE GCDF(M,N,Z)
      INTEGER*4 M,N,Z,IM,IN
      IM=IABS(M)
      IN=IABS(N)
   10 Z=IM
      IM=MOD(IN,IM)
      IN=Z
      IF(IM.NE.0) GOTO 10
      RETURN
      END
```

GCDF would have the following NICKNAME file entry:

```
:nick.GCDF      :load.APLDEMO
                :memb.GCDF
                :link.FORTRAN
                :rarg. (G0 1 3) (1 I4 *) (1 I4 *) (<I4 0)
```

GCDF can be imbedded in the APL function *GCD* to simplify the calling sequence:

```
    ▼
[0]   Z←M GCD N
[1]   Z←GCDF M N 0
    ▼

      24 GCD 32
8

      27 GCD 45
9

      26 GCD 65
13
```

Using *GCD* in conjunction with the *each* operator allows the greatest common divisor computation of multiple pairs of numbers:

```
      24 27 26 GCD¨ 32 45 65
8 9 13
```

Another powerful APL operator is *outer product*, denoted by the combination of symbols '∘.*F*', where *F* is the name of a function. *Outer product* applies the function *F* between items of its

array arguments, in all possible combinations. For example, ∘.+ is a function, derived from the *add* function +, that adds the items of two arrays pairwise in all possible combinations:

```
      10 20 30 ∘.+ 1 2 3 4 5
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
```

Outer product can be used in conjunction with *GCD* to compute the greatest common divisor of all possible pairs of numbers from two arrays; again, the control structure is APL, but the actual computation is done by FORTRAN:

```
      24 26 ∘.GCD 32 45 64
8 3 8
2 1 2
```

Later in this report we will take another look at using APL operators in conjunction with external functions written in FORTRAN.

Global vs. Local External Functions

The process of name association is normally done outside of an application, much like defining APL functions in the workspace. Once an APL name has been associated with an external function by means of the ⎕*NA* system function, the external function is treated as a locked APL function in the workspace. If the workspace is saved and subsequently reloaded, the external function exists in the loaded workspace; the name association task does not need to be repeated. Such a function is called a *global function*; it is known to all programs in the workspace, and any program can use it to perform computations.

In some cases, it may be desirable to have external functions defined only within some top-level APL application program. Such an instance of a function is called a *local function* because it is only known locally within the top-level program and within any other program that the top-level routine invokes. In general, local functions do not exist in the workspace except when the top-level program is in operation.

Local functions are desirable when the operation they perform is applicable only within some well-defined application environment. In such cases, it may be undesirable to have the function defined outside the environment. For example, the function might perform some proprietary operation that is only to be performed within the application environment; the integrity of the application might be jeopardized if the function were made available to the user in stand-alone mode outside the application. Also, using local functions can decrease the 'name pollution' problem in a workspace. The APL system command)*FNS* is used to list the directory of global functions in a workspace, and workspaces that contain many global subfunctions have long directory listings that tend to obscure the names of top-level routines that are the real interest of the application user.

The following steps are required to make an external function local to a main calling routine:

1. Place the name of the external routine in the local name list in the main function header.
2. Perform the name association task in the main routine using the ⎕*NA* system function.

For example, if a top-level routine named *MAIN* uses an external function *EXTFUN*, then *MAIN* will contain the following code fragments:

```
        ∇
[0]  Z←L MAIN R;EXTFUN; . . .
        .
        .
        .
[..] R←3 11 ⎕NA 'EXTFUN'
[..] ⎕ES (R≠1)/'EXTERNAL FUNCTION ASSOCIATION FAILED'
        .
        .
        .
        ∇
```

EXTFUN will exist in the application workspace only while *MAIN* is in actual operation; when *MAIN* finishes execution, *EXTFUN* will cease to exist in the workspace.

Usage Notes:

1. In this example, the return code from ⎕*NA* is checked to ensure that the name association has been accomplished. As a matter of good programming style, it is a good idea to check this value. Recall that a result of 1 indicates a successful association; a 0 indicates failure to associate.
2. The name association operation performed by ⎕*NA* requires extra computer execution time. Therefore, if a routine that uses an external function is called frequently, the external function should probably be global to the calling routine.

Array Element Ordering

The elements of APL arrays are stored in the computer's memory in row-major order. In FORTRAN, elements of arrays are stored in column-major order. Therefore, when passing multi-dimensional arrays from APL to FORTRAN subroutines, the programmer must be aware of this difference in array ordering.

For example, consider the following 3-row, 4-column matrix:

```
11 12 13 14
21 22 23 24
31 32 33 34
```

In APL, the elements would be stored in the following order:

```
11 12 13 14 21 22 23 24 31 32 33 34
```

FORTRAN would store this array in the following order:

```
11 21 31 12 22 32 13 23 33 14 24 34
```

Note that this difference only applies to 2-dimensional arrays (i.e., matrices) and arrays of higher dimension; lists (i.e., vectors) are stored the same in both languages.

There are two ways to compensate for this difference in array ordering when calling FORTRAN programs from APL. First, if the FORTRAN program is designed and written specifically to be used in conjunction with a particular APL application, the easiest and most efficient method is to design the FORTRAN program logic to use the APL storage order. This means that, for a matrix, the FORTRAN program does row operations in place of column operations, and column operations in place of row operations.

As a very simple example, suppose we want an external function *FROW* to accept as input a 3-by-4 matrix and return a length-4 vector containing the first row of the matrix. A use of *FROW* could be the following sequence:

```
      A
11 12 13 14
21 22 23 24
31 32 33 34

      FROW A (0 0 0 0)
11 12 13 14
```

FROW has the following listing:

```
      SUBROUTINE FROW(M,V)
      INTEGER*4 M(4,3),V(4)
      DO 10 I = 1, 4
      V(I) = M(I,1)
   10 CONTINUE
      RETURN
      END
```

The NICKNAME entry for *FROW* could be defined as

```
:nick.FROW      :load.APLDEMO
                :memb.FROW
                :link.FORTRAN
                :rarg. (G0 1 2) (I4 2 3 4) (<I4 1 4)
```

For arrays of dimension greater than 2, this technique can become somewhat confusing to a programmer; in fact, for matrices it takes some mental adjustments when thinking about the design and writing of the code. The key idea to remember is that FORTRAN storage order cycles the *leftmost* subscript most rapidly, while for APL it is the *rightmost* subscript that cycles most

rapidly. Thus simply reversing the order of subscripts in FORTRAN array references will usually do the job.

The second method for compensating for difference in storage order is to transpose multi-dimensional arrays in APL before passing them to FORTRAN routines, and, if appropriate, transpose multi-dimensional array results that are returned. The APL primitive function *transpose*, denoted by the symbol '⍉', accomplishes this operation. This is the preferred method when using pre-existing FORTRAN routines that may, for example, be part of a subroutine library. For large arrays, this method can become computationally expensive, especially if done frequently.

Part 2: APL-FORTRAN Programming Examples

Data Analysis: Dice Simulation

This simple data analysis example illustrates the functional programming style of APL. In this mode of APL application design, a series of computational steps are each performed by separate functional units, with the result of one functional unit becoming the operand of the succeeding functional unit. Because functional units are independent, they can be 'unplugged' and replaced by functionally equivalent units; this allows experimentation with various implementation strategies and fine-tuning of the application.

This example uses the APL2 function *CHARTX* described in the Appendix. *CHARTX* is a versatile APL2 interface to the Graphical Data Display Manager/Interactive Chart Utility [IBM84A].

Simulating Dice Throws

The function *DICE* is used to simulate a prescribed number of rolls of a pair of dice.

```
    ∇
[0]   Z←DICE N
[1]  ⍝ SIMULATES ROLLING DICE <N> TIMES
[2]   Z←?(N,2)⍴6
    ∇

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

The argument *N* is the number of rolls to simulate. The result of executing *DICE* is an *N*-by-2 matrix, each row of which represents a dice roll. *DICE* uses the APL function *roll*, denoted by '?', that produces random numbers; in this particular application, the elements of the result are picked from the pseudo-random uniform distribution in the range 1 to 6.

The function *COUNT* can be used with *DICE* to summarize the results of a series of dice rolls.

```
    ∇
[0]   Z←COUNT A
[1]  ⍝ COUNTS DICE THROWS IN <A>
[2]   Z←+/A
[3]   Z←+/(1+⍳11)∘.=Z
    ∇

      A←DICE 7
      A
2 5
2 4
2 5
6 3
1 1
4 2
5 2
```

```
      COUNT A
1 0 0 0 2 3 0 1 0 0 0

      COUNT DICE 50
2 1 2 9 6 10 9 6 4 1 0

      COUNT DICE 500
14 33 37 50 65 89 73 42 58 22 17
```

The argument to *COUNT* is a dice-roll series produced by *DICE*. *COUNT* computes the sum of the two dice values for each roll, and tabulates the totals of each sum in the series. The result *Z* is a length-11 integer list; *Z*[1] contains the number of 2s rolled in the series, *Z*[2] contains the number of 3s, etc. Note that by definition, the sum of *Z* equals the number of rolls.

The function *EXPECT* computes the expected number of dice-pair sums for a prescribed number of rolls.

```
    ∇
[0]   Z←EXPECT N;T
[1]  ⍝ EXPECTED NUMBER OF EACH SUM FOR <N> DICE THROWS
[2]   Z←N×(T[⍋T←⍳11])÷36
    ∇

      EXPECT 36
1 2 3 4 5 6 5 4 3 2 1

      EXPECT 72
2 4 6 8 10 12 10 8 6 4 2

      EXPECT 500
13.9 27.8 41.7 55.6 69.4 83.3 69.4 55.6 41.7 27.8 13.9
```

The argument to *EXPECT* is the number of dice rolls. The result of *Z* is a list of length 11; *Z*[1] gives the number of expected occurrences of 2s in *N* rolls, *Z*[2] gives the number of expected occurrences of 3s, etc.

Graphical Analysis of Dice Throws

The function *DRAW* is used to plot the actual and expected results of a dice roll series. The main component of *DRAW* is the *CHARTX* function described in the Appendix.

DRAW accepts a two-item list. The first item is the actual results of dice roll simulations as generated by *DICE* and *COUNT*; the second item is a list of expected dice roll results as computed by *EXPECT*.

```
    ∇
[0]   DRAW D;FORMNAME
[1]  ⍝ CHARTS ACTUAL AND EXPECTED DICE ROLLS
[2]   FORMNAME←'DICE'
[3]   (1+⍳11)CHARTX⊃D
    ∇
```

The following APL expression generates the plot in Figure 1.

```
      DRAW (COUNT DICE 36) (EXPECT 36)
```

Results of the following expression are shown in Figure 2.

```
      DRAW (COUNT DICE 100) (EXPECT 100)
```

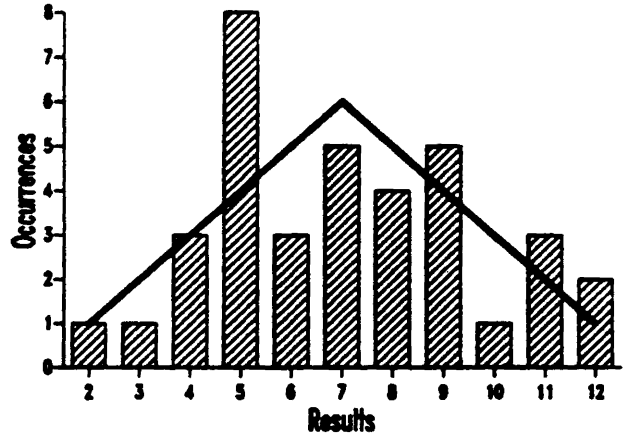

Figure 1: 36 dice throws

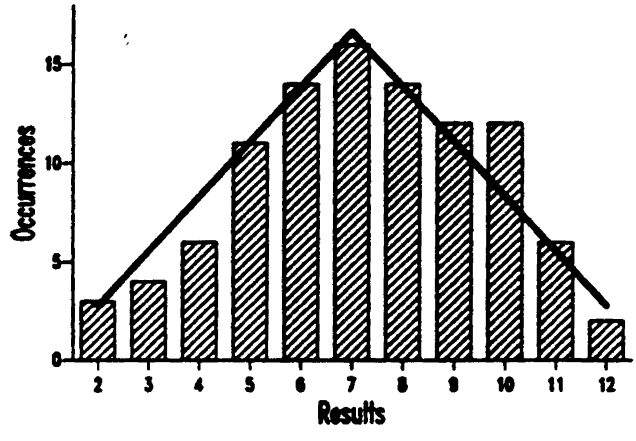

Figure 2: 100 dice throws

Note that as the number of simulated rolls increases, the actual occurrences come closer to the expected occurrences, giving an empirical confirmation of the statistical law of large numbers (see Feller, [FEL50A]). The following expression simulates 1000 dice rolls; the result is shown in Figure 3.

```
DRAW (COUNT DICE 1000) (EXPECT 1000)
```

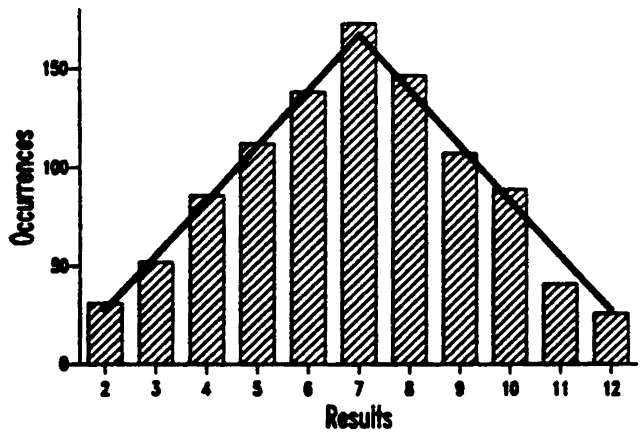

Figure 3: 1000 dice throws

Substituting FORTRAN for APL

So far, we have discussed computational components for analyzing dice rolls that are coded in APL; now let us introduce some FORTRAN components.

The external routine *COUNTF*, coded in FORTRAN, is functionally equivalent to the APL function *COUNT*; it computes the sum of individual dice rolls in a series, and tabulates the totals of each sum. *COUNTF* has the following definition:

```
      SUBROUTINE COUNTF(N,D,Z)
      INTEGER*4 N,D(2,N),Z(11)
      DO 10 I = 1, 11
   10 Z(I)=0
      DO 20 I = 1, N
      J=D(1,I) + D(2,I)
      Z(J-1)=Z(J-1)+1
   20 CONTINUE
      RETURN
      END
```

COUNTF could have the following NICKNAME file entry:

```
:nick.COUNTF    :load.APLDEMO
                :memb.COUNTF
                :link.FORTRAN
                :rarg. (G0 1 3) (1 I4 *) (I4 2 * 2) (< I4 1 11)
```

COUNTF can be imbedded in the APL function *COUNTX* that has the same calling sequence as the APL function *COUNT*.

```
    ▼
[0]   Z←COUNTX A
[1]   Z←COUNTF(1↑ρA)A(11ρ0)
    ▼

      A←DICE 10
      A
2 5
2 4
2 5
6 3
1 1
4 2
5 2
5 3
4 4
5 2
```

```
      COUNTX A
1 0 0 0 2 4 2 1 0 0 0

      COUNTX DICE 100
2 4 7 12 16 15 16 9 10 5 4
```

Likewise, *EXPECT* can be replaced with a FORTRAN program *XPECTF*, used with a cover function *EXPECTX*.

```
    ▼
[0]   Z←EXPECTX N
[1]   Z←XPECTF N(11⍴0)
    ▼

      EXPECTX 36
1 2 3 4 5 6 5 4 3 2 1

      EXPECTX 72
2 4 6 8 10 12 10 8 6 4 2

      EXPECTX 500
13.89 27.78 41.67 55.56 69.44 83.33 69.44 55.56 41.67 27.78 13.89
```

XPECTF has the following definition:

```
      SUBROUTINE XPECTF(N,E)
      INTEGER*4 N
      REAL*8 E(11),T
      DO 10 I = 1, 5
      T=DFLOAT(I*N)/36.D0
      E(I)=T
      E(12-I)=T
   10 CONTINUE
      E(6)=DFLOAT(6*N)/36.D0
      RETURN
      END
```

The NICKNAME file entry for *XPECTF* could take the following form:

```
:nick.XPECTF    :load.APLDEMO
                :memb.XPECTF
                :link.FORTRAN
                :rarg. (G0 1 2) (1 I4 *) (< E8 1 11)
```

Finally, the following expression simulates, tabulates, and plots 10000 dice roll simulations; the result is shown in Figure 4:

```
      DRAW (COUNTX DICE 10000) (EXPECTX 10000)
```

Three of the functions in this expression perform their operations in APL; two perform their operations in FORTRAN.

Array Data Structures: Sparse Matrix Operations

In a 'sparse matrix' most elements have the same value, usually zero. Large matrices that contain a small percentage of nonzero elements can generally be stored and used more efficiently in some format other than the normal 2-dimensional representation. The following describes a format that is commonly used for storing large sparse matrices involved in computations on linear equations (computing inner products and inverses, solving linear equation systems, solving linear programming problems, etc.) This representation assumes that all matrix columns contain at least one nonzero element.

If *M* is a matrix, then *SM* is a sparse representation of *M*:

```
SM ↔ E JX IP
```

E is a list of the nonzero elements of *M* in row-major order. *JX* is a list of the corresponding column indices of the nonzero elements of *M*. That is, the element *E*[*K*] comes from column *JX*[*K*] of *M*. Lists *E* and *JX* have the same length. *IP* is a list of pointers into *E* (and *JX*) that indicate the position of the first nonzero element in each row of *M*. The length of *IP* is one more that the number of rows of *M*.

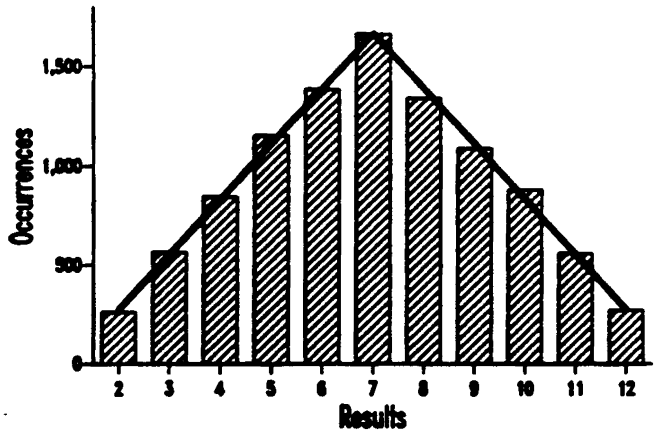

Figure 4: 10000 dice throws

For example, the matrix *A* is represented in sparse form by the array *SA*:

```
      A
1.1 0   1.3 1.4 0
0   2.2 0   2.4 0
0   3.2 3.3 0   3.5

      ⍴ A
3 5

SA←→(1.1 1.3 1.4 2.2 2.4 3.2 3.3 3.5)(1 3 4 2 4 2 3 5)(1 4 6 9)

      ⍴ SA
3

      ⍴⍴ SA
 8  8  4
```

Useful APL functions for handling sparse matrices are the *PACK* and *UNPACK* routines. *PACK* operates on matrices in a 2-dimensional format and produces sparse matrix data structures:

```
    ▼
[0]   Z←PACK M;E;JX;IP
[1]  ⍝ PACK MATRIX <M> AS ROW-WISE SPARSE MATRIX <Z>
[2]   E←(,0≠M)/,M                    ⍝ COEFFICIENTS
[3]   JX←(,M←M≠0)/,(⍴M)⍴⍳1↓⍴M       ⍝ COLUMN INDICES
[4]   IP←+\1,+/M                     ⍝ ROW POINTERS
[5]   Z←E JX IP
    ▼

      PACK A
 1.1 1.3 1.4 2.2 2.4 3.2 3.3 3.5  1 3 4 2 4 2 3 5  1 4 6 9
```

UNPACK operates on sparse matrix data structures to produce a 2-dimensional format:

```
    ▼
[0]   Z←UNPACK S;E;JX;IP;M;N
[1]  ⍝ UNPACK ROW-WISE SPARSE MATRIX <S> TO CREATE MATRIX <Z>
[2]   (E JX IP)←S
[3]   M←¯1+⍴IP   ⍝ NUMBER OF ROWS
[4]   N←⌈/JX     ⍝ NUMBER OF COLS
[5]   Z←(N×M)⍴0
[6]   Z[JX+N×(¯2-/IP)/¯1+⍳M]←E
[7]   Z←(M,N)⍴Z
    ▼

      UNPACK SA
1.1 0   1.3 1.4 0
0   2.2 0   2.4 0
0   3.2 3.3 0   3.5

      UNPACK PACK A
1.1 0   1.3 1.4 0
0   2.2 0   2.4 0
0   3.2 3.3 0   3.5
```

Sparse Matrix Transpose

A common operation in many sparse matrix applications is *matrix transpose*. Transposing a matrix switches the orientation of its rows and columns. In APL, the *transpose* function, denoted by the symbol '⍉', is used to transpose 2-dimensional matrices:

```
      A
1.1 0   1.3 1.4 0
0   2.2 0   2.4 0
0   3.2 3.3 0   3.5

      ⍉ A
1.1 0   0
0   2.2 3.2
1.3 0   3.3
1.4 2.4 0
0   0   3.5
```

The function *STA* is used in an analogous manner to transpose matrices in sparse format:

```
    ▼
[0]   COL←STA ROW;AR;JC;IP;AC;IR;JP;T
[1]  ⍝ SPARSE MATRIX TRANSPOSE IN APL
[2]   (AR JC IP)←ROW
[3]   T←⍋JC                              ⍝ REORDER SEQUENCE
[4]   AC←AR[T]                           ⍝ COEFFICIENTS
[5]   IR←(∊(¯2-/IP)⍴¨⍳¯1+⍴IP)[T]          ⍝ ROW INDICES FROM ROW POINTER
[6]   JP←(1,(2=/JC[T]),1)/⍳1+⍴AC         ⍝ COLUMN POINTER FROM COL INDICES
[7]   COL←AC IR JP
    ▼

      SA
 1.1 1.3 1.4 2.2 2.4 3.2 3.3 3.5  1 3 4 2 4 2 3 5  1 4 6 9

      UNPACK SA
1.1 0   1.3 1.4 0
0   2.2 0   2.4 0
0   3.2 3.3 0   3.5

      STA SA
 1.1 2.2 3.2 1.3 3.3 1.4 2.4 3.5  1 2 3 1 3 1 2 3  1 2 4 6 8 9
```

```
      UNPACK STA SA
1.1 0   0
0   2.2 3.2
1.3 0   3.3
1.4 2.4 0
0   0   3.5
```

While *STA* is a concise routine for performing sparse matrix transpose, it suffers an intrinsic performance penalty. In particular, the *grade* function in line [3], denoted by '⍋', makes the execution time of *STA* proportional to *n* log *n*, where *n* is the number of nonzero elements in the matrix argument.

A more efficient algorithm for sparse matrix transpose, due to Suhl [SUH81A], is easily implemented in FORTRAN. This routine, named *STF*, is functionally equivalent to *STA*; its execution time is proportional to the number of nonzero elements in the matrix argument.

```
      SUBROUTINE STF(M,N,K,CA,IXA,IPA,CZ,IXZ,IPZ)
C     SPARSE MATRIX TRANSPOSE
C     INPUTS:
C       M - NUMBER OF ROWS
C       N - NUMBER OF COLUMNS
C       K - NUMBER OF NONZERO COEFFICIENTS
C       CA - VECTOR OF COEFFICIENTS IN ROW-ORDER (LENGTH K)
C       IXA - VECTOR OF CORRESPONDING COLUMN INDICES (K)
C       IPA - VECTOR OF ROW POINTERS (M+1)
C     OUTPUTS:
C       CZ - VECTOR OF COEFFICIENTS IN COLUMN-ORDER (K)
C       IXZ - VECTOR OF CORRESPONDING ROW INDICES (K)
C       IPZ - VECTOR OF COLUMN POINTERS (N+1)
      INTEGER*4 M,N,K,IXA(K),IPA(M+1),IXZ(K),IPZ(N+1)
      REAL*8 CA(K),CZ(K)
C     CLEAR IPZ...
      DO 10 I = 2,N+1
   10 IPZ(I)=0
C     COUNT ELEMENTS IN EACH COLUMN...
      DO 20 I = 1,K
      J = 1+IXA(I)
   20 IPZ(J) = IPZ(J) + 1
C     COMPUTE INITIAL COLUMN OFFSETS...
      IPZ(1) = 1
      DO 30 I = 2,N
   30 IPZ(I) = IPZ(I) + IPZ(I-1)
C     MOVE COEFFICIENTS AND ROW INDICES...
      DO 40 I = 1,M
      DO 40 J = IPA(I),IPA(I+1)-1
      L1 = IXA(J)
      L = IPZ(L1)
      CZ(L) = CA(J)
      IXZ(L) = I
      IPZ(L1) = IPZ(L1) + 1
   40 CONTINUE
C     RESET COLUMN POINTER...
      J = N + 2
      DO 50 I = 1,N
   50 IPZ(J-I) = IPZ(J-I-1)
      IPZ(1) = 1
      RETURN
      END
```

STF has the following APL2 NICKNAME file entry:

```
:nick.STF       :memb.STF
                :load.APLDEMO
                :link.FORTRAN
                :rarg. (G0 1 9)
                       (1 I4 *)
                       (1 I4 *)
                       (1 I4 *)
                       (E8 1 *)
                       (I4 1 *)
                       (I4 1 *)
                       (< E8 1 *)
                       (< I4 1 *)
```

The use of *STF* can be simplified using the cover function *ST*:

```
     ▼
[0]    Z←ST SM;K;M;N;E;JX;IP
[1]    (E JX IP)←SM
[2]    M←¯1+⍴IP      ⍝ NUMBER OF ROWS
[3]    N←⌈/JX        ⍝ NUMBER OF COLS
[4]    K←⍴E          ⍝ NUMBER OF COEFFICIENTS
[5]    Z←STF (M N K),SM,(K⍴0)(K⍴0)((N+1)⍴0)
     ▼

      UNPACK SA
1.1 0   1.3 1.4 0
0   2.2 0   2.4 0
0   3.2 3.3 0   3.5

      UNPACK ST SA
1.1 0   0
0   2.2 3.2
1.3 0   3.3
1.4 2.4 0
0   0   3.5
```

The difference in execution times between the APL and FORTRAN versions is demonstrated by a simple performance comparison. The execution times for transposing a series of 100-column matrices with varying numbers of rows were recorded using the two routines. The test matrices were all 90 percent sparse; that is, 90 percent of the matrix elements were zero. Figure 5 plots the results of running these tests on a IBM 4381 processor.

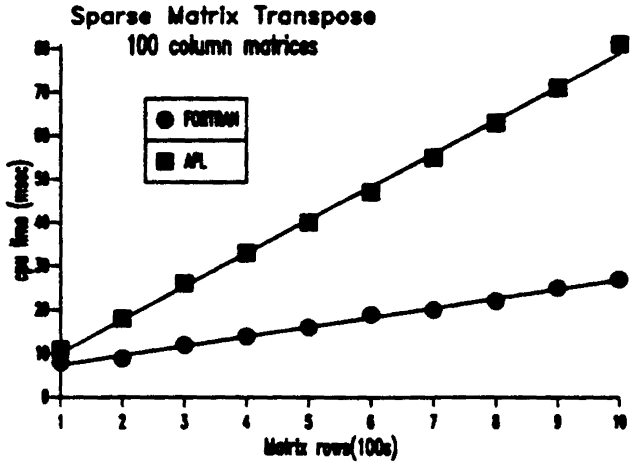

Figure 5: Spare matrix transpose

Matrix-Vector Inner Product

Another common array operation is the inner product, sometimes called '*dot product*', between a vector and the rows of a matrix. For arrays in nonsparse format, this operation is built into APL as the *inner product* operator, denoted as '*F.G*', where *F* and *G* are functions:

```
      A
1.1 0   1.3 1.4 0
0   2.2 0   2.4 0
0   3.2 3.3 0   3.5

      B
2 3 0 0 1

      A +.× B
2.2 6.6 13.1
```

The APL function *SMPA* performs this operation for sparse matrix arguments:

```
    ▼
[0]   Z←SM SMPA V;A;JX;IP
[1]   (A JX IP)←SM
[2]   Z←A×V[JX]
[3]   IP←0,¯2-/IP
[4]   Z←+/¨(1+IP)↑¨(¯1++\IP)↓¨⊂Z
    ▼

      SA
 1.1 1.3 1.4 2.2 2.4 3.2 3.3 3.5  1 3 4 2 4 2 3 5  1 4 6 9

      SA SMPA B
2.2 6.6 13.1
```

A functionally equivalent FORTRAN program *SMPF* can be used to perform this operation. *SMPF* has the following definition:

```
      SUBROUTINE SMPF(M,N,K,A,JX,IP,V,Z)
C---- SPARSE MATRIX-VECTOR INNER PRODUCT
C     INPUTS:
C       M - NUMBER OF ROWS
C       N - NUMBER OF COLUMNS
C       K - NUMBER OF NONZERO COEFFICIENTS
C       A - VECTOR OF COEFFICIENTS IN ROW-ORDER (LENGTH K)
C       JX - VECTOR OF CORRESPONDING COLUMN INDICES (K)
C       IP - VECTOR OF ROW POINTERS (M+1)
C       V - FULL VECTOR OPERAND (N)
C     OUTPUTS:
C       Z - FULL VECTOR RESULT (M)
      INTEGER*4 M,N,K,JX(K),IP(M+1)
      REAL*8 A(K),V(N),Z(M)
      DO 10 I = 1,M
      Z(I)=0.
      DO 10 J = IP(I), IP(I+1)-1
      Z(I)=Z(I)+V(JX(J))*A(J)
   10 CONTINUE
      RETURN
      END
```

The NICKNAME file entry for *SMPF* takes the following form:

```
:nick.SMPF      :memb.SMPF
                :load.APLDEMO
                :link.FORTRAN
                :rarg. (G0 1 8)
                       (1 I4 *)
                       (1 I4 *)
                       (1 I4 *)
                       (E8 1 *)
                       (I4 1 *)
                       (I4 1 *)
                       (E8 1 *)
                       (< E8 1 *)
```

The cover function *SMP* simplifies the use of *SMPF*:

```
    ∇
[0]   Z←SM SMP V;K;M;N;E;JX;IP
[1]   (E JX IP)←SM
[2]   M←¯1+⍴IP      ⍝ NUMBER OF ROWS
[3]   N←⌈/JX        ⍝ NUMBER OF COLS
[4]   K←⍴E          ⍝ NUMBER OF COEFFICIENTS
[5]   Z←SMPF(M N K),SM,V(M⍴0)
    ∇

      SA SMP B
2.2 6.6 13.1
```

Again, execution efficiencies for this operation can be realized using the FORTRAN external function. Varying the numbers of matrix columns, a series of inner products between vectors and 100-row sparse matrices were performed. The matrices were 90-percent sparse. The execution times for both the APL and FORTRAN routines were recorded. Figure 6 shows the results of running these tests on a IBM 4381 processor.

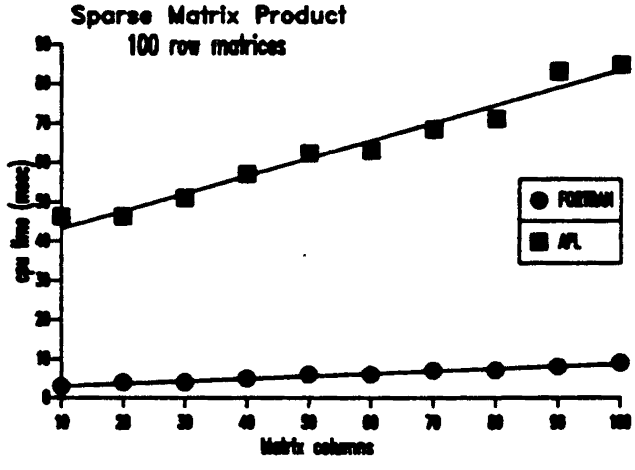

Figure 6: Matrix-vector inner product

These sparse matrix processing examples are typical of the kinds of computations that elude good APL solutions; the structure of the problem does not allow the efficient use of APL array-oriented operations. For these kinds of problems, external functions coded in FORTRAN offer increased processing power for APL applications.

Part 3: Performance Analysis using APL Defined Operators

Performance analysis tools are used to measure, tabulate and interpret the execution efficiencies of computer programs. Such analysis is used to answer questions such as: Given two variations of an algorithm or program, what are their relative or absolute efficiencies when executed in the same computing environment? Or, given two computing environments (e.g., two different processors), what are their relative or absolute efficiencies executing the same routine? Performance analysis also addresses issues such as how execution times change as a function of problem size or complexity, the effect of variations in the computing environment (e.g., compiler optimization levels, computer memory size), and, for multiprogramming environments, the effect of machine load on execution performance.

In this section, we describe a simple performance analysis tool that is useful for analyzing a program's execution efficiency for variations in problem size. It is implemented in APL2 and takes advantage of many APL2 concepts and facilities; it is, however, especially useful for measuring, tabulating and reporting the execution performance of FORTRAN subroutines.

Note: While execution performance is an important consideration for determining the usefulness and applicability of a computer algorithm or program, it is only one of many factors that should be considered when selecting a software solution for a particular problem. For a general discussion of evaluating alternate software solutions, see Crowder, *et. al.* [CRO79A].

Test Problem Generators

If we have an APL function, say *FN*, to be performance-tested, then a *test problem* for *FN* is an array that is an appropriate argument for *FN*. For example, recall the APL function *SDA* for computing standard deviations. A valid test problem for *SDA* is a numeric list.

To do performance analysis for a range of test problem sizes, it is helpful to have an automatic method for generating test problems. To accomplish this, we define for *FN* a related function, *FNG*, called the *FN test problem generator*. The argument to *FNG* is a single integer *N* that specifies the *size* of a desired test problem; the result is a test problem of size *N* for *FN*. For example, *SDAG* generates test problems for *SDA*:

```
    ▼
[0]   Z←SDAG N
[1]   Z←.1×?Nρ10
    ▼

      SDAG 3
0.2 0.3 0.5

      SDAG 5
0.2 0.7 0.7 0.5 0.4
```

SDA and *SDAG* can be used in conjunction to generate and use test problems:

```
      SDA SDAG 10
0.2416609195

      SDA SDAG 100
0.2711899703

      SDA SDAG 1000
0.2873239287
```

In an analogous manner, *SDFG* generates proper test problem arrays for the FORTRAN version of the standard deviation calculation *SDF*:

```
    ∇
[0]   Z←SDFG N
[1]   Z←0 N(.1×?Nρ10)
    ∇
      SDFG 3
 0 3  0.4 0.2 0.3

      SDFG 5
 0 5  0.7 0.1 0.8 0.6 0.8
```

To perform timing comparisons, the second element of the APL2 system variable ⎕AI is referenced to obtain elapsed processor time in milliseconds:

```
      A←SDAG 10000
      T←⎕AI[2]
      JUNK←SDA A
      ⎕AI[2]-T
156

      A←SDFG 10000
      T←⎕AI[2]
      JUNK←SDF A
      ⎕AI[2]-T
68
```

Thus the APL version required 156 milliseconds to compute the standard deviation of 10000 floating point numbers on an IBM 4381 processor; the FORTRAN version required 68 milliseconds.

While the operational sequence described above is useful, it can become tedious. Fortunately, APL2 offers an elegant method for packaging this sequence into a more useful format.

The FUNTIME Defined Operator

FUNTIME is an APL2 defined operator; it takes two function operands and an array argument:

```
    ∇
[0]   Z←(FN FUNTIME FNG) N;T
[1]   N←FNG N                ⍝ CONSTRUCT FN ARG LIST
[2]   Z←⎕AI[2]               ⍝ TIME
[3]   T←FN N                 ⍝ INVOKE FN
[4]   Z←⎕AI[2]-Z             ⍝ TIME
    ∇
```

FN is the function to be timed and *FNG* is a test problem generator appropriate for *FN*. The single integer *N* is the test problem size to be generated by *FNG*. The result Z is the execution time in milliseconds of applying *FN* to the test problem *FNG N*.

FUNTIME can be used to perform a single test:

```
      (SDA FUNTIME SDAG) 10000
154

      (SDF FUNTIME SDFG) 10000
70
```

Used in conjunction with the *each* primitive operator, *FUNTIME* can be used to perform a series of tests:

```
      ⍳8
1 2 3 4 5 6 7 8

      2000×⍳8
2000 4000 6000 8000 10000 12000 14000 16000

      (SDF FUNTIME SDFG)¨ 2000×⍳8
15 28 41 55 68 82 96 109
```

Because APL2 can distinguish the structure of arrays using the *depth* function, denoted by '≡', handling nonsingle array arguments using *each* can be incorporated into the *FUNTIME* operator:

```
    ▼
[0]   Z←(FN FUNTIME FNG) N;T
[1]   →(0<≡N)/L1          ⍝ BRANCH IF NOT INTEGER
[2]   N←FNG N             ⍝ CONSTRUCT FN ARG LIST
[3]   Z←⎕AI[2]            ⍝ TIME
[4]   T←FN N              ⍝ INVOKE FN
[5]   Z←⎕AI[2]-Z          ⍝ TIME
[6]   →0
[7]  L1:
[8]   Z←(FN FUNTIME FNG)¨ N
    ▼

      (SDF FUNTIME SDFG) 2000×⍳8
14 28 42 55 69 81 95 108
```

FUNTIME can be used in conjunction with the *CHARTX* function to picture performance statistics. The graphical result of the following sequence is shown in Figure 7.

```
      T←2000×⍳10

      A←(SDF FUNTIME SDFG) T
      A
15 28 42 55 68 82 95 109 122 135

      B←(SDA FUNTIME SDAG) 2000×⍳10
      B
32 63 93 123 154 182 213 243 272 303

      T CHARTX ⊃A B
```

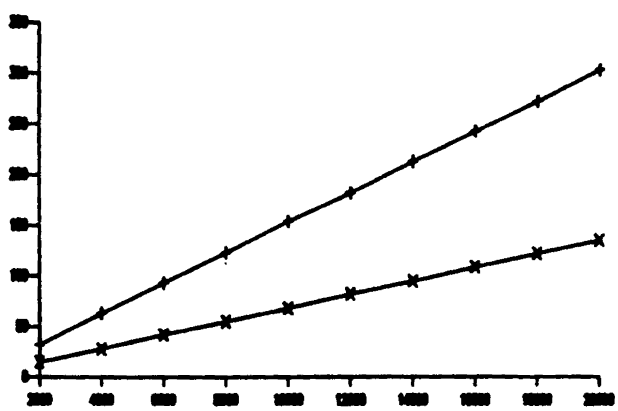

Figure 7: Plotting performance data

Reporting the average of several similar performance experiments is often desirable, especially on multiprogramming computer systems where transient user load can introduce slight variations in individual execution times. Using the *AVG* function in conjunction with both *FUNTIME* and the *enclose* function (denoted by '⊂') can be useful for this type of analysis:

```
      N←5 8ρ2000×ι8
      N
2000 4000 6000 8000 10000 12000 14000 16000
2000 4000 6000 8000 10000 12000 14000 16000
2000 4000 6000 8000 10000 12000 14000 16000
2000 4000 6000 8000 10000 12000 14000 16000
2000 4000 6000 8000 10000 12000 14000 16000

      Z←(SDF FUNTIME SDFG) N
      Z
14 28 42 55 68 82 95 109
15 28 41 55 68 81 94 108
15 28 42 55 68 82 95 110
15 28 41 55 68 82 94 109
14 27 42 54 68 82 95 109

      AVG¨ c[1] Z
 14.6  27.8  41.6  54.8  68  81.8  94.6  109
```

More concisely, the three previous steps may be combined as

```
      AVG¨ c[1] (SDF FUNTIME SDFG) 5 8ρ 2000×ι8
 14.6  27.8  41.6  54.8  68  81.8  94.6  109
```

References

[CRO79A] Crowder, H., R. Dembo, and J. Mulvey, 'On Reporting Computational Experiments With Mathematical Software,' *ACM Transactions on Mathematical Software* 5 (1979) pp. 193-203.

[FEL50A] Feller, W., *An Introduction to Probability and Its Applications: Volume I*, Wiley (1950).

[IBM83A] *VM/SP System Product Interpreter Reference*, IBM form number SC24-5239 (1983).

[IBM84A] *GDDM Interactive Chart Utility*, IBM form number SC33-0111 (1984).

[IBM85A] *APL2 Programming: System Services Reference*, IBM form number SH20-9218 (1985).

[IBM86A] *VS FORTRAN Version 2 Programming Guide*, IBM form number SC26-4222 (1986).

[SUH81A] Suhl, U., personal communication.

Appendix: CHARTX - An APL2 / ICU Data Interface

CHARTX is an APL2 function that offers a call interface to the Graphics Data Display Manager Interactive Chart Utility (ICU). Data can be passed to ICU in a variety of formats using APL2 general arrays. *CHARTX* also offers a facility for using predefined ICU chart formats.

Free and Tied Data

ICU allows the simultaneous graphical display of several 'groups' of data. For example, a graph with three line plots would have the data for each line plot represented as a data group. ICU makes the distinction between two types of data format modes for representing data groups -- *free data* and *tied data*. In free mode, each data group has its own set of X values or coordinates; each group's X values are independent of other groups. In tied mode, all data groups have the same set of X values.

CHARTX handles both ICU data format modes; the mode is determined from the structure of the arguments to *CHARTX*.

Using CHARTX for Tied Data

For tied data, *CHARTX* has the following call sequence:

```
XT CHARTX YT
```

where

YT is the array of Y values. *YT* is a simple numeric scalar, vector, or matrix. If *YT* is a scalar or vector, it forms one data group containing the element(s) of *YT*. If *YT* is a matrix with *M* rows and *N* columns, it forms *M* data groups, each group consisting of *N* elements from the rows of *YT*.

XT is the simple numeric array of X values. If *YT* is a scalar or vector, then *XT* must be the same shape as *YT*. If *YT* is a matrix, then *XT* must be a vector, the length of which is the same as the number of columns of *YT*. That is, $\rho XT \leftrightarrow {}^{-}1\uparrow\rho YT$.

If *XT* is not specified, *CHARTX* uses a default X-coordinate vector consisting of consecutive integers that is appropriate for *YT*, starting with $\square IO$.

Examples

```
CHARTX 12 22 18 32 7     ↔   (⍳5) CHARTX 12 22 18 32 7

1 2 5 6 9 CHARTX 12 22 18 32 7

CHARTX 1 10 ∘.× ⍳12      ↔   (⍳12) CHARTX 1 10 ∘.× ⍳12

(2∘⍳12) CHARTX 1 10 ∘.×⍳12

CHARTX 1 2 ∘.○ 0.1×⍳120
```

The results of executing these expressions are shown in Figure 8.

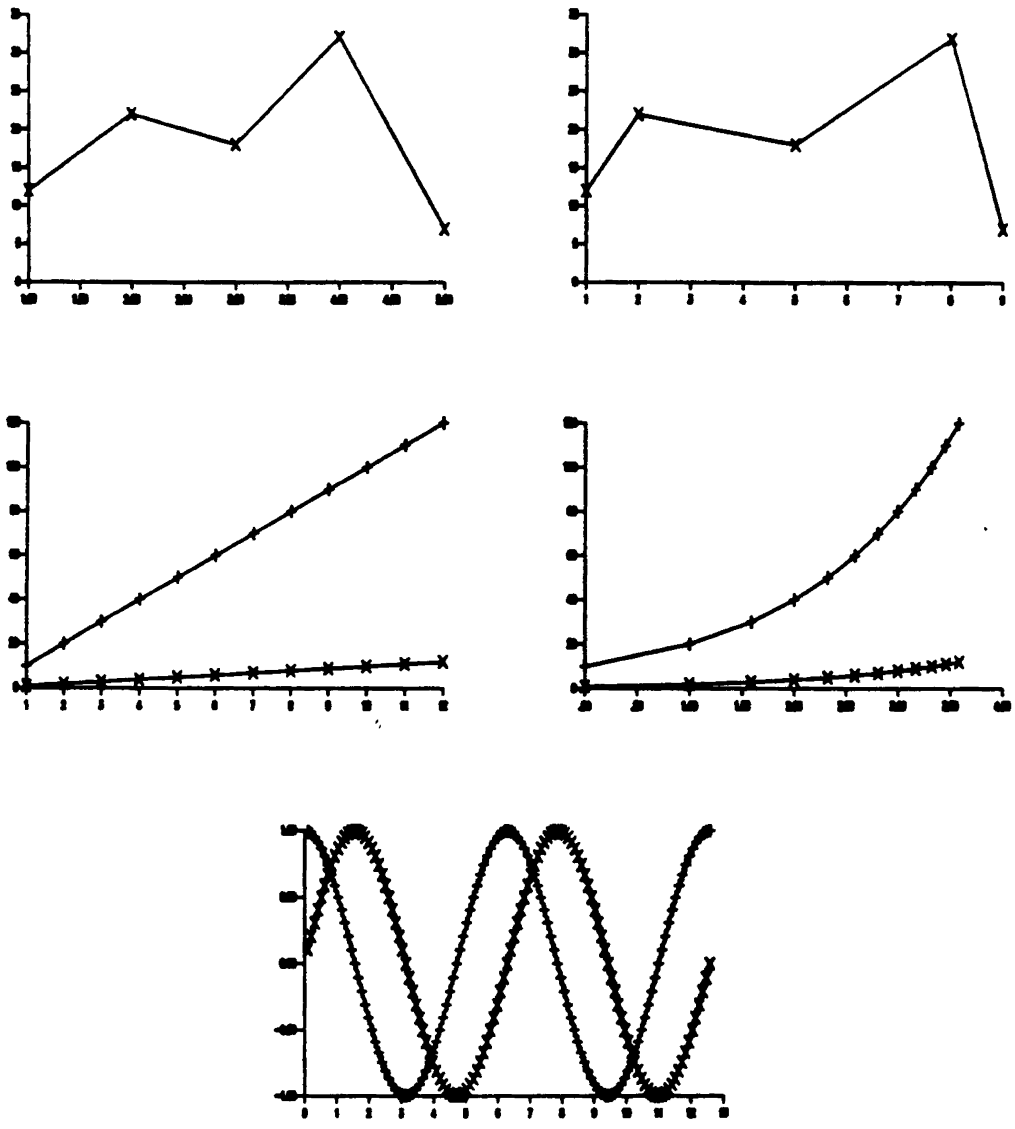

Figure 8: ICU tied data examples

Using CHARTX for Free Data

For free data, *CHARTX* has the following call sequence:

```
XF CHARTX YF
```

where

YF is the array of Y values. *YF* is a numeric vector of depth 2, each item of which is a simple scalar or vector. Each item of *YF* forms an independent data group.

XF is the array of X values. *XF* must have the same structure as *YF*. Items of *XF* form the X-coordinates for corresponding items of *YF*.

If *XF* is not specified, *CHARTX* uses a default X-coordinate array, each item of which consists of consecutive integers, starting with ⎕*IO*, that is appropriate for the corresponding item in *YF*.

Examples

```
CHARTX (3 7 16) (10 14 8 3 0)
                      ⟷ (⍳3) (⍳4) CHARTX (3 7 16) (10 14 8 3 0)

(2 3 4) (⍳5) CHARTX (3 7 16) (10 14 8 3 0)

CHARTX ?⍨⍳⍨5⍴10
```

The results of executing these expressions are shown in Figure 9.

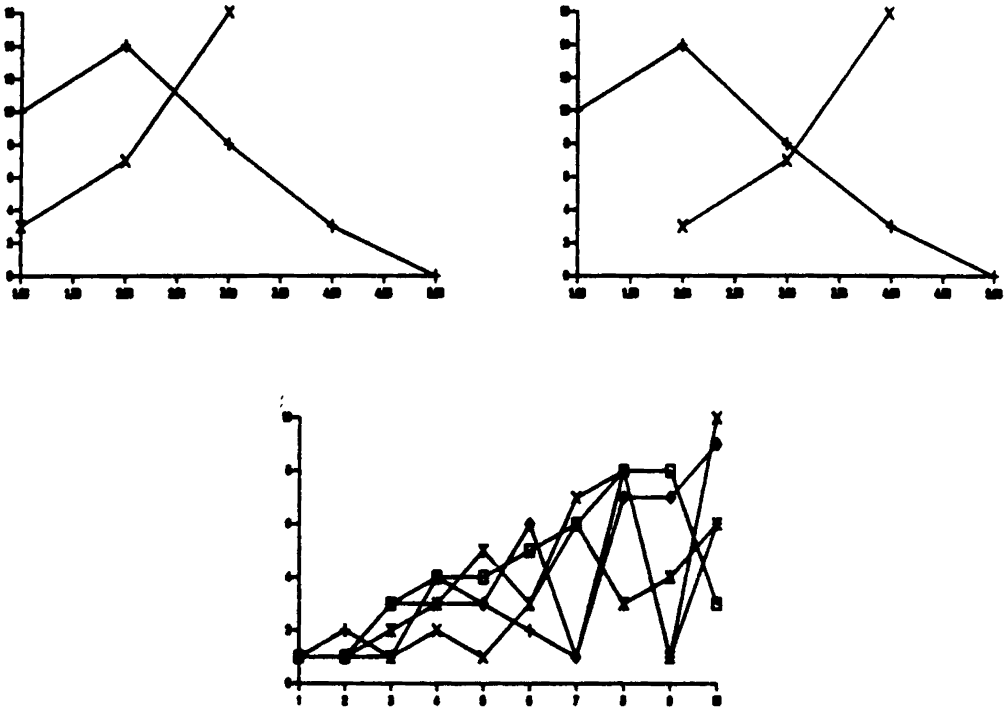

Figure 9: ICU free data examples

Usage Notes

1. The global variable *FORMNAME* can be specified as the name of a predefined chart format. If *FORMNAME* is undefined, or if *FORMNAME* has the value '*', then the default format is used. The default format is a line graph with autoscaled axes, default line colors, default axis markers and labels, etc. If *FORMNAME* is assigned the name of an unknown chart format, then an error message is issued.

2. Some facilities available in the ICU chart call are not used by *CHARTX*. These include specification of chart keys, labels, and heading. If you wish to use these facilities, you must modify the *CHARTX* function.

3. The main purpose of *CHARTX* is to allow the quick and easy generation of data in APL2, and to provide a mechanism for transferring this data to ICU. Once in the ICU environment, you can modify the chart type and format to suit your needs by using the ICU interactive facilities. Data transferred to ICU can be displayed using the following chart types:

Bar charts	Polar charts	Scatter plots
Histograms	Surface charts	Venn diagrams
Line graphs	Tower charts	Pie charts

```
∇
[0]   X CHARTX Y;IO;CHTCTL;DAT;CTL;BIND;NG;NE;DC;C
[1]   ⍝ Invoke ICU from APL2
[2]   C←⎕FX 'Z←IO N' 'Z←⎕⎕AF (4ρ256)⊤N'   ⍝ Local fun for integer-char. conv.
[3]   →(1<≡Y)/L1                          ⍝ Branch if free data
[4]   ⍝ TIED DATA...
[5]   Y←(¯2↑1 1,ρY)ρY                     ⍝ Ensure Y is a matrix
[6]   ⍎(0=⎕NC 'X')/'X←⍳¯1↑ρY'            ⍝ Set X if not specified
[7]   ⎕ES((ρX←,X)≠¯1↑ρY)/'LENGTH ERROR'   ⍝ Ensure correct data lengths
[8]   BIND←0                              ⍝ Set BIND parm of CHART call
[9]   NG←1↑ρY                             ⍝ Num data groups
[10]  NE←ρX                               ⍝ Num elements in each group
[11]  DC←,0                               ⍝ Data control - ignored for tied data
[12]  Y←,Y
[13]  →L2
[14]  ⍝
[15]  L1: ⍝ FREE DATA...
[16]  DC←∈ρ¨Y←,¨Y                         ⍝ Data control - length of each group
[17]  ⍎(0=⎕NC 'X')/'X←⍳¨DC'               ⍝ Set X if not specified
[18]  ⎕ES(~DC≡∈ρ¨X←,¨X)/'LENGTH ERROR'    ⍝ Ensure correct data lengths
[19]  BIND←1                              ⍝ Set BIND parm of CHART call
[20]  NG←ρY                               ⍝ Num data groups
[21]  NE←⌈/DC                             ⍝ Num elements - max group length
[22]  X←∈X
[23]  Y←∈Y
[24]  ⍝
[25]  L2:
[26]  ⎕ES(∨/2≠126 ⎕SVO¨'CTL' 'DAT')/'AP126 SHARE ERROR'
[27]  ⍝ Build CHART control parm...
[28]  CHTCTL←''
[29]  CHTCTL←CHTCTL,IO 0          ⍝ LEVEL     0=GDDM R2 format (simple form)
[30]  CHTCTL←CHTCTL,IO 2          ⍝ DISPLAY   1=home panel, 2=graph display
[31]  CHTCTL←CHTCTL,IO 0          ⍝ HELP      0=no display pfkey info, 1=display
[32]  CHTCTL←CHTCTL,IO 0          ⍝ ISOLATE   0=all facilities
[33]  ⍎(0=⎕NC 'FORMNAME')/'FORMNAME←''*'''  ⍝ default if FORMNAME undefined
[34]  CHTCTL←CHTCTL,8↑⍕∈FORMNAME  ⍝ FORMNAME  *=default
[35]  CHTCTL←CHTCTL,'*        '   ⍝ DATANAME  *=data supplied by CHART call
[36]  CHTCTL←CHTCTL,IO BIND       ⍝ BINDING   0=tied,1=free
[37]  CHTCTL←CHTCTL,,IO NG        ⍝ NG - number of data groups
[38]  CHTCTL←CHTCTL,,IO NE        ⍝ NE - number of elements
[39]  CHTCTL←CHTCTL,IO 0          ⍝ KEYL      0=no keys
[40]  CHTCTL←CHTCTL,IO 0          ⍝ LABELL    0=no labels
[41]  CHTCTL←CHTCTL,IO 0          ⍝ HEADINGL  0=no heading
[42]  CHTCTL←CHTCTL,'*        '   ⍝ PRTNAME   *=unknown printer name
[43]  CHTCTL←CHTCTL,IO 0          ⍝ PRTDEP    0=DEFAULT
[44]  CHTCTL←CHTCTL,IO 0          ⍝ PRTWID    0=DEFAULT
[45]  CHTCTL←CHTCTL,IO 0          ⍝ PRTCOPY   0=DEFAULT
[46]  DAT←CHTCTL                  ⍝ assign AP126 data variable
[47]  CTL←¯10,(ρCHTCTL),(ρDC),DC,(ρX),X,(ρ,Y),(,Y),3ρ0 ⍝ assign control var.
[48]  →(0 8≡1↑C←CTL)/0,L3
[49]  ⎕ES 'AP126 OR GDDM ERROR. RC=',⍕C   ⍝ unknown error
[50]  L3: ⍝ GDDM ERROR . . .
[51]  CTL←107 96                  ⍝ FSQERR
[52]  ⎕ES 88↑8↓DAT
∇
```