REORGANIZATION OF THE APL LIBRARIES

by

E. Armitage
J. Chenier
C. Leibovitz
D. Precht
Y. Zia

University of Alberta

ABSTRACT

The University of Alberta has decided to reorganize the APL library. The functions are being rewritten to conform to a new set of standards. New standards have also been developed for the documentation.

At the time of the start of the project, the APL program library was a kind of museum dedicated to the art of programming in APL language rather than a program library at the user's service. Its division into workspaces was not necessarily functional but reflected instead a partition according to the contribution of the different "schools" and "artists".

The library tended to illustrate what APL could do (and how elegantly and in how many different ways) and not how it could best help the user.

As a result, the library had no unity and the functions had no standards. The library contained a number of obsolete functions. (In one case, the author was kind enough to indicate in a comment line that the function was not reliable and was quite inefficient.)

Duplication could not be avoided. A function performing less well than one already in the library could not be rejected, each one was a different masterpiece. Both had to be kept in the spirit that the Mona Lisa does not exclude the Venus de Milo.

Freedom of art does not go with strict laws. The anarchy of the rules for the distribution of the parameters between the arguments and the global variables illustrated the versatility of the language but was not helpful to the user. A lot could be said also about the absence of rules concerning the amount of internal checking, programmed error messages, or fixing the allowed CPU time.

Likewise, the documentation of the program library was insufficiently standardized and it was apparent that better could be achieved in the light of accumulated experience in the use of APL.

The purpose of the reorganization of the APL library was
1) to reduce the amount of duplication and to discard 'junk' programs.
2) to establish new standards for function structures and documentation.
3) to improve the grouping of functions into workspaces and workspaces into libraries (according to functional classification).
4) to establish the areas in which development work is most needed and to start work accordingly.

Reducing the amount of duplication

This requires a review of the content of the existing library and an evaluation of the functions so that we can decide 'what we are going to keep'. The rule adopted was that of J.W. Tukey: IF A THING IS NOT WORTH DOING, IT IS NOT WORTH DOING WELL. Therefore, inefficient algorithms are to be rejected even if they are perfectly coded.

It is known, however, that the techniques for efficient coding in APL are different from those applying, say, to Fortran or Algol coding. In particular, it is important in APL to keep at a minimum the amount of interpretation. Coding should generally be concise and avoid as much as possible, repeated interpretation of a line through loops or iterations. What is less well known is that the practice of choosing the most efficient algorithm (in terms of the number of arithmetic operations) does not lead necessarily to the most efficient APL function (however well it may be coded). The overall efficiency of an APL function is measured by both the CPU time required by the algorithm itself and that required in its interpretation during execution. It may well be that a 'less' efficient algorithm may need less interpretation and have therefore a greater overall efficiency than that of a 'more' efficient one.

An APL function may have to be used a number of times with different values for the arguments. It is sometimes possible to code the function so that the different arguments are combined in one vector argument (or array argument) and in such a way that the code is interpreted only once for all the values of the argument. In such a case, the efficiency of the function increases dramatically and may equal or exceed that of a corresponding Fortran routine. In fact the interpretation time spent per element of the vector or array argument, is now divided by N (N being the number of elements of the vector argument). If N is large enough, the time spent on interpretation becomes relatively negligible. However, even with N=2, the interpretation time is already halved (relatively). This feature was exploited in coding some of the functions.

In evaluating the functions, we must consider the following:

Unless a function is well documented it may be difficult to discover its true purpose. A function may perform badly except in a few cases. The function is therefore a poor one unless it has been specifically written to handle those few cases. Another function may be poor from all points of view except that of core saving. It is only in the light of the purpose that a correct evaluation may be done. Once the purpose is given the evaluation job is better defined. If we do not know the purpose of the function, we must provide for an additional job effort to avoid the following possible mistakes.

a) We may mistake the function for a general purpose one and recommend its use without noticing a number of inefficiencies (due to modification of the coding or the algorithm for a special purpose).

b) We may mistake the function for a general purpose one and discard it because of an apparently poor performance without knowing that the routine had a more limited but important purpose.

The Algorithm

It is often possible without too much effort to recognize the algorithm on which a program is based. It is not easy to check if the algorithm has been scrupulously respected or if it has been modified. Moreover, if the algorithm has been modified we must find out if the performance has improved, or if the modification was unwarranted or if it had a special purpose (and which one).

The Coding

The coding is often undocumented and its logical structure is "mysterious". The how's and the why's of the coding are often very hard to establish. Even when the coding is clear it must be checked for all possible paths.

The Robustness

A good function must provide for correcting measures in all pathological cases. If for instance a divisor could be zero, we should check if it is before performing the division. If the divisor is zero the function should either deliver a message and stop, or branch to other computations according to the case. The robustness of a function can be checked "logically" by trying to determine the different pathological cases, by testing the program with various inputs and by comparing it with a similar program of known robustness.

The Documentation

The adequacy of the documentation can be checked by
1) reference to standards of documentation,
2) using the routine relying on the documentation,
3) examining it for pertinent information that may be useful either to the general user or to a beginner (either in computing or in the subject field).

Testing

Testing is a difficult art because it must be engineered
1) to follow the different logical paths of the program,
2) to find the validity domain of the program, and
3) to find out its performance in almost pathological cases.

Ideally, the evaluation should state the correct purpose of the function and find out how well the algorithm and the coding are fit for the purpose. It would include a study of its robustness, the result of a thorough series of tests, and finally an appreciation of the available documentation.

It became clear to us that in such a case it would take too much time to completely evaluate the APL library. We therefore decided that the evaluation would be made in stages.

As a first stage, for instance, the main algorithm would be recognized and dealt with independently of possible slight modifications. Functions would be compared (to pick the better) instead of being considered singly. Testing would be done in a limited sense to find the performance for the usual cases instead of the almost pathological ones.

Once the library is reorganized it will always be possible to improve the evaluation of each program.

STANDARDS FOR FUNCTION STRUCTURE

The standards for function structure were nonexistent. Suppose, for instance, that one is interested in a function computing the integral of f(x) in the interval [a,b]. An APL function giving an approximation to this mathematical expression will need extra parameters such as the precision required, the maximum number of iterations permitted, the starting number of intervals into which [a,b] is to be divided, etc. Those extra parameters are called control parameters here as opposed to the parameters in the original mathematical expression. We must now decide which of the parameters (control and regular) should constitute the arguments of the function, which should be global variables, which should have preassigned values. How should the user modify the preassigned values? Should there be programmed error messages? In what circumstances? What should the options be once an error message is delivered? Should there be a check on the amount of CPU time used, etc., etc.,..?

We have decided to implement the following standards in the use of parameters:

1) The regular parameters should become the arguments of the APL function. If there is more than one regular parameter and if there is a natural way to divide the parameters into two classes, the function will be dyadic. No strict rule has been devised to decide which of the parameters would constitute the left argument and which would constitute the right argument of the function. However, in case one of the parameters is the name of a function then it would constitute the left argument.

2) The control parameters should be global variables to which the function assigns default values. As a result, the use of the function becomes quite similar to the use of the mathematical expression approximated by the function.

3) Each function, say F, containing control parameters should have a global variable called DEFAULTF acting as a switch; whenever the value one is assigned to DEFAULTF all control parameters of F regain their default values.

4) Each workspace containing functions using control parameters will have a niladic function with no explicit result called DEFAULTALL that sets all switches (DEFAULT1, DEFAULT2, etc...) to one, where ALL stands for the first three characters of the workspace name.

5) Whenever possible, to avoid functions with no explicit result.

```
     ∇DEFAULTALL              ∇Z←...F1...               ∇Z←...F2...

[1]  DEFAULTF1←1         [1]  →4×⍳(DEFAULTF1≠1)    [1]  →4×⍳(DEFAULTF2≠1)

[2]  DEFAULTF2←1         [2]  TT1←.....            [2]  TT2←

[3]  'OK'                [3]  NN1←.....            [3]  NN2←.....

     ∇                   [4]  ...............      [4]  ...............

                         ...................       ...................

                              ∇                         ∇
```

F1 and F2 use two control parameters each named respectively TT1, HH1, and TT2, HH2. Lines 2 and 3 in the body of the functions F1 and F2 assign the default values. Line 1 in these two functions is a conditional branch to line 4 to bypass the assignment of default values in case DEFAULT1 or DEFAULT2 have been assigned a value other than one.

With this kind of arrangement the user has the following options:

a) He may assign values to the control parameters and these will be effective provided he assigns also a value other than one to the switch DEFAULTF corresponding to the function F. He may then save his workspace.

b) As long as the user does not modify the value of the switches and that of the control variables, any reloading of the workspace will have the functions ready to be executed with the previously assigned values of the control variables.

c) As long as the user does not reassign the value one to a switch, the user may assign new values to the control variables.
d) Whenever the user would like to be sure that the default values are used, he only needs to type DEFAULTALL and carriage return without having to reassign the default values.

6) Whenever a function could take more than, say, 10 sec. for its execution (depending on the value of the regular and control parameters), an additional parameter should be used to determine the maximum CPU time allowed. The function should be written in such a way that it becomes interactive when the allocated amount of time has been used. The function should be able to display whatever information is needed to decide if it is better to interrupt execution or if it is advisable to allow for an additional amount of CPU time.

7) Our implementation of the convention for control variables is such that when the default values are used, the previous values of the global variables are not modified.

In the following example of the structure of the function, the control variables CP1 and CP2 are local. They have corresponding global variables CP1 and CP2.

If the option chosen is the default one, only the local variables CP1 and CP2 are assigned default values. If the option is for user-defined values, the local variables CP1 and CP2 are assigned the values of the global variables CP1, CP2 defined by the user.

```
    ∇Z←...FUNCTION...;CP1;CP2;...
[1]  I21←⌶21
[2]  S∆FUNCTION←STOP
[3]  →OPT×⍳(DEFAULTFUNCTION≠1)
[4]  CP1←(ASSIGNING DEFAULT VALUES TO CONTROL PARAMETERS)
[5]  CP2←...
[6]  →START
.....................................
[ ] OPT:CP1←CP1
[ ]  CP2←CP2
.....................................
[ ] START:START OF COMPUTATIONS
.....................................
[ ] LOOP:→MESSAGE×⍳CPU<.0166666666667×(⌶21)-I21
[ ] GO:START OF LOOP COMPUTATION
.....................................
[ ]  →LOOP×⍳........(END OF THE LOOP)
.....................................
[ ]  END OF COMPUTATION
[ ]  →0
[ ] MESSAGE:(MESSAGE WITH OPTIONS:*GO OR →STOP)
[ ] STOP:Z←(PARTIAL RESULTS)
    ∇
```

8) If a function requires specific arguments, it is often desirable to check them and issue appropriate error reports before execution. Whenever possible and advisable to do so, checking should be done.

The checking of input parameters has been separated in a modular way from the algorithmic part. Therefore, there will be two functions instead of one: the first will not check the input, the second will first check the input and then will call the first one. This will allow more flexibility and will tend to satisfy different kinds of users. In this case, naming conventions will indicate functions by FUNCTION and FUNCTIONCK, respectively. The error message delivered should indicate the name of the function issuing the error report.

DOCUMENTATION STANDARDS

It was recognized from the beginning that the goals of the project must, for practical purposes, be set in two steps:

1) to make available a collection of useful functions and documentation formed according to our newly developed standards

2) to develop a collection of workspaces that would encompass all the university disciplines and in which no function would be included except after an evaluation that would show that it represents the APL state of the art (in algorithm and in coding).

The amount of material and the variety of subject areas processed, developed, evaluated and/or produced is of sufficient magnitude to require the work of a group of persons. The development is to take place in gradual steps from setting down details for the formulation of the project, design and evaluation of initially non-interactive functions to, finally, the production of packages built to be more interactive for the users who are not computer oriented.

Each of the functions is to be placed in a library with a number corresponding to the first two digits of the SHARE classification code. This scheme was already in use at the University of Alberta for its general program library. Its use by the APL program library allows these two collections to be placed together in a reference catalog of programs.

The documentation for the APL public library functions is to be made available to users in both hard copy and on-line forms. Each subject area will have a separate documentation manual. In exceptional cases, several subject areas may be included in one manual.

For each given library number, the documentation has been placed in a workspace called LIBDOC. The documentation was therefore kept in a workspace entirely sepatate from the functional workspaces. Since printed manuals of the documentation would be readily available for the user, the on-line documentation was maintained as a quick and convenient reference for the user as well as a source of the most current documentation. These workspaces were also designed for ease in use of the APL print train and in tape preparation for transporting to other installations. Placing the documentation in a separate workspace also makes additional space available in the functional workspace.

The workspace LIBDOC contains all the documentation for a given library number. LIBDOC provides an easy and consistent access to the documentation for the structural units of any particular workspaces. These units are hierarchical to enable the user to access a generalized or detailed description of the object desired. The variable LISTWS lists the available workspaces in that library and briefly describes their contents.

Each workspace in a given library is described in a variable of the same name in LIBDOC. This variable either describes the contents of the workspace or gives a brief explanation of the purpose of that work-space with a list of variables which give more details.

Each stand-alone or driving function in a workspace is documented in a variable named functionHOW. The user need know only the name of the driving function to obtain all subfunctions needed by the main function. These can be obtained by copying the group named by the function name suffixed by GP. Groups of related functions will be called by a mnemonic name.

Examples were considered a worthwhile addition to the on-line documentation to allow to find quickly an illustration of input, execution and output. The workspace containing these examples was named XMPLS and each such example called functionXMPL.

The format for documentation variables LISTWS, wsname, functionHOW, and functionXMPL has been explicitly outlined in the attached form no.-1.2b. These variables include details enabling the user to locate the object, determine its syntax purpose and descripton, and to allow him to contact the author or a consultant on further information if so desired.

A limited numerical analysis library as defined in the first stage of development has now been completed. The manual has been produced. A statistical library is presently in the development stage.

The next steps taken by the project will be advertised as soon as decisions are made with respect to them.

APLP
Form No.-1.2b
9/9/75

<u>Documentation Units and Standards of Documentation</u>

The present documentation scheme includes the following structural units:

1. "LIBDOC" Each public library has a workspace called LIBDOC which contains all the stored documentation for that library.

In each LIBDOC there is a variable called

2. "LISTWS" which lists the available workspaces in that library and briefly explains their contents.

3. "WSNAME" For each workspace in a given library, there is a variable of the same name in LIBDOC. This variable either describes the entire workspace if the workspace has a

single purpose, or gives a brief explanation of the purposes of the workspace and a list of variables which give more detail.

4.	"FunctionHOW"	variables will be used to document stand alone or driving functions.
5.	"NameGPHOW"	variables will be used to document a group of related functions referred to by the label "NAME".
6.	"XMPLS"	Each public library will have a workspace called "XMPLS" which will contain examples of input, execution and output of stand alone or driving functions in that library.
7.	"FunctionXMPL"	variables will contain an example of input, execution and output of the indicated function. These variables will be placed in the workspace "XMPLS".

Format for the documentation units within the LIBDOC workspace will be:

1. "LISTWS"

```
Lib. no. LIBDOC LISTWS
                        A Brief description of the purpose of this library.
WORKSPACE    NATURE OF CONTENTS

.                .........
.
.
```

2. "WSNAME"

```
Lib. no. LIBDOC WSNAME
LOCATION:  Lib. no. WSNAME
                         A brief description of the purpose of the workspace followed by a list of functions
                         in the workspace together with documentation for each or a reference to further
                         documentation of the function in FunctionHOW or NameGPHOW.
```

3. "Function HOW" or "NameGPHOW"

```
Lib. no. LIBDOC FunctionHOW (or NameGPHOW)
LOCATION:  Lib. no. Wsname Function (or NameGP)
SYNTAX:
PURPOSE:
FunctionGP or NameGP MEMBERS: (If any)
ARGUMENTS:  (If any)
DESCRIPTION:
RESTRICTIONS:
ORIGIN:
DATE REV:
CONTACT:
```

Format for the sample program run located in the XMPLS workspace will be:

"FunctionXMPL"

```
Lib. no. XMPLS FunctionXMPL
FUNCTION LOCATION:  Lib. no. WSNAME Function (or Groupname, if any)
DOCUMENTATION LOCATION:  Lib. no. LIBDOC FunctionHOW
SAMPLE PROBLEM:
AUXILIARY FUNCTION DEFINITION:  (If any)
INPUT:
OUTPUT:
```