7103

Int. Stat. Rev., Vol. 42, No. 3, 1974, pp. 291–298/Longman Group Ltd/Printed in Great Britain

A User's Guide to the Evaluation of Statistical Packages and Systems [1]

J. A. Nelder
Rothamsted Experimental Station, Harpenden, Herts, England

Introduction

There is widespread concern today that the general availability of statistical packages and systems (Schucany, Murton and Shannon [6]) may bring dangers as well as benefits. Obvious dangers are (i) that the programs may not do what they purport to do, and so supply the unsuspecting user with quite misleading results, (ii) that the procedures they contain may be grossly misused, being applied to unsuitable data, again with the user being unaware of the situation, and (iii) that systems may force the user into certain modes of analysis because these are what the system provides rather than what the user really needs.

Terms like "assessment" and "validation" are usually taken to refer to the logical question "does this program do what it purports to do and how can I prove this true or false?" There is, however, a much wider problem of assessment, namely that facing a scientist when, armed on the one hand with his data and theories to be explored, and on the other with information on programs from his computer centre, he has to match what is on offer against his needs.

Because the analysis of data is, or ought to be, an open-ended process, no formal theory of this kind of assessment is possible. None the less, I think that users can be given some useful information that will help them to assess existing programs before they commit themselves to what may become an extensive period of use involving much of their own time and perhaps much computer time also. The diversity of form and scope among programs makes this assessment a formidable task. In this paper I discuss some of what seem to me the more important considerations. If a potential user has a thorough grasp of these he will be equipped at least to ask awkward questions of the right people and to demand answers.

1. How does it run?

There are several ways in which a program can be brought to an executable state. Three important ones are considered below with their pros and cons.

1.1. *Program with an Own-language Driver*

These consist of a set of subprograms for which the user must supply a main program and perhaps other subprograms. These must be compiled, linked to the standard package and then run. An example is provided by Yates' Rothamsted General Survey Program Part 1 [9], where the input routines and the derived variate operations are provided by the user in Fortran.

The main advantage of this technique is that the user has access to the full power of a general-purpose programming language (GPL) (usually the same one in which the package was written)

[1] Substance of a talk given at a meeting entitled "The assessment and validation of scientific software", held on 19 December 1973 at Imperial College, London, and organized jointly by the Royal Statistical Society and the British Computer Society.

and this gives him greater flexibility than the usually more restricted problem-oriented language (POL) supported by the package itself (see 1.3). Thus in RGSP the user can specify complex input structures for his data by writing the appropriate Fortran instructions.

The main disadvantage of this technique is that it requires the user to master appreciable parts of the system *at the level of the language they are written in*. How difficult this is will depend critically on the documentation, a matter we consider in more detail later, and on the user's skill with the language in question. This skill also determines how long it will take to get the driving program to work.

1.2. *Programs with Translators*

In this scheme the program presents the user with a problem-oriented language intended to simplify the specifying of a class of problems. The IBM Continuous Systems Modeling Program (CSMP) [3] is an example for simulation. Having written his instructions in this language, the user presents them to a translator (which is part of the system concerned), which in turn translates the program into a language for which the machine has a compiler. The translated program is then compiled, linked with the rest of the system and run.

This scheme has the advantage that problem specification uses a problem-oriented language which should relieve the user of many programming difficulties. It also has the advantage that the translated program is compiled into machine code, and so should run faster than a program using an interpretive language.

The corresponding disadvantages are that the problem-oriented language may lack some of the generality of the underlying language, thus imposing constraints on the problems that can be specified. (Possibly this may be overcome by including new subprograms in the underlying language, in which case the situation reduces to the first type.) On some machines the fact that four steps are involved in each run (translate, compile, link, run) can lead to appreciable system overheads, but this is a variable characteristic.

1.3. *Programs with Interpretive Languages*

These programs present the user with an interpretive language in which he writes his instructions. The program exists in loadable form in the machine, and when called will interpret the instructions and execute them. ASCOP (Cooper [1]), GENSTAT (Nelder [4]), and SPSS (Nie, Bent and Hull [5]) are examples of such systems in statistics. In all these the underlying Fortran program analyses and interprets the user's instructions issuing calls to appropriate Fortran routines to execute them.

Advantages of this scheme include the problem-oriented language, and having only a single job stop at each run as against four for the second type. The disadvantages are the restrictions in the interpretive language and the fact that interpreting is always slower in execution than compiling, and may sometimes be much slower. Again there are ways of modifying the basic process; for example, Genstat has a directive OWN which allows the user to add a new facility to the system with an arbitrary set of parameters. However, to do this the user must know how to write an interpreter for that system, which implies detailed knowledge of its internal data structures, and he must also link his new part to the rest of the system before running it, so that the disadvantages of the translator system then appear.

1.4. *Do-it-yourself*

A fourth option always open to the user is to do it himself, an option that some would say is only too often used, with the resultant proliferation of many poorly written, poorly tested

programs. Of course, you can do it yourself at various levels – often there are good published algorithms for critical operations. Too often these are not used – the idea of a literature search for programs has made little progress.

1.5. *Extensibility*

Whatever type of system is chosen its capacity for being extended will be critical for any but the most routine use. The potential user has thus not only to assess whether his needs are met by the existing system, but also if not, whether the system is extensible in the direction required.

2. Characteristics of Problem-oriented Languages

Given that a POL exists for your kind of problem, usually as an interpretive language of a particular system, the question then arises whether it is flexible enough for your particular needs. The scope of POLs varies enormously and it is essential that the user understand certain fundamental characteristics of languages in order to make the required assessment.

2.1. *Branching*

Branching instructions are fundamental to GPLs. It is absolutely necessary that the programmer be allowed to branch on conditions and so break up the sequence of execution in arbitrary ways. However, branching is not a universal property of POLs. Some, such as SPSS, allow only a single sequence of operations. (It is true that some directives, e.g. for stepwise regression, may contain implicit branching *within* them, but none is allowed between directives.) The advantage to the implementor of a language without branching is that he does not have to save the instructions internally. They can be executed as they are met and then discarded. A branched language must allow instructions to be saved internally in a coded form ("compiled", though not of course into machine code if the language is interpretive) so that branching can occur.

There is an important interaction between the type of language, i.e. whether compiled or not, and the mode of running, i.e. whether batch or interactive. An interactive program allows for real-time branching by its very mode of action. The user can look at the output and decide which way to go next. Formal branching facilities are thus much less important than with batch running.

For any but the simplest type of analysis batch-type POLs without branching are severely limiting to the user. Why then are they so widely accepted? It may be that most analyses are very simple-minded so that conditions do not appear in their specification. There are also many users of packages, too many I would say, who don't want to think quantitatively about their data, and who regard the program as a black box capable of delivering *the* analysis which will make their data look thoroughly respectable to referees and editors, far too many of whom are taken in by this ploy.

A user who has come to regard analysis as a multi-stage process will soon find batch systems having languages without branching cripplingly restrictive.

2.2. *Data Structures*

A second important characteristic of POLs is the class of data structures they support. What kind of data can you declare, read in, operate on, store, retrieve, and print out? Most statistical systems recognize the *data matrix* as being of basic importance. This is a two-dimensional structure indexed by *units* and *variates* and many kinds of experimental and survey data can be cast in this form, particularly if qualitative variates (factors) are accepted. These serve to define mutually exclusive subsets of the units. All builders of statistical systems face a dilemma at this point – whether to take the row (one unit) or the column (one variate) as the basic internal structure. The advantage of using a unit is seen best with survey data having many units needing basic tabulations. The tables can be updated one unit at a time, so that the number of units is unimportant and space is needed for only one at any time. While the addition of more units causes no trouble, the addition of more variates does, because extra cells have to be created scattered along the data, or a new data matrix begun which may need internal accessing of each record in order to insert one item. Such new variates can be created by fitting a model to the data and saving fitted values and perhaps residuals. It is for this reason that programs concentrating on analysis of experimental data have tended to take the variate as the basic unit, and because the number of units in this situation is often not large, storage problems have not been severe. The variate is easier to manage internally because all its values are of the same type, whereas those of a unit need not be.

There is no perfect solution to the data-matrix dilemma. In Genstat we have taken the variate as the basic unit to give the flexibility to create new variates easily, but we have allowed for the many-units situation by allowing certain directives, READ, TABULATE, and SSP to work sequentially on blocks of units, the data being lost from each block after processing. (They can of course be saved, if required, on backing store.) While blocks of 1 unit would be grossly inefficient with an interpretive language, blocks of 10 give satisfactory speed without using large amounts of store.

At least as important as the data matrix are the other structures which the system supports. As analysis proceeds, derived structures are produced from the data matrix and become the input to the next stage. An example is the SSP matrix (which numerical analysts now frown on). The user must ask – what does the system allow me to do with it? A good starting question is – can I name it? If you can't, as in SPSS, then you will not be allowed to have more than one active at any time, so that you cannot, e.g. add two SSP matrices together. If you can name it, as in Genstat, then you can have several, and it is likely that more scope exists for further operations.

Another basic structure is the *multi-way table*. All survey programs must be able to form and print multi-way tables derived from data matrices. However, some stop at this point and do not allow operations on tables considered as basic data structures. Again the naming facility is crucial and its absence will be a sign that the originators did not think of these structures as being *input* structures to an analytic procedure.

An analogous situation arises with character strings, or text. If these can be named, then it takes very little extra coding to allow these to be inserted in the input stream of instructions at a suitable place. This gives the user a macro-facility. Furthermore, if such text structures can be stored and retrieved along with other structures, then macro libraries can be created and referenced just as easily as sets of data. Such a facility is particularly valuable to the statistician who wishes to create standard procedures for his clients for each subsequent use by them. In a properly designed system, it takes remarkably little extra code to provide what can be a powerful tool.

Action that is taken by implementors on naming and manipulating data structures is closely bound up with assumptions about the analytic process itself, and these need bringing into the open.

3. Multi-stage *vs*. Single-stage Analysis

Almost everyone accepts the idea that the analysis of a substantial body of data must be a multi-stage process of trial and error. Our models can only be tentative, may be contradicted by the data themselves, and will usually need revision. But although everyone may accept this

principle, far too many people accept computer programs that, far from encouraging a multi-stage approach, actively foster the single-stage procedure

```
input
analyse
output
stop.
```

Those who do not wish to think may welcome the magic black box that tabulates their data in all possible ways, does all possible regressions, calculates all possible statistics, prints the results on several kilometres of paper and then stops. The intelligent user will reject this approach, but he needs to recognize symptoms that a program is based implicitly on a single-stage view of analysis. The critical question is – does the system allow the output from a procedure to be named as instances of suitable data structures, and used subsequently as input to other procedures? If the answer is "no" then, real multi-stage analysis is going to be at best difficult and at worst impossible. Thus facilities for naming output and the definition of a common set of structures for both input and output are key features of a system supporting multi-stage analysis. A further essential feature is of course the ability to save, and subsequently to retrieve, not just the raw data but data structures derived from them.

Example. Suppose a table of counts has been created from a data matrix and it is now required to fit a log linear model to the table and calculate the discrepancy from the data. The tabulation procedure must first allow the output to be saved as a multi-way table. Then this table must be able to be used, either directly or indirectly as the input to a program for fitting log-linear models. If the system allows weighted regression with the naming of vectors of fitted-values and weights, and also iteration by a branching facility in the language, then the log-linear fitting can be programmed by the user directly. If it does not, then the user must either persuade the originators to extend the system, or do it himself. It is an open question which will take longer.

4. Characteristics of Algorithms

The algorithms of a system pose two problems for the user: (1) do they do what he wants to do? and (2) do they work properly?

4.1. *What do they do?*

There are many techniques of analysis and no attempt can be made to specify all the desirable features of each. Some general points of guidance can, however, be given. It is important to discover how missing values are treated. Are they, in the first place, allowed at all in a data matrix? May they occur in derived variate calculations? What happens when an SSP matrix is formed from a data matrix containing missing values? Can missing values be easily replaced by some other values?

In regression analysis the potential user should find out how aliasing (particularly partial aliasing) of effects is handled. Degeneracies should lead merely to comment, not to halting of the calculations. If they do, then it is virtually certain that no decent facilities have been provided for including factors (leading to terms like a_i in the model) as well as variates (leading to terms like bx). Be suspicious if automatic procedures (like stepwise) regression are provided without the steps of which they are made being separately available. Without these you will not be allowed to think for yourself.

It is important that algorithms should be able to work on subsets of units as well as on all of them. This facility is strictly one of data-access rather than of algorithmic specification, but the two aspects are often not clearly separated.

4.2. *Do they work?*

A now well-known paper by Wampler [7] reported putting some admittedly nasty regression problems with nearly singular matrices to several packages with generally dismal results. Many of the algorithms used were not numerically stable, and though they produced results, these might contain no correct significant digits at all. Thanks to the work of Wilkinson, Golub and others there is now no excuse for not using stable algorithms for linear operations and for extracting latent roots and vectors. None the less, time spent on submitting searching test data to a program may not be wasted, and a healthy pessimism is always justified. For example, Francis [2] has described the output from four programs for the analysis of variance of data from a two-way table with unequal numbers in the subclasses. One program gave the wrong answer because it treated the data as orthogonal, two more gave incomplete answers and did not explain the incompleteness to the user, and the fourth one gave some incorrect sum of squares through failure to recognize the marginality relations between the main effects and the interaction term. (The results from this last program were not recognized by the author as incorrect.) Thus a data structure for which the form of the analysis of variance has been known for forty years (Yates [8]), is still being incorrectly treated in widely used programs. Let the user beware.

5. Space

Space is required for both program and data, in core and in backing store. The user must understand something of the constraints imposed both by the operating system and the program itself. He should know what partition size is required and what the consequent priority for his job is likely to be. If the program is overlaid, then a complex run may involve many overlay changes; it may be possible ro reduce the number considerably by grouping together the instructions that use the same segments.

For data the user needs to know how much space is available, both in core and on backing store, whether there are restrictions on individual dimensions of structures and lists, and if space is recoverable (by garbage collection) during a run. Some space restrictions may be very difficult to circumvent, and one must be absolutely clear that the data and the problem are going to fit if frustration is to be avoided.

6. Documentation

To convert a problem into a form where a program can be written, or a system used for its solution is a complex process. What is needed is an intermediary. There are two possibilities for most users; one is for the user himself to be the intermediary, and the other is for the local expert to take on the job. The first question the user must ask is – does the local expert exist? Whether the local expert exists or not depends very much on the policy of the computer centre the user has to deal with, for centres differ widely in the extent to which they think the user ought to be supported in this way. Some give all assistance short of actual help.

If the local expert does not exist, the user must do it himself, and face the translation problems involved. He must master the program's jargon, and match its concepts with his own. Most importantly he must establish whether, despite differences in terminology, he can actually do what he wants. Some standardization of words used to describe data structures would be of enormous help here. The user is totally dependent at this point on the documentation provided with the program. It is very difficult to provide good documentation, and still far from clear how to do it. For Genstat we have arrived at the following state of affairs: to use the system at the *external level*, i.e. without knowing its internal structure the user needs

(i) a prospectus;
(ii) user's guides;

(iii) a reference manual;
(iv) worked examples;
(v) a notice board.

The prospectus is a document a few pages long, which outlines the scope of the system, the data structures it supports and the algorithms it provides, together with the machines on which it is available, and the partition size required to run it. From the prospectus the user can form a first opinion on whether to pursue the idea of using the system further.

The user's guides give introductions to various parts of the system. They are written quite informally and do not try to cover all the options; they try to lead the user into the system by easy stages. These guides do not discuss the statistical background to the procedures, but assume that the user understands why he is proposing to use a particular technique on his data.

The reference manual is written formally, beginning with the syntax from character set to program, followed by all the directives in the system, with concise descriptions of their options, etc. Appendices summarize the syntax, the diagnostics, machine-dependent features and a general index. It can be read as a book, but generally will serve as a work of reference.

The worked examples are supplied as a source file held on disc or tape and allow the user to obtain copies as required; they are internally annotated with comments and can be read as independent documents.

The notice board is another source file containing known restrictions and faults in the system classified by directives. It ought to be null but never is.

To use the system at the *internal level*, i.e. to modify the program at the Fortran level, the user requires documents covering

(i) the internal representation of the data structures and "object" code;
(ii) the system's accessing functions;
(iii) descriptions of common blocks and subprograms;
(iv) the linking structure.

To undertake the writing of a new interpreter would presuppose a user of considerable sophistication, who would usually need the assistance of the originators.

Conclusion

The hurdles facing the potential user who wants to use a computer to analyse his data can easily be made to seem so many and so huge that one may wonder how anybody gets anything done at all. Although things do get done yet the amount of frustration is still, I believe, very large. Existing general-purpose programming languages are not adequate for many users. POLs are better but there is no guarantee that the class of problems envisaged by the originator include the user's current problem, and it is often too difficult to find out whether it is so or not. Difficulties arise from inadequate standardization of terms, inadequate documentation and inadequate program support. It is probably no accident that the activities that would remedy these inadequacies do not carry the kudos in the academic world that they should.

References

[1] Cooper, B. E. (1972). *ASCOP: A Statistical Computing Procedure* (*Version* 3). Science Research Council.
[2] Francis, I. (1973). Comparison of several analysis of variance programs. *Journal of the American Statistical Association*, **68**, 549–565.
[3] International Business Machines. (1969). *System/360 Continuous Systems Modeling Program* (*360A–CX–16X*) *User's Manual*. IBM.
[4] Nelder, J. A. and Members of the Rothamsted Statistics Department. (1973). Genstat Reference Manual. *Inter-University/Research Councils Series*, *Report No. 3*, *Second Edition*. Edinburgh: Program Library Unit, Edinburgh Regional Computing Centre.

[5] Nie, N. H., Bent, D. H., Hull, C. H. (1970). *SPSS: Statistical package for the Social Sciences*. New York: McGraw-Hill.
[6] Schucany, W. R., Murton, P. D., Shannon, B.S. (1972). A survey of statistical packages. *Computing Surveys*, **4**, 65–79.
[7] Wampler, R. H. (1970). On the accuracy of least squares computer programs. *Journal of the American Statistical Association*, **65**, 549–565.
[8] Yates, F. (1934). The analysis of multiple classifications with unequal numbers in the different classes. *Journal of the American Statistical Association*, **29**, 51–66.
[9] Yates, F. The Rothamsted General Survey Program (Parts 1 and 2). *Inter-University/Research Councils Series, Reports Nos. 2 and 14*. Edinburgh: Program Library Unit, Edinburgh Regional Computing Centre.

7104

Barea, J. M. & Brown, Margaret E. *J. appl. Bact.* **37**, 583–593.

Effects on Plant Growth Produced by *Azotobacter paspali* Related to Synthesis of Plant Growth Regulating Substances

J. M. Barea* and Margaret E. Brown

Soil Microbiology Department, Rothamsted Experimental Station, Harpenden, Hertfordshire, England

(*Received 27 February 1974 and accepted 20 May 1974*)

Summary. Treating seedling roots of several plant species with cultures of *Azotobacter paspali* changed plant growth and development and significantly increased weight of leaves and roots; effects were probably caused by plant growth regulators. Culture supernatant fluids contained indolyl-3-acetic acid, at least 3 gibberellins and 2 cytokinins. The added inoculum of *A. paspali* survived in plant rhizospheres for only a few weeks and no nitrogen was fixed in the root zone of young *Paspalum notatum*, the grass with which *A. paspali* is associated.

The nitrogen fixing bacterium *Azotobacter paspali* was found by Döbereiner (1966) in soils with a pH range of 4·9 to 7·8 and to occur more abundantly in rhizospheres of *Paspalum notatum* and *P. plicatum* than in soil away from roots, but it was absent from root samples of 81 other pasture plants. *Azotobacter paspali* improved growth of *P. notatum* by fixing atmospheric nitrogen in the rhizosphere (Döbereiner, Day & Dart, 1972), an ability not known to be shared by other species of *Azotobacter* when associated with roots of any plant species. However, other species of *Azotobacter* alter plant growth by producing the growth regulators indolyl-3-acetic acid and gibberellins (Brown & Burlingham, 1968; Lee, Breckenridge & Knowles, 1970; Azcón & Barea, 1973). This paper presents the results of examining culture supernatant fractions of *A. paspali* for these substances and for cytokinins, and the effects on plant growth following root treatment with bacterial cultures. Studies on cytokinin production were made because recently these substances were found in extracts of cells of *A. chroococcum* (Coppola, Percuoco, Zoina & Picci, 1971), of *Agrobacterium tumefaciens* (Romanow, Chalvignac & Pochon, 1969), and of *Corynebacterium fasciens* (Helgeson & Leonard, 1966; Klämbt, Thies & Skoog, 1966), and in RNA preparations from a wide range of micro-organisms (Skoog & Armstrong, 1970). Cytokinins were also found in culture supernatant fractions of *Arthrobacter* sp. (Blondeau, 1970) and *Rhizobium japonicum* (Phillips & Torrey, 1970).

Methods

Cultures

Cultures of *Azotobacter paspali* were grown for 14 days (Brown & Burlingham, 1968) in 100 ml amounts of medium containing (g/l); sucrose, 5·0; K_2HPO_4, 0·8;

* Present address: Estacion Experimental del Zaidin, Seccion de Microbiologia, Avenida Cervantes, Granada, Spain.