ACID—A user-oriented system of statistical programs

by R. A. BAKER and T. A. JONES

Esso Production Research Company
Houston, Texas

A common problem in any scientific or business effort is to organize large masses of data for human interpretation. Today we will describe a program system which is designed to do just this; we call it ACID, which is an acronym for Automated Classification and Interpretation of Data. Our system may not be unique, but we believe it is well-designed. The user can get his work done with this system, which is set up as an "executive design"—one program controls a lot of other programs.

The executive, or interface, program puts a friendlier face on the system for the user. It works with him, not against him, which for many users is a unique experience. Too often we find a puzzled and bedraggled user on the point of either rage or defeat in his contest with Program X, which has stifled every effort on his part to communicate what is to be done. Running the cantankerous program becomes his major project, and the objectives of the real project get lost in the effort. This is less likely with ACID because the user gets in and out quickly. The major problems fall to making the analyses, not running the program. This is particularly a problem if the user must run several programs in succession.

ACID is focused on a somewhat heuristic pattern recognition problem (in the general sense of that term) although the concepts discussed here are certainly not limited to this. In the pattern recognition studies, the computer organizes the information, and the user tries to recognize and interpret the organization. ACID may be instructed to provide several alternative organizations, perhaps with different interpretations. The heart of the classification scheme we use is cluster analysis. This technique is as old as large-scale computer processing, but is none the worse for wear, as we have had good success with it. One of the secrets of success is having turnarounds within the time frame of the project and the user's patience; this includes pre-processing of data as well as the actual analysis. The user will not experiment with his computerized tool if it's all he can do to get it to work once or twice. If he does not experiment, he will not be likely to get the results he is looking for. So far ACID has kept well within the patience and project times of its users.

As an example of the kind of project ACID is designed for, we studied Galveston Bay, a major estuarine system in the vicinity of Houston. We have 85 sampling stations; approximately 50 species of mollusks were observed and counted at the stations. Our objective was to find a pattern among these samples by use of the contents of the biological collections at the stations. One might think that it would be fairly easy to quickly and efficiently solve such a problem. One need only go to the published literature for programs that will do the job. After all, cluster analysis is some 15 years old, and there are at least 20 programs available for doing the job. That was our original thought and course of action.

We obtained a program and ran our data with it. And ran it. And ran it. We soon found the practice to be nowhere near as simple as the concept. First our data arrays wouldn't fit the program dimensions. So we trimmed some data vectors. Then we couldn't transpose the matrix for an alternate analysis. Then we found the data formats were not usable with other programs, creating added complications in the computation of simple statistical summaries, making data transformations, and in the use of different cluster and multivariate analyses.

After great tedium we managed to perform the desired cluster analyses, and obtained four sample groupings. The procedure divided the bay complex into four areas: ship channel and tidal pass, forebay, backbay, and brackish bay-margins. This concluded the classification phase. Next, in the interpretative phase, we wished to determine the species that defined each of the groupings. This required the use of other programs, thus repeating the above problems with formats, dimensions, etc. Again, after much effort unrelated to the interpretation, we managed to find that only 6 species were needed in generating the pattern in the bay complex.

We found the actual analysis to take less time than the preparatory work of getting the data ready. And we're talking about data manipulation, not data reduction. This made cluster analysis a tool of questionable value. Was it worth months of effort, especially when much of this time would be spent in re-coding and keypunching and so forth in getting ready to run the analytical program? We decided "no," and that's why ACID was developed. Over a period of a year-and-a-half we wrote some 25 programs—large and small—and toward the end of that period combined them into the executive framework of ACID. Most of the programs in ACID are actually auxiliary to the cluster analysis step.
ACID is essentially a sub-operating system. It does for the applications user what the operating system does for the programmer. It keeps track of those essential but nuisance items such as the location of programs and data sets within the system. The user may have three or four data sets each at various levels of transformation in the system. If he wants to operate on a particular data set, he prefers to call it up by a code name, not by location, because the physical location of the data set on tape or disc is of no interest to the user. The system also provides useful services during operation, such as error checking. For example, within the limits of Fortran, ACID attempts to tell the user what happened before going off the air with an abnormal end. It also checks input parameters and data sets for certain types of errors. In addition, if the user is requesting more storage than is available, ACID tells him how much more is needed. It also dynamically dimensions arrays, so as to allow virtually any size data set to be processed. As with the operating system, most of what ACID does as a system is invisible to the user. He sees only the functional programs that were stand-alone before being incorporated in ACID.

The executive design gives us the flexibility of a library and the convenience of a single program. The basic system is set up with an executive that can call any of the individual programs. Figure 1 shows the structure whereby each of the operational programs is independent of the others and at the same level. Each is interfaced to the executive by a small driver subroutine. The user inputs a control card with the name of the desired option on it (e.g., STATISTICAL SUMMARY). At this point the executive calls the appropriate program. This program reads any necessary controls and performs the analysis. Control remains in this program until a special control card instructs a return to the executive. The various options are processed sequentially.

The control language used is basically hierarchical in nature (Figure 2). Some commands go to the executive program and some to the functional programs, and in general the commands are kept on separate cards and follow a sequence from highest to lowest level in the structure. For instance, we might wish to perform transformations on some of the variables in our data set, followed by the computation of some simple statistical summaries. The TRANSFORMS card instructs the executive to call the appropriate program. The program control then is used to define the data set required, and to set other controls. Following in the hierarchy comes the controls for each of the individually desired transformations. The controls for a second data set may follow. The end-of-program card returns control to the executive; by this time, two new data sets, 1A and 2A, have been created. The STATISTICAL SUMMARY card calls for the next program option to be executed. This sequence is similar to that of the computer operating system, and we have found it to be effective.

The ACID data sets must be easily accessed by the user. In ACID most of the data are in the form of rectangular arrays, with samples arranged along one side and variables across the top; the body of the array consists of the observed values. The data sets are stored as arrays in binary form on a sequential file. Tapes are normally used if the data sets are to be retained permanently. In addition to the data, there is header information for each data set. This information includes a code name assigned to the data, the dimensions of the data array, and code names to identify each sample and variable. These names are invaluable when the user is interpreting his results, and are also conveniently carried with the data in this form.

Several data sets may be placed on the same file, and new data sets are normally created in the editing steps. The user assigns a four-character code name to each data set as he instructs the program in its operation. The user thus inputs the name of a data set to operate on and a name for the resulting data set. The program finds the first data set on the file and inputs it; it then operates on the data and forms a new data set. This new set is placed on the data file under its own name; the original data set is also left unchanged on the file.
One of the original difficulties we found in running several independent programs was that we invariably had data sets with dimensions that exceeded those of the programs. The ACID system solves this by using a form of Dynamic Core Allocation. In this case, no arrays are defined directly in the programs. Whenever a data set is input, the number of rows and columns in the data set are used to compute the actual dimensions required for the program to process that data set. Then a routine is called that allocates the required number of words of core and defines the arrays. This core is released after the data set has been analyzed.

In addition, the executive structure allows us to modify and add programs with little disturbance of the program system. The functional programs are essentially stand-alone applications programs. When they are put into ACID, the main change is in their input sections. In the system they read data sets formatted and maintained by ACID on a sequential file, rather than reading decks from the card reader. In addition, a short driver routine is used to allocate the array space dynamically. Thus, we have the ability to develop functional programs independent of the system or to obtain programs from the published literature or elsewhere and add these programs to the system.

As part of the internal structure of ACID we have consciously kept the I/O as high up in the subroutine stacking as practical. This makes it much easier to come into a program and make a patch because the route from input to output is shorter and more direct. It also makes any data-editing more apparent, and the editing can be specific to the input data. Whether making a modification to the program or tracing a bug, a direct route from input to output makes the task much simpler. On the other hand, we found it impractical to concentrate all of the I/O in one program, such as the executive, because that reduced our ability to treat the programs in modular fashion. Program modification is naturally much simpler if the modification in no way affects another program.

Along this line, we might mention a "good" idea that didn't work out. An unsuccessful feature, since abandoned, was to concentrate all the output, error messages in particular, into one program. The concept was to store several error codes and then call a program at the end of the program that would interpret the codes and print the messages. Changing a message would be a simple matter of changing the error-printing routine. The functional programs would not have to be changed at all. The idea is not new with us, and we expect that it might work in some environments. We found it to be a poor idea for ACID. In the first place, we found that the error message format statements were very useful in supplementing our internal documentation. Secondly, we found that considerable library effort would be required to maintain an error printing program since every new option in ACID would probably require an addition to the error-printer. Thirdly, we found that the probability of an error message changing was low once it was established. This slow-changing character pretty well obviated the need for an error printer.

One feature of ACID which seems common to library maintenance programs but that is rare in applications programs is the activity log. The activity log keeps track of which functional programs have been called, in what sequence they were called, whether they operated error-free, and the data of the call. The activity log tells us how often each function in the system is used. It provides an objective evaluation tool. Keeping track of the sequence of functional steps gives us an idea of what functions might be combined. We might mention that the greatest problem in the interpretation of the activity log seems to be user habits. Once a user establishes a sequence of steps that work he's unlikely to change that sequence even if he no longer needs some of the steps. We cannot say that the activity log has been a resounding success, but it does provide interesting information.

Another useful feature in ACID is a card editing program. The card editor lists all the input cards before any action is taken on them and makes whatever general editing operations are possible at that level. It also has the capability to generate control records from a more general input instruction. We mean rather low quality editing at this level; really good editing can be done only by the functional programs, which know specifically what to look for. The best that can be hoped for from the card editor is that it will catch the gross absurdities which would totally confuse the functional programs. Both the activity log and the card editor are part of what might be called the "executive suite"; their existence is useful only because the functional programs have been put together as a system.

In conclusion, we have found the executive, as exemplified by ACID, to be very effective. It brings us previously lacking user orientation. It brings us flexibility in large program systems. It brings us the opportunity to grow in a changing environment. We have four other comparable, but non-statistical, program systems with this same design currently operating. The design has been successful in every case; the systems are in use and rarely out of order for repairs. We recommend the design as a natural extension of the operating system.