```

114 331 001 001 004 001 002
10/18/73 0954
EYES CABPAGE
RESPONSE=NUF=WRONG
LETTUCE EYES
RESPONSE=NUF=WRONG
CABPAGE EYES
RESPONSE=VFC=RIGHT

```

(This output would continue for the entire solution sequence until the 10 correct trials criterion was obtained or until the maximum number of trials had been presented.)

Fig. 4. A sample of output directed to the permanent data file by the real-time "interpreter" program after processing the sequential file information shown in Fig. 2 and after receiving the S responses shown in Fig. 3.

reached, the "extra" information would not be presented, but would be skipped. Therefore, when the criterion is reached, the real-time program would scan through the sequential file, leaving all commands uninterpreted until the first occurrence of the !BREAK command. Commands would once again be interpreted once the !BREAK command was encountered.)

An example of the type of output that would be directed to the sequential file by the nonreal-time "generation" program is shown in Fig. 2.

Once the sequential file was output by the nonreal-time "generation" program, this file would be interpreted in real time by the real-time part of the experiment. The real-time part of the experiment is called the "interpreter" program, since it will interpret each of the 11 commands inserted in the sequential file by the "generation" program. The "interpreter" will input each record from the sequential file and execute the real-time operation specified by the keyword

command. In Fig. 3, a sample of the output directed to a S's terminal, as a result of processing the sequential file in Fig. 2, is presented. The responses input by the S would be requested by the "interpreter" program and processed as specified.

The "interpreter" program would also direct output to the permanent data file as a result of processing the file shown in Fig. 2 and the S's responses indicated in Fig. 3. A sample of several trials of output is presented in Fig. 4.

APPLICATIONS AND CONCLUSIONS

The multiprogram approach to real-time experimentation has been used to successfully implement a large number of real-time experiments. While many nonreal-time "generation" programs have been developed, only a single real-time "interpreter" program has been developed. Since the "interpreter" program is exceedingly versatile, it has proven to be adequate for most experiments. Because the "interpreter" is small (less than 1,500 locations), several copies of it are simultaneously in operation. The "interpreter" program can be adapted to purposes other than concept identification experiments. Computer-assisted instruction, serial list learning, and virtually any task that stores materials on a sequential file can be implemented. Such implementation only requires a nonreal-time "generation" program to be developed. It is hoped that the reader has acquired some ideas on how to utilize the multiprogram approach to implement new real-time experiments.

Behavior Research Methods & Instrumentation
1974, Vol. 6, No. 2, 262-266

Monitoring experimental results*

FRANK RESTLE and W. O. SHAFFER
Indiana University, Bloomington, Indiana 47401

No matter whether an experiment is a success, a failure, or a mixture of the two, the E needs to see his results immediately to be able to plan his next study. For this purpose, he needs a rapid, simple, relevant analysis of the data. We define a version of this data-summarizing problem and describe a monitor program to solve it. One major decision is what search method and data file organization to use: We use the n-dimensional array method. We specify dependent variables in two parts: first, a measure code like proportion correct, reaction time, etc., and second, a series of coefficients to specify what items will make up the desired dependent measure. Our program is interactive and makes extensive use of default options. Some suggestions are made for the implementation of monitor programs on other hardware.

There are three ways an experiment can come out: it can be a failure, a qualified success, or a complete success.

*This project was supported in part by Grant MH 16817 from the United States Public Health Service. The authors appreciate the thoughtful reading of this paper and the comments given by N. John Castellan, Jr.

If the experiment is a complete failure, this may mean, among other things, that all Ss responded at chance, that response times were very long, or that an apparatus failure produced invalid data. A superficial analysis of the overall results will usually reveal such problems, and the E will want to make some change in

his procedure and try again—or he may change his thinking.

If the experiment is a qualified success, it either shows a predicted relationship or uncovers a new fact, but with some complications. A replication may not replicate exactly, counterbalancing may result in interactions, or a variety of unexplained differences may arise. A detailed analysis, dividing data by conditions and scoring results appropriately, can reveal a partial success. The E will want to tie up the loose ends by explaining or eliminating unexplained lumps in the data, and this calls for new experiments.

If the experiment is a complete success, cleanly demonstrating a significant new finding, the E will at once want to exploit his success by testing various extensions and possible explanations of what he has found. He will want a rapid and complete analysis to determine the magnitude and details of his results, and to find leads to further studies. Thus, no matter what the result of the experiment, the E will want to analyze his data and determine what he should do next.

In our laboratory, the analysis of results has often been the critical factor that limits our progress. Each experiment requires a unique analysis, and we simply cannot write programs fast enough. Several research projects, including dissertation studies, have developed analysis “packages” that were very useful for those projects. We set out to generalize the concept of an analysis package.

PRIMARY DESIGN CONSIDERATIONS

We begin by listing the primary requirements for a general program to give the E a quick look at his data. The following seven requirements will be discussed below: storing and retrieving data, separating factors and levels of independent variables, handling many different dependent variables, being a monitor program and not a general statistical analysis package, handling many experiments, being interactive, and being fast.

The design of a monitor program starts with the problem of data retrieval. A program to summarize data must search a data base, i.e., data from the experiment. Emphasizing the retrieval aspect of this program leads quickly to consideration of how the data should be stored. Certainly, a well-organized data file will facilitate writing efficient retrieval programs.

A second major consideration for this program is the ability to partition the data in various meaningful ways as specified by the E. The program must be able to provide a breakdown of the data by factors and levels of the independent variables. Further, the E must be able to look at different dependent variables, e.g., proportion correct responses, reaction time measures, responses for the first five items vs the last five, etc.

This program was not intended to be a completely general, statistical analysis package; instead, this program was to be used to *monitor* the experiment. Monitoring

does not usually require chi square, t test, correlation, factor analysis, n-dimensional analysis of variance, or multidimensional scaling. To be able to monitor the experiment—to decide whether all the data are random noise, to see if differences are interesting or nonexistent or unexpected—such descriptive statistics as means, standard deviations, and frequency distributions would provide answers to many of the E’s questions. The programming task became manageable when we realized that there would be no significance levels calculated anywhere in the program. It is enough for the monitor program to tell the E whether he should continue and do statistical analyses.

We wanted the monitor package to accommodate a variety of experimental designs, with different numbers of factors and associated levels, and to handle experiments that we consider “typical.” For example, our first set of data for this package came from 103 Ss, who each read three different stories, presented in each of three ways, and answered 30 questions about each story; true-false responses and latencies were recorded. In the retrieval program, this experiment was treated as a 3 by 3 by 37 Ss (maximum) with 30 questions per cell. We realize that many laboratories (e.g., physiological laboratories) may not agree that this type of data is “typical,” and our ideas may have to be modified before being useful to others.

Although a batch-processing approach is possible, we preferred an interactive approach to enable the user to pursue differences as they were uncovered rather than specifying all his questions in advance by a (perhaps lengthy) series of control cards. In an interactive system, with an appropriate sequence of questions, an E may be able to locate what appears to be a large main effect as being due to a particularly easy series of items in one list in one group in one condition.

An interactive program must be fast. The user sitting at the terminal should not be kept waiting, lest the benefits of interaction turn into a burden.

The above set of requirements is mentioned early in the paper partly to argue that the problem is potentially manageable by putting the proper pieces together. More importantly, we mention this list because we achieved all of these considerations in our laboratory. Our initial interactive version of this program was written in basic FORTRAN IV, runs in less than 6,000 16-bit words of core memory, uses 21,000 words of disk space for the example data described above, and prints out a calculation every 6 to 10 sec. We are quite pleased with our initial success.

SEARCH METHODS AND DATA FILE ORGANIZATION

As indicated earlier, a well-organized data file makes it easier to write efficient search programs. There are three methods for structure. Each will be discussed in turn.

Method 1 we shall call the “sequential label check”

method. For this method, stored with each S's data is a complete identification code, e.g., level specification for all factors. The search program to use such a file will begin with the first S on the file and proceed sequentially to the end of the file, asking "do I want this S's data, yes or no" for each S on the file, as he is encountered. Two variations on this method are possible. For example, if calculations for six conditions are desired, the program could make six complete passes through the data file and compute statistics for one condition on each pass. Alternatively, one pass could be made and tallies could be kept on six separate sets of variables. The latter method is faster, but uses more tallying space and makes a slightly more complicated "do I want this S" decision. In choosing between these two variations, the speed advantages of the latter make it generally preferable. We have tried the former method and found execution to be painfully slow, but we would not choose either because many queries of our system (especially after the first few) involve only one or two groups and include only about 1/10th of the data file; thus considerable time is saved by not accessing Ss whose data will not be used in the current computations.

Method II we shall call the "table-driven" method. This generic name is used to lump together the "address tables" and "pointers" methods, and mixtures of the two. To avoid accessing unneeded Ss, a table of addresses is computed once and maintained at the beginning of the data file or kept in memory. For a study with six groups, the table might contain the starting address for each group. If each group had three treatments, three more starting addresses might be kept with each group for the beginning of each of these three treatments. Extending this scheme to a five-factor design produces a very complicated table, including start and end addresses for all cells in all factors except Ss; we suspect that for three or more factors, the program would require another function for search and retrieval within this complicated address table. However, for programmers experienced in handling complicated sets of tables, the "address table" method could be a very good choice.

The "pointer" method stores with each S a pointer to the next "similar" S. This seems efficient, but leads to five pointers per S for a five-factor design. This pointer or link scheme would also get complicated quickly and use more file space. A combination of these two methods (e.g., address tables for all but one factor and one set of pointers) would alleviate some problems but would create others. Thus, we rejected this class of methods, because adding each additional factor increases space and work in a roughly exponential fashion, while adding factors to an n-dimensional array method increases work by adding a linear component (one extra multiplication in the address computing function).

Method III we shall call the "n-dimensional array" method. In this method, the data are stored without labels or pointers but in a highly structured fashion—namely, the way a multidimensional array

would be stored. To locate a subset of data, a function uses the factor and level specifications (typed in by the user) to calculate a series of addresses where the desired data are stored. One disadvantage of this method is that the computer must compute addresses; however, since this calculation can be made very fast, this difficulty is relatively minor. An occasional problem is wasted file space due to missing cells (not nested designs but "nonrectangular" designs). Unequal number of Ss is the only frequent cause of this problem that we have encountered; however, since group sizes typically have not differed by more than 10 Ss, the space problem has not been considered serious. We believe the speed of this method more than compensates for these disadvantages.

In summary, the n-dimensional array approach is working well for us. We believe it is superior to the "sequential label check" method in only accessing Ss whose data will be used in the current calculation and only accessing them once. We believe that the n-dimensional array approach is superior to "table-driven" methods because the latter require more space and decrease in relative efficiency as the number of dimensions increases.

IMPLEMENTING AN n-DIMENSIONAL ARRAY APPROACH

Our monitor program handles independent variable retrieval, using the n-dimensional array approach. This structure is the same as that in many factorial design programs. The essential pieces of our system are (a) a program that reads cards with level specifications and stores them in a disk file in the highly structured form of a multidimensional array, (b) a function which computes addresses given the subscripts indicating the desired element of the multidimensional array, and (c) an interactive retrieval program.

The storage program reads in the number of factors and number of levels for each factor, uses a variable format to read data cards and check level specifications, appends a sequential S number within each cell to the level specification, copies the cards to a temporary file, and uses the counted maximum number of Ss per condition to write data to the final file in the appropriate n-dimensional layout using Ss as the last factor.

In the interactive program, approximately one-third of the code is devoted to interactive input and output to the user (including the detection of several default options and error conditions). Another third is devoted to calculation routines for means, standard errors, and response frequency distributions. The final third is the retrieval section, and it consists mainly of an awkward piece of code which performs the simple function of incrementing subscripts. This section is analogous to a highly nested set of n FORTRAN DO loops used to access the appropriate elements of the n-dimensional array.

The components of the data storage program and the interactive program can probably be programmed with only the usual number of bugs by a FORTRAN programmer with 1 year of experience. However, the handling of n -dimensional arrays is not a standard feature in FORTRAN. Thus, many programmers are unfamiliar with the techniques. The techniques are well established and found in some form in many computer science books (Knuth, 1968, p. 296; Wegner, 1969, pp. 266-268; Elson, 1973, pp. 112-114). In the remainder of this section, we outline the computational procedures used by the address calculation function.

Many programmers have discovered that in the one-dimensional array the address of Element I, J can be computed by

$$IJ = (J - 1) \times IMAX + I$$

where $IMAX$ is the length of the I dimension. The result of the multiplication tells how far into the array to go to find the desired column, then the position in the column is added on. Reference books explain how to generalize to, for example, a four-dimensional element I, J, K, L in an array of size $IMAX \times JMAX \times KMAX \times LMAX$, where the address may be computed by

$$IJKL = (L - 1) \times IMAX \times JMAX \times KMAX + (K - 1)$$

$$\times IMAX \times JMAX + (J - 1) \times IMAX + I$$

Because the same multidimensional data array will be used throughout the program execution, we can save time by computing a few terms which will be constant for a given array. Computing these terms once in advance and storing them in a table in memory will save considerable time. To begin, we define the constants

$$LMULT = IMAX \times JMAX \times KMAX$$

$$KMULT = IMAX \times JMAX$$

$$JMULT = IMAX$$

and by algebraic expansion obtain

$$IJKL = L \times LMULT + K \times KMULT + J \times JMULT + I \\ - LMULT - KMULT - JMULT$$

Finally, we define a term called the "lower bound effect" as

$$LOWER = LMULT + KMULT + JMULT$$

and obtain

$$IJKL = L \times LMULT + K \times KMULT + J \times JMULT + I \\ - LOWER^1$$

Thus, for any four-dimensional array, we can define four constants which greatly simplify the calculations. In general, there are n such constants used for calculating addresses in an n -dimensional array. Perhaps the best source for this last set of computational shortcuts is an ALGOL reference manual for any large university computer.

IMPLEMENTING DEPENDENT VARIABLE MANIPULATION

Dependent variables are specified by the user in two parts. First, he enters a measure code such as PC for proportion correct, RT for reaction time, or RF for response frequency distribution. Second, the user enters a series of weights or coefficients to specify what questions will make up the desired dependent measure. For instance, an entry of RT followed by a series of five 1s followed by all 0s would give the reaction time for only the first five items. These coefficients are completely analogous to weights which specify planned comparison tests, e.g., the first two items could be compared to the next four by entering $-2, -2, 1, 1, 1, 1$, and all the rest 0s. This system gives us a great deal of flexibility in specifying and creating dependent variables.

DEFAULT OPTIONS

A general tip to designers of interactive systems is to put in as many default options as you can think of. Users quickly tire of repetitively typing in the same long, detailed, complete set of answers to the same set of questions while changing only one parameter. In our interactive program, an option to type a zero instead of a complete list has been very popular. For example, typing a zero can give you one mean for all factors; typing a factor number and a zero can give you all levels of that factor; typing a zero after the first factor can give you all remaining levels of all remaining factors; and typing a single zero to specify coefficients sets all coefficients to one, thus giving data for all questions. Thus, our general rule is "when in doubt, type a zero" and let the program do it for you. Another popular option involves specifying coefficients by typing "99" to indicate that all coefficients are the same as the previous set. Also, we make our prompting messages as brief as possible. We are now collecting suggestions to make the system even more convenient for users.

IMPLEMENTATION ON OTHER HARDWARE

Our monitor package uses the equipment available in our laboratory, which includes a CRT terminal and a random access disk. However, almost all the key elements of our system can be implemented on other hardware.

The one crucial element in our type of monitor system seems to be a mass storage device. Since it is easy to collect more data than can easily be fit into core

memory, some form of secondary storage is essential. Yet even the smallest minicomputer owner need not despair, for it does not have to be *your* secondary storage. By transferring your paper tape or cards immediately to a file at the university computing center, you should be able to do many of these things either interactively or using batch processing.

A random access device is also not essential. Sequential access files can be made to perform the same function by tolerating an occasional rewind operation or by being careful to generate your n-dimensional array addresses in a constantly increasing manner and occasionally calling a "skip" routine. However, initially structuring data into n-dimensional form is more difficult on sequential devices, and it may be more efficient to switch to the "sequential label check" search method.

Another possible approach to getting a quick look at the data is doing some small analysis on-line at the end of the experiment. Many PDP-8 users who run a single S at a time find that the end of the experimental session, between Ss, is the best time to print a data summary. The user must do considerable calculating to gather his data together for a general monitor system, but still can keep track of where his experiment is going by using just the on-line individual summaries.

CONCLUSION

In conclusion, we believe that every E must be able to

look at his data in order to make wise decisions about how long to run an experiment and which experiment to run. We propose a general, interactive program with the capability of looking at both independent and dependent variables in various combinations. Although the methods of looking at data may differ in different laboratories, we strongly encourage Es to transform bits inside the computer into interpretable results as rapidly as possible.

REFERENCES

- Elson, M. *Concepts of programming languages*. Chicago: Science Research Associates, 1973.
- Knuth, D. E. *The art of computer programming*. Vol. 1. *Fundamental algorithms*. Reading, Mass: Addison-Wesley, 1968.
- Wegner, P. *Programming languages, information structures, and machine organization*. New York: McGraw-Hill, 1969.

NOTE

1. An alternative formula, based on polynomial arithmetic techniques, is

$$IJKL = [(L \times LMAX + K) \times JMAX + J] \times IMAX + I - LOWER$$

In assembler language, or on most efficient FORTRAN compilers, this alternative would be faster for a fixed number of dimensions. For a variable number of dimensions (the case our system deals with), both formulae would have to be implemented in a FORTRAN DO loop; in this case, the question of which formula would produce more efficient assembler language code would be more dependent on the compiler used.