A MONTE CARLO PROCESSOR
Version 2.0

by

David Alan Grier

TECHNICAL REPORT No. 54
October 1984

Department of Statistics
University of Washington
Seattle, Washington 98195
A system to do Monte Carlo experimentation is herein described. Its design goals are to make Monte Carlo Experimentation easier to do, to help to unify the scattered Monte Carlo literature, to decrease the time required to do a Monte Carlo experiment and to encourage researchers to treat their Monte Carlo work as experimentation and to use good design of experiment techniques in their work. The system takes an experimental design as input and produces a FORTRAN program to perform the experiment. The experimental results are loaded, by the system, into anyone of many statistical packages for analysis.
Preface

Monte Carlo methods are an important part of statistical research. With the ever increasing availability of computers plus the growing interest in computationally dependent statistics, such as the bootstrap, these methods are growing in importance. A recent survey of papers published in *JASA*, *Biometrics*, *Applied Statistics* and *Technometrics* estimated that 18% of articles published in those journals used some Monte Carlo technique in their research (Hauk and Anderson, 1984).

To date, however, there is no focal point for research in Monte Carlo techniques, at least in terms of computer software. The literature for Monte Carlo techniques is scattered over at least four fields, statistics, computer science, numerical analysis and operations research and it is often difficult for someone trying to do a Monte Carlo experiment to find all the information that they need to perform that experiment. For example, a researcher who wishes to compare the power of several different statistical tests over different stochastic models could find the algorithms needed to generate the proper random quantities and to do the tests but, without a familiarity with Monte Carlo methods might not realize that using a common variable technique would be a good way to minimize the variance of the differences of the power functions. Of course, a Monte Carlo experiment performed without the common variable technique is just as valid as an experiment done using that technique. The reason for using common variables is that differences between the techniques may be hidden by the variances of the experiments if common variables are not used.

A greater problem still is that of using Monte Carlo methods with known deficiencies. Hauk and Anderson (1984) report that a journal article published in 1981 used the random number generator RANDU, an algorithm whose shortcomings have been known for at least a decade (Knuth, 1969). Also, it is common to see articles which use the Box-Muller transform to generate normal random deviates with no mention of the problems of using linear congruential generators with that transform. (Bratley, Fox and Schrage, 1983)

Recent books on the subject of Monte Carlo experimentation have been helpful in unifying the Monte Carlo literature and disseminating information about good Monte Carlo technique, (Bratley, Fox and Schrage or Rubenstein, 1981) but there still does not exist a software system which includes a large part of the body of information on Monte Carlo techniques. This report is a proposal for such a system. We will do our best to see that it contains the best

would also like to thank Richard Kronmal. He first proposed the basic idea of a Monte Carlo Processor and the author would not have made so much progress without his help.
has become the focal point for research into Generalized Linear Models, and that future additions to knowledge will be included in the system so that they be widely used.

But perhaps the most important reason for having a system to do Monte Carlo experimentation, is that we would like researcher to be able to think in terms of the design of the Monte Carlo experiment. Monte Carlo work is experimentation, just like experimentation in agriculture, chemistry, biology or any field, and to be done properly demands an experimental design. So often proper design is overlooked. One must assume that this is due, at least in part, to the fact that programming a Monte Carlo experiment is often a long and arduous task, demanding that the researcher collect the proper algorithms to generate the random quantities, write the code to conduct the experiment and accumulate the data. So much effort is spent in writing the program that the researcher is often grateful beyond measure that it works at all and has little time for making sure that the experiment is properly designed. In addition, a good design may become apparent only after the first results of a preliminary experiment but by then the researcher may lack the time to be able to program and to carry out that designed experiment.

The Monte Carlo Processor, described here, allows a researcher to think in terms of experimental design. They will need to identify the questions that the Monte Carlo experiment is to answer, the experimental output which will answer those questions and the design which will get those outputs. The Monte Carlo Processor will take a design of an experiment and produce a program which will perform it. If the need arises, the researcher may change design or the elements (parameters, distributions, etc.) of the design and rerun the experiment. While all this could be done in a high level language such as FORTRAN (if FORTRAN may be called a high level language), this system changes the focus of the work which the researcher will have to do from "How can one program a given Monte Carlo experiment?" to "What kind of Monte Carlo experiment will best answer my questions?".

A final reason for this Monte Carlo processor is that it will help solve Monte Carlo problems faster. There are a great number of details in programming a Monte Carlo experiment, accumulating data, the control flow of the whole program, etc., which need to be done and debugged for each experiment. These details can be handled by the Monte Carlo processor. The total time required to do the experiment, from the designing of the experiment, through the preparation of the program to do the experiment, is therefore reduced.

for discrete simulation, such as queue structures, are not implemented. They will come with later versions., but rather an attempt to include in one system, all the tools to do Monte Carlo experimentation. Accordingly, while we have
followed the dictum that "Simple things should be simple to do" and have made sure that we have selected what we feel are the best possible algorithms. We have made sure to document the properties of the methods we choose to use and made it possible for the user to choose between different methods with different properties and even to add new algorithms if that seems desirable.

What follows is a description of that processor. Section 1 is a very brief introduction to Monte Carlo experimentation. Section 2 contains examples of the use of the Monte Carlo Processor. Section 3 describes the structure of the Processor and sketches its current implementation. The design decisions and the portability concerns are discussed in Section 4. Finally, section 5 is a summary of the language.
Section 1: Introduction to Monte Carlo experimentation

Before we can go ahead and discuss the proposed Monte Carlo Processor, we should first present the basic ideas of Monte Carlo experimentation in order standardize our terminology. Monte Carlo experimentation involves the use of random numbers, generated on a computer, to solve problems. In the typical Monte Carlo experiment, the researcher will repeatedly generate pseudo random numbers, apply functions (usually some statistics which are being studied) and accumulate the results of those repetitions. Often, they will be interested in the distribution or variance of those functions of the random numbers. The random numbers will come from one of some family of distributions. Choosing the design of the experiment will involve choosing which distributions and which statistics to use in the experiment.

Before we get too far afield in abstractions, let us ground the idea of a Monte Carlo experiment in a concrete example. This example is in the spirit of the Princeton Robustness study (Andrews et al., 1972). However, it is simple in order to illustrate the basic ideas. In our example, we will study the variance of the sample median relative to the variance of the sample mean. The mean of a sample from the Normal distribution has a smaller variance, asymptotically, than does the sample median. However, when the sample is taken from the Cauchy distribution, a distribution with heavier tails than the Normal distribution, the situation is reversed. The median has a smaller variance. A reasonable question to ask is how much heavy tailedness is needed for the median to have smaller variance than the mean. What would we do would be to compare the variances of the two statistics on data sets generated from some family of distributions that start at the Normal family and get progressively heavier tailed. The t family or the Contaminated Normal family might be a good family to choose. For the purposes of this example, let us choose the t family and include the Normal distribution for comparison purposes. Out of the t family, let us use the t distributions with 1, 2, 5, 10 and 30 degrees of freedom. The t distribution with 1 degree is the Cauchy. As the degrees of freedom increase, the t distribution more closely resembles the Normal. For each distribution that we chose, we would generate a number of samples. Let us call the number of samples n. Each sample would be of a certain size, say k, and we might want to vary this size in order to compare the variance of the two estimators. In this case, let’s vary k over the set {10, 25, 100}. From our n sets of samples of size k, we can get n values for the mean and median from each of our chosen distributions. By computing the variance of these n values, we can compare the variances of the
The process of doing a Monte Carlo experiment can be divided into three parts. There is first the design of the experiment. This involves choosing the statistics to be studied, the distributions from which the random numbers are to be generated, the number of replications of the experiment, the other parameters that need to be chosen (such as sample size in the above example) and how these elements are to be put together in a unified design. For example, if we performed the experiment, described above, 1000 times for each combination of sample size (10,25,100) and distribution (t(1),t(2),t(5),t(10),t(30) and Normal) then we would be doing a factorial design with fixed replications. There are sequential designs, D-optimal, block designs and others which might be used in a Monte Carlo experiment. In our Monte Carlo Processor, a high level command language is used to specify the design.

The second part of the process of doing a Monte Carlo experiment is the actual performing of the experiment. That is done by our Monte Carlo Processor. It takes the design of the experiment and generates a FORTRAN program to do the experiment. The results of the experiment are stored in a file.

The third and final part of the process is the analysis of the results of the experiment. That is done, in our system, by a conventional statistical package. Our Monte Carlo Processor will take the experimental results and put them in the most acceptable form to be read into any of several different statistics packages. The Monte Carlo processor does have some ability to analyze the results. But most researchers will probably want to work in an environment that is more suited for data analysis.
Section 2: Introduction to examples

Before presenting the design of the Monte Carlo Processor, we will present a few examples of Monte Carlo experimentation drawn from the journals or current research projects. After a brief explanation of each example, we will show how it could be done with our Monte Carlo Processor. We supply only a terse explanation of the details of the experiments in the form of comments, but the reader should still be able to make sense of the code as presented here. The details of the syntax of the language can be found in Section 5. For the motivation behind each of the experiments, the reader is directed to the original journal article.
Example: Power of Tests

The following example is based on a Monte Carlo experiment by Bartels, published in *The Ranks Version of von Neumann's Ratio Test for Randomness*, *JASA*, 77, 377, (March 1982), pp 40-46. In the article, the author compares the power of the von Neumann ratio test for autocorrelation, the power of a rank version of the same test and the power of a runs test. We will restrict ourselves to the von Neumann test and the rank version of the von Neumann test. We will perform an experiment to calculate the power of these two statistics under various deviations from the null hypothesis of no correlation when the tests are performed at the .05 level. Using an ar(1) model we will generate data with autocorrelation ranging from -1 to +1. We will generate data from 4 different distributions, the Normal, the Cauchy, the Contaminated Normal (with contamination parameter .05 and contamination standard deviation 10), and the Double Exponential. Finally, we will compare these statistics on data sets of length 10, 25 and 50.

In doing the Monte Carlo experiment, we will make two functions which will calculate the critical values for the two tests (see page 44 of the above-mentioned article).

The data returned by the experiment to the statistics package for analysis will be the power of the test for the given set of parameters.

In the code, given below, the prompts from the User Interface are given in bold and comments are contained in /* slashes and stars */.

**set-up code**

```plaintext
function critvn(t) {
  if (t == 10) { critcv = 1.36} /* critical values for */
  else if (t == 25) { critcv = 1.45} /* von Neumann test */
  else if (t == 50) { critcv = 1.59} /* critical values for rank */
  else critcv = 0 /* error case */
}

function critrnn(t) {
  if (t == 10) { critcv = 1.04} /* von Neumann test */
  else if (t == 25) { critcv = 1.36} /* critical values for rank */
  else if (t == 50) { critcv = 1.54} /* critical values for rank */
  else critcv = 0 /* error case */
}
```
random numbers

x = mkarma(@dist, r, 0)  
?dist: rnorm(t)  
       rcauchy(t)  
       rnorm(t,.05,.10)  
       rdeexp(t)  

?t: 10,25,50     /* sample sizes */  

?r: seq(-1,1,.1)  /* auto correlations from -1 to 1 */

statistics
vn = ifelse(sum(diff(x)**2)/(var(x)*(len(x) - 1)) > critvn[len(x)], 1, 0)  
     /* von Neumann test */  
     /* both tests return 1 for reject, 0 for accept */  
rvn = ifelse(sum(diff(rank(x))**2)/(len(x)*((len(x) ** 2 - 1)/12)  
          >critrvn[len(x)], 1,0)  
     /* rank version of von Neumann test */  
     /* backslash is continuation char */

return(vn,mean(accum(vn)))
return(rvn,mean(accum(rvn)))

number of replications: 5000
Example: Statistics on a Circle

In this experiment, we are going to study random movement on a circle. Examples of this kind of movement include shifts in direction of the wind and the direction which birds choose to fly or whales choose to swim (in open ocean). We will explain the mathematical model using the example of the flying bird. The basic model says that a base direction is chosen (the initial direction a bird is flying). The bird chooses a new direction by deviating from that base direction. This new direction becomes a new base direction and the bird repeats the process a finite number of times. Mathematically we would say that the final direction is the sum of the initial direction plus a number of rotations.

In the experiment done here, we are going to choose rotations at random from the cardioid distribution, a distribution which corresponds, in a sense, to a triangular distribution on a straight line. We would like to know the final distribution of a finite sum of those transformations from a given initial starting direction. Our intuition suggests that if there are enough rotations, then the resulting distribution might be close to uniform on the circle. The cardioid distribution has one parameter, rho and for rho equal to 1/2, the cardioid distribution is uniform on the circle and it is known that the sum of uniform transformations on the circle are uniformly distributed. We will be trying to find the small sample distribution of the sum of transformations for rho less than 1/2, for which there is no theoretical result.

**set-up code**

```r
function cardioid(k,rho) {
    real(x,100) /* this function is for neatness */
    x = rvec(2,k) /* it generates cardioid data*/
    cardioid = atan(ifelse(runif(k)>2*rho*(x[1,1]+1),x[1,1],x[1,1]),[x,2])
}
/* above statement generates angles */
/* of transformations */
```

**model code**

**random numbers:**

theta = cardioid(k,rho)

```r
?k: 5, 10, 15, 20 /* sample sizes */
```

Repetitions: 300 /* number of replications of */
/* experiment */
Example: The Bootstrap

The Monte Carlo Processor also has the ability to do a bootstrap using the rep function. The following example is drawn from Efron and Gong, A Leisurely look at the Bootstrap, the Jackknife and Cross-Validation, The American Statistician, vol 37, no 1 (Feb. 1983), 36 - 48. The authors are using the bootstrap to estimate the distribution of the sample correlation. The data is this problem are LSAT scores (the x variable) and undergraduate GPAs (y) for a population of students.

```r
z = read("dataset.1")  # data are read in
x = z[,1]               # x in first col
y = z[,2]               # y in second col
rhocdf = rep(cor(rsample(x,y,len(x)),1000))
```

The vector rhocdf contains the raw bootstrap data for use in the target statistic package for estimation of the distribution of the correlation coefficient. Taking the functions mean(rhocdf) and var(rhocdf) would have produced the bootstrap estimate of the mean and variance of the correlation coefficient.
Section 3: Basic Design

The Monte Carlo Processor produces a FORTRAN program, hereafter called the MC program, which performs the Monte Carlo experiment. That program is compiled by the local FORTRAN compiler and runs asynchronously (in the background, in batch mode) of the Monte Carlo processor. The most basic form of the program generated by the Monte Carlo processor is shown below:

\begin{verbatim}
Initialization code

Loop over parameters

Set up any fixed model code

Loop over replications of the experiment

Generate Random quantities

Evaluate Statistics

End Replication Loop

Store results of experiment

End loop over parameters
\end{verbatim}

Figure 1.
Structure of generated Program

The Initialization Code (1) will consist of code which reads in values from a file or does any other computation which has to be done in order to set up the experiment. The Set Up Fixed Model Code (2) is for the calculation of any quantities which may change as parameters change but are unaffected by the generation of the random numbers. A good example of a use for this section of code would be the situation in which someone is performing a Monte Carlo experiment using a linear model, \( Y = X\beta + \varepsilon \), and that the column vector \( \beta \) was one of the parameters. One would rather not recompute the matrix product \( X\beta \) each time a new set of random errors, \( \varepsilon \), is generated. But, the quantity \( X\beta \) may be calculated in
Section 3.1: Structure of the System

The Monte Carlo Processor can be seen as four logical units. These units are: the User Interface, the Program Generator, the Function Library and the Converter Routine. The relationship between these units is pictured in Figure 2 below.

![Figure 2. Structure of System](image)

The User Interface controls the whole system. Through it, a researcher prepares and executes experiments, modifies and re-executes old experiments and prints the results of experiments or directs them to statistical packages, such as S, Minitab etc., for analysis. The designs of experiments are specified in a language called, appropriately enough the Monte Carlo Language (MCL). The User Interface takes these designs and passes them to the Program Generator.

The Program Generator translates the experimental design into a FORTRAN program, which it, in turn, passes to the local FORTRAN compiler. The Function Library contains all the routines needed to manipulate the structures of the experiment as well as the routines to generate random numbers and evaluate certain statistics. It may be used as a traditional FORTRAN Library, such as EISPACK or IMSL. In normal usage, the Function Library is bound to the FORTRAN program and the resulting program is executed. Experimental results as
Section 4.0: Design Decisions

The original design of this program had it operating within the scope of a standard, interactive statistical analysis package. There was a certain strong argument for that kind of implementation. After all, many of the statistical functions would already exist in a statistical package and it would be very easy to analyze the results. An initial complete design was created for the S (Becker and Chambers, 1984) statistical language by the author as part of the work for his Phd dissertation (Grier, 1984) and an elementary implementation was made in the spring of 1984.

At first glance, the S package seemed the ideal vehicle for the Monte Carlo Processor. It contained a large family of statistical procedures and random number generators already. It had a nice macro facility and one could add new functions to it without recompiling the whole thing. The preliminary version of the Monte Carlo Processor took a description of the experiment and converted it into S code and passed the S code to the S processor. This implementation was heavily tested during May of 1984. The basic concept for the user interface and the generated programs were similar to those in the design described in this document. They proved to work reasonably well. The major drawback of the S implementation was that it was far too slow. The lack of speed appeared most dramatically in one study (McKnight, 1984), but was present in all applications. A modest study of a survival analysis problem with a 3 x 3 x 2 factorial design with 50 replications took over 48 hours of wall clock time on our Vax 11/750 (3+ hours cpu). A careful study of S showed that it evaluated expressions by creating a parse tree for each expression and then evaluating each leaf and thereby slowly reducing the tree. Such a method is perfectly adequate for a statistical analysis package but is not too good for an application which demands a great degree of repetition such as Monte Carlo work. There was simply too much redundant creating of parse trees and there appeared no easy way of generating and executing some kind of intermediate code.

Hence it appeared that the only choice was to make a special system solely for Monte Carlo work. Our experience was that a good sized Monte Carlo Experiment could use a large amount of computer time even when programmed in FORTRAN and that it would probably be best to run the experiment in some kind of batch mode but that we would want some kinds of interactive tools for debugging and analyzing the results. Once we removed the design from the S environment, there appeared no reason why we should limit ourselves to just one statistical language. The question then became:

as to how we were going to analyze the experimental data, the only decision we had left to make was how to run the Monte Carlo experiment itself. We felt that we had three options. 1) We could make a compiler for an S-like language, 2) we
could make a new interpreter which would store some kind of internal code and would be faster for repetitive problems than the S interpreter or 3) we could design a language which would be compiled into a FORTRAN program. To cut to the heart of the matter, we chose the last implementation. The first was rejected because it was not portable from machine to machine. The second was rejected because, while one might be able to make an interpreter that was faster than the S interpreter for repetitive problems, we could not guarantee that it would be fast enough to be useful for even medium sized problems. The third technique seemed to be both portable and adequately fast. It was thought portable, because standard FORTRAN compilers exist on most machines. FORTRAN tends to be fast because it is a fairly low level language, many of the compilers have optimizing options, and the scheme for associating memory with data structures is simple.

After choosing the implementation that performs the Monte Carlo experiment by producing a FORTRAN program which is compiled and run in some kind of batch mode, we were able to see other advantages of our design. Subroutines from any FORTRAN package, such as LINPACK, EISPACK, IMSL could be incorporated easily. Our library of routines would be available for use outside of the Monte Carlo Processor environment. While it would not be as easy as in S to add new functions, we could still make it so that it was not too difficult to expand the language.

We also decided on using the UNIX compiler tools, Lex and Yacc, for producing the Processor. Given that we would need to produce both a lexical analyzer and a parser for our package, we saw little reason why we should write them ourselves from scratch when such tools as Lex and Yacc were available. The primary advantage that they gave to us was that we were allowed to think in terms of the design without having to worry about how we were going to implement the parser. In addition, we became free to implement and test ideas in the language, about which we felt unsure. Changing and correcting implementation of the design became no more difficult that changing the design of the language.

Section 4.1: Portability issues

As with most software produced in this age, we claim that our system is highly portable (as well as user friendly and all the other buzz words). The system was developed on a VAX 11/750 running Berkeley UNIX 4.2. The development process has been highly dependent upon the UNIX software tools. How-
output program is written in a very conservative dialect of standard FORTRAN 77. All arithmetic machine dependencies such as number format and machine epsilon are adjusted automatically (Malcolm 1972). There is no symbol processing done in FORTRAN. IO is handled very simply using standard FORTRAN FORMAT editing symbols. The author's feeling is that the FORTRAN programs, which actually do the Monte Carlo experiment, can be run on any machine with a standard FORTRAN 77 compiler with no conversion at all.
Section 5.0: User Interface

The User Interface is the only part of the Monte Carlo Processor with which the researcher has any direct contact. While running the User Interface, the researcher can specify an new experiment, direct the output of an experiment to a specific statistical package (such as S, ISP, SPSS, BMDP etc), edit and rerun an old experiment or stop and restart an running experiment. The User Interface takes the following commands:

edit<name>

Edit and run an old experiment. <name> is the original name of the experiment given in the "new" command. See section 5.1.2 below.

load<name>

Load results from experiment <name> into statistics package <pack>.

load raw <name> <pack>

Load raw results (see description of accum command under language summary) into statistics package <pack>.

new<name>

Define and run a new experiment. <name> is the name of the experiment. The description of the experiment is stored in a file called <name>.e See section 5.1.1 below.

stop<name>

Stops experiment immediately. All results completed are saved. The experiment may be restarted at this point with nothing lost.

restart<name>

Restarts experiment where it was last stopped. If experiment has never been run, it is started from the very beginning.
run<name>

Runs experiment <name> as above. It substitutes the new parameter values for the old set.

Section 5.1.1: new command

When one is setting a new experiment, one uses the new command. This command will then prompt the user, asking for the five sections of the program. The first thing the User Interface will do, after receiving the "new" command, is to prompt the researcher for the Set Up Code by typing on the screen:

Set-up Code

The researcher will respond by typing in any code for this section. (See language summary below for a description of the language to be used.) To get out of this section and move on to the next, the researcher simply types two consecutive carriage returns (<cr>), and the User Interface will prompt:

Fixed Model Code

Here the user types in any code for this section and moves on to the next section the same way they left the Set-up Code section, by typing two consecutive carriage returns. The next two sections of code, Random Numbers and Statistical Procedures are handled in the same way. The last section of code, Store Data, is generated by the Program Generator from the "accum" and "return" statements in the preceding four buffers. The user doesn't have to specify directly the code in that section.

Parameters are specified as the researcher types in the design of the experiment. The User Interface assumes that any quantities on the right hand side of an assignment statement are numeric parameters and will prompt the user for the values of the parameters for which the experiment is to be performed. To specify a functional parameter, such as distribution for the random quantities, the researcher uses the following syntax:

lines following the prompt. The entry of two consecutive carriage returns gets the user out of the prompt.
At the end of the execution of the new command, the User Interface will prompt for the number of replications for each experiment to be performed. The default value is 1.

Section 5.1.2: edit

The edit command allows a researcher to go back and modify an old experiment and then to rerun it. In the current implementation of the "edit" command, the experiment description file is loaded into an editor and the researcher can manipulate any part of the file.
Section 5.2: Program Generator

The Program Generator takes the experiment description from the User Interface and turns it into a FORTRAN program. The Program Generator is actually a compiler (or translator if you will). The language it accepts as input is formally defined and summarized below. It is this language which a researcher users to describe the Monte Carlo experiment.

Section 5.2.1: Language Summary

Data Structures

All data is of type real. The program supports scalars, vectors, and tables. Matrices are a subtype of tables. The maximum size of Vectors and Tables must be declared. For example, the statement:

```
REAL(a,5,5)
```

declares "a" to be a matrix with maximum size 5 x 5. It is, initialized as having actual size 1 x 1 and as having with value 0. Its actual size is increased when a structure is assigned to it. An error is generated if either dimension exceeds 5.

Unary Operators

```
-
```
Unary minus

Binary Operators

The following binary operators work on all data structures. The actual size of all operators must be the same or else an error is generated. An exception to this rule is that a scalar may be used on either side of these operators with any structure on the other.

```
=
```
assignment
The following operators work on vectors or matrices.

\%
\%
\%c
\dot product (Matrix multiplication)
cross product (Kronecker product)

The following are logical operations and produce a 0 for true and 1 for false. They may be used on structures in the appropriate places.

== .EQ.
<= .LE.
>= .GE.
< .LT.
> .GT.
! .NOT.
&& .AND.
|| .OR.

Control Structures

if (expression1) statement1
if (expression1) statement1 else statement2
while (expression1) statement
do ident = expression1, expression2, expression3 statment
for(expression1;expression2;expression3) statement

Compound Statements
Any group of statements may be grouped in braces {} and treated as a single statement.

Functions
The functions will probably contain a subset of the S language functions. Since the S functions are written in FORTRAN, we are actually dealing with

anoova analysis of covariance
aperm permute an array
append
apply
approx
arima
backsolve
c
cancor
cbind
ceiling,floor,trunc,round
chol
code
cumsum
cut
diag
diff
discr
eigen
fft
gs
hat
ifelse
lifit
len
loglin
max,min
mean
median
ncol,nrow,ndims
order,sort,rank
prcomp
prod,sum
reg,robreg,rigreg
repeat
rev
seq
solve
svd
trig and
edit data, data merging
apply function to part of structure
approximate function using data
fit arima model using Yule-Walker eq
backsolve linear equations
combine expressions into vector
canonical correlation analysis
bind vectors into table
rounding functions
cholesky decomp
create category from discrete data
cumulative sum of structure
cut continuous data
make diagonal matrix or get diagonal
difference vector
discriminant analysis
eigen values of matrix
fourier transform
gram-schmidt decomposition
compute hat matrix
if else function
minimum absolute residual fit
length of structure or ith dimension
fit loglinear model
maximum and minimum of structure
mean of vector
median of vector
number of rows, columns or dimensions
sort and rank data
principal component analysis
product or sum of structure
various linear regressions
duplicate values
reverse vector
sequence of numbers
solve linear equation and invert matrix
singular value decomposition
Random Functions

runr  uniform from several congruential generators
runit uniform from tausworthe generator
rnorm normal from several transforms
rt  Student's t
rgamma gamma
rf  F
rbin binomial
rber bernoulli
rcauchy cauchy
rexp exponential
rweibull weibull
rchisq chi-square
rpois poisson
rbeta beta
rgeom geometric
rneg negative binomial
rmult multivariate Normal
arma stable arma
rmat random matrix with columns following Haar distribution
rsample random sample with replacement
rwsample random sample without replacement
rcoy random covariance matrix from Normal
mkarma make arma without Normal errors
rtable random table following particular logistic model
rtable random table following particular log linear model
runifo uniform order stats
rexp exponential order stats
rnorm normal order stats
rpp Generalized Poisson Process
mkmat Regression matrix with specific condition and leverage

Distribution Tables and Quantiles:

- tks Kolmogorov - Smirnov
- tbin qbin Binomial
Control of Uniform Random Variables

There are several uniform random number generators including a congruential generator which we believe to be best for general settings, general congruential generators, tausworthe, generators, list generators, multiple disjoint congruential generators and potentially more. Each or any of these generators may be used as input for the non-uniform random number generators. In addition, one may define a new generator and use it as input to any of the random generators. The following commands control the use of Random Number generation.

\[
\text{seed(}<\text{function name}>)=<\text{expression}> \quad \text{Sets seed for random number generator}
\]

\[
\text{uniform(}<\text{function name1}>)=<\text{function name2}> \quad \text{Sets the generator of uniform to }<\text{function name2}>
\]

\[
\text{random function }<\text{function name}> [<\text{code}>] \quad \text{makes new random number generator}
\]

Experiment Control:

In the first implementation, the system will only do a full factorial experiment. The researcher may decide whether or not random numbers are reused (to reduce variances). The code generated will have the following structure:

Initialization code

\[
\text{Loop over parameters} \quad (\text{loop 1})
\]

Evaluate Statistics
End Replication Loop (Loop 2)

Store results of experiment

End loop over parameters (Loop 1)

The following parts of the language are for the specifying of what is to be done within this structure of a program. For the most part, the user will have, depending on which path through the User Interface he or she takes, will have little contact with this syntax of these commands. The User Interface will prompt the researcher for the specific information or will be able to fill it in with no help from the researcher.

**selectbuf** (<name>)
There are 6 code buffers:
- Declarations
- Initial Set-up
- Model Set-up
- Random Generation
- Statistic Evaluations
- Data Output

**return** (<string>, expr)
fun(expr) is returned as data to stat package desired. The <string> is the name by which it is to be identified. It may be any valid identifier name. However, when the data is loaded into the target statistical package, the name may be modified to suit the appropriate convention.

**genopt** (<optname>)
There are only 2 options at the moment:

**independent**:
- generate new random numbers for each set of parameters.

**reuse**:
- pull data from previous run.
into FORTRAN subroutines which produce different functional based on
different values of a input variable.

accumulate (identifier)

    Returns a structure which contains all the data which is contained
in the structure <identifier> over all the replications of an experiment
with a given set of parameters.

rep (expr1, expr2)

    Repeats expr1 a total of expr2 times. It is used primarily for the
bootstrap.

perform (boolean expr, expr)

    Determines the number of replications. Each experiment is per-
formed until the <boolean expr> is true or the number of times the
experiment has been performed exceeds the values of the expression.

jackknife (ident)

    Used in performing jackknife. The function produces a structure
of copies of <ident> all lacking a different element. The syntax:
var(fcn(jackknife(x))) produces jackknife estimate of variance.

Macros and procedures

    The Monte Carlo system has the ability to interface with any user
supplied FORTRAN function or subroutine. They simply put the com-
piled subprogram in a library called userlib and call it using the syn-
tax:

call <subroutine name>(<arg list>) for subroutines.

fortran (<subprogram name>(<arg list>) for functions.

    The processor also has the ability to make macros and local func-
tions. A Macro is just a collection of code which is grouped together for
convenience sake. It has no local address mode and can take no argu-
macro <macro name> {statements}
Local function definition:

function <function name>(<arg list>) {statements}
Section 5.3: Function Library

There are actually two function libraries. One is the System Library. It contains the routines needed to support the data structures, perform the operators, do input/output and things of that matter. The other library is the called the User Function Library. It contains the FORTRAN routines which generate the random quantities, calculate the statistics, calculate quantiles, percentiles and other things of use in Monte Carlo experimentation. It is written in standard FORTRAN and may be used outside of the Monte Carlo processor environment. It contains routines to do the all the functions listed in section 5.2.1 and the reader is referred there to find a list of functions in the library.

Random Functions

Section 5.2.1 contains a list of all the functions in the function library which generate random numbers. The Function Library will contain several different routines for generating random numbers from each probability distribution. For example, for the Normal distribution, there will be at least the Box-Muller method, Marsaglia's Monte Python Method, a rejection-complement method and an approximate inverted cdf method. Each method has it's advantages and disadvantages and we expect that some researchers will have a need to choose the appropriate method. For each distribution, there will be a default algorithm which we feel to be the best general purpose algorithm.
Section 5.4: Conversion Routine

The conversion routine takes the output from the FORTRAN program produced by the Program Generator and Loads it into a statistics package. There are actually two files produced by the FORTRAN program. One is the returned file, which is names <name>.ret. It contains all the data which was specified in the "return" statements in the experiment description file and all the information needed to recover that data. The other file is called the raw file or the accum file. All the values, specified in a "accum" command in the experiment description, are returned to this file. There may well be some duplication between the two files. The point of the second file is that if the researcher only returns a function of the results of the experiment and later decides that perhaps another function should have been performed on the results of the experiment, then the raw data will be available for use. Needless to say, the raw file can be quite large and care should exercised in dealing with it.

The following statistics packages may be used in conjunction with this Monte Carlo Processor:

BMDP
GLIM
ISP
Minitab
S
SAS
SPSS

The Converter will also produce a self describing file. That file is a character file. All numeric quantities are stored in a field of 20 characters, followed by a single blank. If the number is a real quantity, the decimal point may be anywhere in that field. Large and small numbers are stored in FORTRAN E format. All character strings are preceded by a numeric quantity (20 characters and a blank) which indicates their length. All fields are separated by a single blank character.

File Format

Number of data records (numeric)
Experimental Data records (described below)
Format of Parameter Records

Length of Name (numeric)
Name (character)
type (numeric)
0 = numeric
1 = function

Number of values (numeric)

Numeric Data Records or
Character Data Records (described below)

Experimental Data record

Length of Name (numeric)
Name (character)
Length of Data (numeric)
Data (numeric)

Numeric Data Record

Data (numeric)(length given above)

Character Data Record

Length of value (numeric)
Value (character)
Bibliography

Andrews, D. F. et al.,

Bartels, Robert,

Becker, Richard A. and Chambers, John M.,

Bratley, Paul, Fox, Bennett L. and Schrage, Linus E.,

Efron, Bradley and Gong, Gail,

Grier, David,

Hauch, Walter W. and Anderson, Sharon,

Knuth, Donald,

Malcolm, Michael A.,

McKnight, Barbara,
Validity and Efficiency in Inferences About Tumor Incident Rate, Proceedings of 144th Annual Meeting of ASA, 1984.

Rubenstein, Reuven Y.,
Availability

Although several preliminary versions of this program have been completed, the first program product version is still in production. It will run on a DEC VAX under Berkeley UNIX. (And, presumably, on a VAX running other versions of UNIX.) The first conversions will include converting it to run on a VAX under VMS and to run on non-DEC equipment running UNIX. We will also prepare a version which has the User Interface, Program Generator, and Converter running under UNIX and the FORTRAN Program to do the Monte Carlo experiment will be loaded to another for execution. Current plans call field testing in the third and fourth quarters of 1985. The first release will be in the fourth quarter of 1985. Enquiries should be sent to:

MCP Project
C/o David Alan Grier
Department of Statistics
GN-22
University of Washington
Seattle, WA 98105

Phone: (206)543/7237  9:00am to 12:00 pm
1:00am to 5:00 pm Pacific Time