AUTOMATED SIMULATIONS

Robert A. Stine
Department of Statistics, The Wharton School, Univ. of Penn.
Philadelphia, PA 19104-6302
stine@wharton.upenn.edu

1 Introduction

Fast, inexpensive computers have become the experimental laboratories of research statisticians. Exploratory simulations provide quick feedback on the validity of conjectures and often suggest modifications leading to new approaches. Simulations reject false conjectures and identify suitable candidates for the rigorous task of theorem proving.

In spite of the growth in computing power, the tedium of simulation programming often results in studies that fail to identify important properties. Software that automates the construction, running, and follow-up analysis dodges many of these problems. This relief frees time for thinking about the design and implications of the study rather than debugging code.

This paper describes the design and use of a computing environment that automates the tasks of describing, running, and analyzing simulations. The paper begins with a short illustration intended to convey broad concepts needed in an efficient simulation program. Building on this discussion, I describe the design of the software and use a prototype implementation to run a small simulation.

Interested readers may obtain this prototype and further documentation via anonymous FTP from the author at `hilbert.wharton.upenn.edu`. The prototype is written in X-Lisp and requires the Lisp-Stat software described in (Tierney 1990).

2 An Example

The features needed in a general-purpose simulation tool are evident in even a simple example. Suppose that we would like to understand the properties of three estimators of a population parameter θ, say $\hat{\theta}_1, \hat{\theta}_2$, and $\hat{\theta}_3$. The research interests are two-fold. One objective is to understand how each estimator behaves when applied to samples increasing size, say $n_1 < n_2 < n_3 = n$, from each of two populations identified by the distributions F_1 and F_2. The second objective is to identify which estimator performs best for a given combination of sample size and population. One estimator might be anticipated to be best for small samples, another for larger samples.

Consider the task of writing a simulation to investigate these questions. Assume that we have decided to run a simulation consisting of N trials (*i.e.* N samples for each combination of sample size and population). A traditional view of this task is to think of it as the process of computing the elements of a four-way array, say T. The first dimension of T identifies the trial of the simulation. The remaining dimensions represent the experimental conditions. In this case, let T be an $N \times 3 \times 3 \times 2$ array where $T_{ijk\ell}$ is the value on trial i of estimator j for sample size n_k and population $F_\ell (i = 1, \ldots, N, j = 1, 2, 3, k = 1, 2, 3, \ell = 1, 2)$.

The simplest way to compute this array is to nest a sequence of four iterative loops, as in the following pseudo-code:

```
[0] dimension T[N,3,3,2]
[1] for i = 1 to N do
[2]   for j = 1 to 3 do
[3]     for k = 1 to 3 do
[4]       for l = 1 to 2 do
[5]         let x = Gen(n[k],l)
[6]         let T[i,j,k,l]=Est(j,x)
```

where `Gen(n[k],l)` is a function that generates a sample of size `n[k]` from the distribution F_ℓ and `Est(j,x)` evaluates $\hat{\theta}_j$ using the given sample data. Given the functions `Gen`

and `Est`, this program is simple to write, but it ignores a number of important considerations.

Interest in comparison suggests modifications to this program. Most of these changes require spreading the computations otherwise hidden in `Gen` across the loops. For example, applying each estimator to the same samples obviously improves the comparison among the estimators. Since we are also interested in the effect of sample size, it is useful to correlate the samples of differing size. The most direct way is to nest the samples, letting $x_{ik\ell}$ be the first n_k elements of $x_{i(k+1)\ell}$. Given this choice, it seems most easy to generate the largest sample $x_{i3\ell}$ first and extract n_1 and n_2 elements for the smaller samples. Similarly, comparison across distributions suggests correlating these samples as well, such as by computing each from a common uniform sample,

$$x_{i3\ell} = F_\ell^{-1}(u_i),$$

where u_i denotes a sample of size n from the uniform distribution on $[0, 1]$ and F^{-1} denotes the inverse probability transform. This scheme also reduces the amount of calculation since it is typically easier to index a vector than generate a new sample. Although the inverse probability transformation $F^{-1}(u)$ is often an inefficient method for obtaining a random sample from a specific distribution F (notably the Gaussian), it does produce the most highly correlated random samples with given marginals in the bivariate case (Whitt 1976).

These enhancements complicate the naive algorithm. Each loop now includes the explicit calculation of intermediate results that are invariant within subsequent iterations. The order of the iterations is crucial:

```
[0] dimension T[N,3,3,2]
[1] for i = 1 to N do
[2]   let u = GenUniform(n[3])
[3]     for l = 1 to 2 do
[4]       let x1 = FInverse(u,l)
[5]       for k = 1 to 3 do
[6]         let x2 = Select(x1,n[k])
[7]         for j = 1 to 3 do
[8]           let T[i,j,k,l]=Est(j,x2)
```

Again, this is not such a hard program to write, but the coding is becoming more tedious, particularly with regard to indexing and holding intermediate results.

Were every simulation a static entity, this sort of programming would probably be adequate for most tasks. In practice, though, the structure of most simulations evolve with the accompanying research. Suppose that an initial run of the simulation suggests the need for a larger size sample, say $n_4 > n_3$. The associated changes to this code fragment are not localized. Not only does the dimension of `T` change in line `[0]`, but the addition of n_4 also requires changes to lines `[2]` and `[5]`. Alternatively, suppose that initial runs suggest that sample size is not an interesting factor. In addition to eliminating the loop at line `[5]` and the accompanying calculation of `x2` in `[6]`, this modification changes of dimension of `T` throughout the program. The interpretation of the dimension indices also change. After this modification, the third index indicates population distribution, not sample size.

Neither of these simple algorithms addresses problems associated with memory use and data analysis. Even in a simple simulation such as this it is impossible to hold all of T in memory for large N. One approach to this limitation is to accumulate recursively certain "sufficient statistics", namely the mean and variance, across trials. If all estimators were Gaussian, such a scheme would be ideal. However, this is a dangerous assumption from which to begin and one needs more flexibility in the choice of how results are to be accumulated. Some simulations need to compromise, retaining all of the results in some cases, partial results in others. Finally, both of these code fragments are parts of a larger program. This larger program is responsible for parsing input from the user and recording the results of the simulation for later use. Both of these tasks are often ignored in the haste to obtain some results.

The methods of this paper address each of these concerns while providing a flexible, extensible simulation environment.

3 Overall Design

The proposed simulation environment incorporates the features suggested in the enhanced algorithm of the preceding section. The software accumulates trial results in an

array, but one of smaller dimension and nature. The indices of the "array" are a combination of levels of experimental factors, omitting the trial index. For the illustration, each "cell" of this smaller $(3 \times 3 \times 2)$ array holds a collection of items associated with the results of the trials. For example, a typical cell index is of the form `(mean, size20, Gaussian)` rather than, say, $(1, 2, 1)$. Each cell accumulates its results independently of other cells. Some cells record every trial, others keep track of pertinent summary measures.

The flow of data into these cells is analogous to that of the enhanced algorithm. The process begins with a random number generator. Let $R(n, F, s)$ denote a pseudo-random number generator producing a sample of size n from the distribution F given seed s. Typically, F is the uniform distribution on $[0, 1]$ or the standard normal distribution denoted Φ. Let $s_i, i = 1, 2, \ldots, N$, denote the sequence of seeds associated with the trials of the simulation and $u_i = R(n, F, s_i)$ denote the associated random samples.

The next stage of the simulation transforms the data according to the experimental conditions, or factors. The factors associated with the simulation conditions come in two varieties. Blocking factors, like population distribution and sample size, control the transformation of initial random samples into the data used in the cells of the simulation. These determine the data used as input to estimation. Control factors, such as the choice of which estimator to use, determine the type of estimation scheme applied to the output of the blocking factors. Other examples of control factors include level of robustness, degree of smoothing, and method of variable selection. These factors do not affect the flow of data into the estimator, but rather change the manner in which the estimator operates. Most importantly for the simulation, the same data appear for all levels of these factors.

This division of the factors in a simulation provides a concise view of the underlying structure. A blocking factor is a set of functions that share a common domain and range. Compositions of functions from the blocking factors produce the data fed to the estimation routines. Some notation makes this discussion more precise. Suppose that the simulation has J blocking factors labeled $B^1, B^2, \ldots, B^J$, and that the j^{th} factor has b^j functions,

$$B^j = (B^j_1, B^j_2, \ldots, B^j_{b^j}), \tag{1}$$

where $B^j_b : D^j \mapsto R^j$ for $b = 1, \ldots, b^j, j = 1, \ldots, J$. Here D^j and R^j denote the common domain and range of the functions of the j^{th} factor. For trial i, the output of this sequence of blocks is the data

$$x^i_{b_1, b_2, \ldots, b_J} = B^J_{b_J} \circ B^{J-1}_{b_{J-1}} \circ \cdots \circ B^1_{b_1}(u_i) \tag{2}$$

where $1 \leq b_j \leq b^j (j = 1, ..., J)$.

As an example, consider the two blocking factors of the simulation in §2. These factors are population shape and sample size. Denote the factor for the two populations by

$$B^1_j(u) = F^{-1}_j(u), j = 1, 2,$$

where F^{-1}_j denotes the inverse of F_j. The factor for the three sample sizes is

$$B^2_k(x) = \pi_{n_k}(x) = (x_1, \ldots, x_{n_k}), k = 1, 2, 3,$$

where and π_{n_k} is the projection that takes the first n_k elements of its argument ($n_k \leq n$). Given the random sample u_i, the data of size n_k from distribution F_j for trial i is

$$x^i_{jk} = B^2_k \circ B^1_j(u_i).$$

The ordering of the compositions in equation (2) is important. For blocks whose functions do not commute, external information (*i.e.*, the user) must define the ordering. When the blocking functions commute, as with F^{-1} and π_{n_k} this example, efficiency becomes the guide. Efficient evaluation of compositions of blocking factors requires at a minimum intermediate data. As noted in §2, it makes most sense to apply F^{-1} first and then select the nested samples, leaving the 3×2 array of blocks with samples $\pi_{n_k} \circ F^{-1}_j(u)$. Efficient evaluation of these compositions requires saving the intermediate results $F^{-1}_j(u)$ so that the inverse distribution transformation is applied once to the entire sample rather than repeated on subsets of varying size. This is exactly the purpose of the loop invariants in the modified code of the introductory example. These loop invariants are an automatic consequence of manner in which the implementation in §4 applies the blocking factors.

A similar notation represents control fac-

tors. Each control factor C is a set of symbols $\{c_1, c_2, \ldots, c_k\}$ that denote variations on the estimation routine. Let K denote the number of control factors $C^1, C^2, \ldots, C^K$ associated with the estimator and let c^k denote the number of options associated with the k^{th} control factor. The value of the estimator computed from the output of blocks $(b_1, b_2, \ldots, b_J)$ for trial i is

$$\theta_{(c_1,c_2,\ldots,c_K)}(x^i_{b_1,b_2,\ldots,b_J}), \tag{3}$$

where $1 \leq c_k \leq c^k$, $k = 1, \ldots, C$. The simulation then records this result in the cell identified by the indices $c_1, c_2, \ldots, c_K$, $b_1, b_2, \ldots, b_J$.

This design is flexible with regard to how information is recorded in each of the cells. Let $T_m[b_1, \ldots, b_J, c_1, \ldots, c_K]$ denote the contents of the cell with these indices through m trials of the simulation. This information could be the entire sequence of results, or just the current mean and variance. Each cell of the simulation has an accumulation mapping $A_{b_1,\ldots,b_J,c_1,\ldots,c_K}$ which describes the effect of adding an additional trial to the current set of results in this cell,

$$\begin{aligned} T_{m+1} \quad & [b_1, \ldots, b_J, c_1, \ldots, c_K] = \\ & A(\, \theta_{(c_1,c_2,\ldots,c_K)}(x^{m+1}_{b_1,b_2\ldots,b_J}), \\ & \quad T_m[b_1, \ldots, b_J, c_1, \ldots, c_K]\,) \end{aligned} \tag{4}$$

One can use recursive schemes such as those discussed in Chan and Golub (1983) to update the means and variances as single values arrive. Tukey and Tukey (1992) discuss alternative accumulation strategies, focussing upon the choice of features of the replications to retain.

The analysis of a simulation requires the standard suite of tools used in the analysis of any designed experiment. Rather than recreate these tools, it makes more sense to take advantage of the existing tools. The strategy used here is to communicate with external software via standard ASCII files. Each cell of the simulation writes its symbolic index and contents to a file. The data written, of course, depends upon how the simulation results have been summarized via the mapping A in (4). It is also possible for programmable statistics packages like S or SAS to have the simulation software write a small script which instructs the package as to how to read the data file and run the analysis.

While advantageous to use external statistical analysis software, it is occasionally quite useful to have some "browsing" features built into the simulation. When the external ANOVA is run, it is likely that the data will reveal unusual results. Typical simulations make it difficult to recover the associated block of data so that it might be studied to learn the source of the anomalous behavior. This system records a subsequence of the seeds $s_1, s_2, \ldots$ so that one may rebuild the sample that produced the unusual behaviour. Graphical browsers of the results of the simulation enhance this capability by allowing the user to select an interesting trial from a plot and rebuild the associated sample interactively.

4 Implementation

This section briefly describes some of the methods used to implement this design. Consider the flow of data through the blocks in equation (2). The desire for clarity and simplicity suggests that the code for the simulation have a similar structure in which the functions of each factor are sequentially mapped over the results of the previous factors. While efficient, the storage demands of this direct method are excessive and something closer to the nested iterative scheme is better. Delayed evaluation streams (Abelson, Sussman and Sussman, 1985, §3.4) overcome the problems of inefficiency while retaining a clear, concise syntax. The resulting code makes the factor composition appear sequential while retaining the efficiency of nested iteration.

The prototype implementation uses object-oriented programming. Subclasses of the *Collection* class defined in SmallTalk (Goldberg and Robson 1983) represent both types of factors as well as the simulation cells. Each blocking factor is an instance of a *Dictionary*. The keys of each are symbolic labels for the functions, and the values associated with the keys are the functions themselves. Control factors are instances of the class *Set*. Instances of the class *SimulationCell* are ordered collections. Subclasses of this class specify the accumulation function A of (4). One subclass compresses the trial

replications, retaining only the information needed to update the mean and variance incrementally. The other subclass records everything. Each instance of a *SimulationCell* also knows how to plot, summarize, and write its data to a file. This last common method is essential in the last step of the simulation, namely communicating with external applications. When requested to write an external file, the simulation merely traverses its cells and tells each to append its data to a specific file.

5 An Example

This example returns to the illustration of §2. The three estimators the mean, median, and the 20% trimmed mean (the average of the middle 60% of the sample). The blocking factors are sample size (10, 20, and 30) and population distribution (normal and log normal). With this choice of populations, the generator of the input random samples is $R(30, \Phi, s)$ so that the u_i are $N(0, 1)$ samples. The first blocking factor is $B^1 = (I, exp)$ where I denotes the identity function and $exp(x) = e^x$. The second blocking factor is the sequence of projections $B^2 = (\pi_{10}, \pi_{20}, \pi_{30}) = (\pi_{10}, \pi_{20}, I)$.

To build this simulation, I begin by constructing the simulation factors. Hopefully the code is, parentheses aside, simple enough that those unfamiliar with Lisp can follow along. Here are the two blocking factors:

```
(def size-factor (make-dict
  (list (cons 'n=10 #'(lambda (x)
                (select x (iseq 10))))
        (cons 'n=20 #'(lambda (x)
                (select x (iseq 20))))
        (cons 'n=30 #'(lambda (x) x))))
  :name 'size))

(def dist-factor (make-dict
  (list (cons 'normal
              #'(lambda (x) x))
        (cons 'lognormal
              #'(lambda (x) (exp x))))
 :name 'dist))
```

Next I define the control factor and estimator. The options used within the case statement of the estimator function must agree with the symbols in the control factor. The leading colon in the name of the option set conforms to the optional argument convention used in Lisp-Stat (X-Lisp).

```
(def est-options
  (make-set '(mean trim median)
  :name ':estOpt))

(defun center (x &key estOpt)
  (case estOpt
    (mean     (mean x))
    (median   (median x))
    (trim     (trimmed-mean x))))
```

Finally I combine these to build the simulation object. The first argument defines the maximum sample size 30; the second is the estimator function. The next two arguments are the blocking and control factors. The order of the blocking factors determines the order of the function composition in (2). Here, B^1 is the population factor and B^2 is the sample size effect.

```
(def sim (make-simulator 30 #'center
   (list dist-factor size-factor)
   (list estimator-options)
   :generator #'normal-rand)
```

The following method directs the simulation to run 20 trials.

```
(send sim :run 20)
```

Subsequent calls to this method append additional trials to the simulation cells.

A browser allows the user to inspect the contents of any cell or to compare the values obtained by varying the levels of one factor. The underlying structure follows the familiar "model-view-controller" paradigm. The model is the collection of simulation cells, the view is a plot of data from a subset of cells, and the controller is an instance of the *SimulationView* class. The following command creates a simulation view.

```
(def simview (make-simulator-view sim)
```

This command augments the standard Lisp-Stat menu with an item that allows the user to select subsets of an associated collection of simulation cells. For this example, I asked for a plot of the simulation trials of the trimmed-mean for the lognormal samples, varying the sample size. The associated plot appears in Figure 1. In the figure, I have selected the

icon associated with a trial of the simulation which consistently yielded the smallest estimates for all sample sizes. Given this selection, further menu commands associated with this comparison plot instruct the simulation to rebuild the data associated with this trial.

In keeping with the example of §2, suppose that we decide to change the simulation by adding a new maximum sample size, say $n_4 = 50$. In this environment, I would simply change the definition of the size factor to be:

```
(def size-factor (make-dict
 (list (cons 'n=10 #'(lambda (x)
                (select x (iseq 10))))
       (cons 'n=20 #'(lambda (x)
                (select x (iseq 20))))
       (cons 'n=30 #'(lambda (x)
                (select x (iseq 30))))
       (cons 'n=50 #'(lambda (x) x))))
 :name 'size))
```

and create a new simulation with a larger maximum sample size with the following command:

```
(def sim (make-simulator 50 #'center
   (list dist-factor size-factor)
   (list estimator-options)
   :generator #'normal-rand)
```

The remaining definitions are not altered. If instead I decided to eliminate the size factor and have all of the trials be of size $n = 30$, the following command which eliminates the sample size factor would build the needed simulation program:

```
(def sim (make-simulator 30 #'center
   (list dist-factor)
   (list estimator-options)
   :generator #'normal-rand)
```

On systems with a graphical user interface (*e.g.*, a Macintosh), these changes are all a matter of a few cut and paste operations.

6 Summary

The computing environment shown here automatically produces a simulation program that incorporates several obvious experimental design features. These features eliminate unnecessary calculations and enhance the comparison of estimators across the experimental conditions. The automated nature of the software also eliminates the tedious of computer programming that can discourage the use of experimental simulations.

Clearly the software has room for improvement. A design based on the simple notion of composing factors does not require the complete factorial implementation shown here. For example, a special experimental design like a Latin square might be more appropriate. Ideally, one would like to have a program that could recognize from preliminary trials that certain interactions were non-essential and modify the factor composition appropriately. The object-oriented design permits extensions that can accommodate many of these enhancements through natural extensions of the object classes provided.

In terms of the implementation, I find it hard to argue that Lisp is the most efficient environment for simulation programming. Fortran or machine language programs will always be faster, once they are written and debugged. Still the comparisons of the efficiency of X-Lisp to S shown in (REFERENCE) suggest that X-Lisp is comparable in speed to this popular research package.

Bibliography

Abelson, H., G.J. Sussman, and J. Sussman (1985). *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA.

Goldberg, A. and D. Robson (1983). *Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading MA.*

Chan, T.F. and G. H. Golub (1983). Algorithms for computing the sample variance: Analysis and recommendations. *American Statistician*, 37, 242-247.

Tierney, L. (1990). *Lisp-Stat.* Wiley, New York.

Tukey, P. and J.W. Tukey (1992).

Whitt, W.(1976). Bivariate distributions with given marginals. *Annals of Statistics*, 4, 1280-1289.