NBER WORKING PAPER SERIES

IMPLEMENTING AND DOCUMENTING RANDOM NUMBER GENERATORS

David C. Hoaglin*

Working Paper No. 75

COMPUTER RESEARCH CENTER FOR ECONOMICS AND MANAGEMENT SCIENCE
National Bureau of Economic Research, Inc.
575 Technology Square
Cambridge, Massachusetts 02139

March 1975

*NBER Computer Research Center and Harvard University, Department of Statistics. Research supported in part by National Science Foundation Grant GJ-1154X3 to the National Bureau of Economic Research, Inc.

Abstract

As simulation and Monte Carlo continue to play an increasing role in statistical research, careful attention must be given to problems which arise in implementing and documenting collections of random number generators. This paper examines the value of theoretical as well as empirical evidence in establishing the quality of generators, the selection of generators to comprise a good basic set, the techniques and efficiency of implementation, and the extent of documentation. Illustrative examples are drawn from variout current sources.

Contents

IMPLEMENTING AND DOCUMENTING RANDOM NUMBER GENERATORS

1. INTRODUCTION

Across the fields of statistics and computer science, from theoretical to applied, simulation and Monte Carlo continue to play a significant role. The variety of clever applications is great, but often it seems that the technical foundations are shaky. The random-number generators, on which this whole experimentation structure rests, are still all too often incautiously selected, haphazardly implemented, and inadequately documented. Taking examples from among the available generators, algorithms, routines, and libraries, the balance of this paper examines prevailing practices in selecting, implementing, and documenting random-number generators and offers some recommendations.

First, however, we should look further at the question of prevalence: How widely are simulation and Monte Carlo used in statistical research? In preparing a position paper on publication of computation-based results [8] David Andrews and I had occasion to go carefully through the 1973 volumes of Biometrika and Journal of the American Statistical Association, counting papers of various kinds. We found that 20% of all papers involved simulation results, and the individual percentages in the three bodies of papers (JASA Applications, JASA Theory and Methods, and Biometrika) departed surprisingly little from the overall figure. Even without comparable data for the computer science literature, the overall conclusion is clear: simulation is an important component in quite a lot of research.

2. UNIFORM GENERATORS

A source of uniform random numbers is at the heart of almost all algorithms for generating non-uniform distributions, so it deserves a lot of attention and is the natural place to start. This observation is hardly new, but in view of the typical quality of available generators it still needs emphasis. For example, wherever there is a computer with a word size of 32 bits, one is likely to find the poor generator RANDU [10] -- its persistence has been remarkable. Of course, this generator (along with most others in common use) is multiplicative-congruential, and all such generators are well-known to produce output sequences which have regular structure. Specifically, the set of all n-tuples $(X_i, X_{i+1}, \ldots, X_{i+n-1})$ forms a lattice in Euclidean n-space [13]. For some high-accuracy multidimensional calculations this type of defect may render all congruential generators undesirable. Various schemes for permuting, randomly shuffling, or otherwise modifying the output of such basic generators offer reasonable improvement, but there is still much to be learned.

Returning now to the basic congruential generators, we should be aware of one important argument in their favor: they are the easiest to analyze theoretically. As a result we can determine (over the full period of the generator) several indicative properties of these generators and thus have a much clearer picture of what we can expect of them and what we cannot. The lattice structure of the generators provides the basis for the two most widely used theoretical tests: the spectral test

, 12] and the lattice test [2, 14]. Briefly, e spectral test in n dimensions looks at (in a andardized reciprocal scale) the distance tween adjacent hyperplanes in the most widely parated family of hyperplanes in the lattice of tuples produced by the generator. The lattice st looks at the length ratio of longest and ortest sides in a reduced basis for the lattice n-tuples. It is not surprising, then, that ese two tests are rather closely related. Still, may help our understanding to apply both tests d study the results in dimensions 2 through 6. at actual test criteria should we use? in uth's notation [12] for the spectral test we lculate C_n for n=2,3,4,5,6 and require that all ese values be at least 1 . This is the more ringent of the two criteria Knuth suggested, but cent empirical experience [9] suggests that it not unreasonably difficult to find multipliers ich meet this requirement. For the lattice test : us use L_n to denote the length ratio of long- : side to shortest side. Marsaglia suggested 4] requiring $L_n \leq 2$, and this seems sensible, ain for n=2,3,4,5,6. (Formulated in such simple rms as "$C_n \geq 1$" and "$L_n \leq 2$", the lattice test pears to be the more stringent of the two). Two ints summarize this discussion of theoretical operties: these theoretical tests are much tter for screening congruential generators than e known empirical tests, and no congruential nerator should be put into use without passing em.

3. A BASIC LIBRARY

:'s turn now to what we can do with a carefully osen uniform generator. What other generators uld we put with it to form a serviceable set most simulation purposes? We should not have ch difficulty agreeing on a basic library, and reasonable list should look much like this:

Continuous distributions

- Uniform (0,1)
- Gaussian (0,1)
- Exponential
- Gamma (and χ^2)
- Beta (and F)
- Student's t

Discrete distributions

- Uniform
- Binomial
- Poisson

umber of other distributions may suggest them- ves for specific applications, but any reason- e library should support those in the basic t.

each non-uniform distribution we of course it to use exact and efficient algorithms and oid such methods as inverse c.d.f. approximations and the Central Limit Theorem. Clever exact methods for the Gaussian and the exponential have been available for about ten years, but until recently the situation for the general case in most of the other distribution families was not so encouraging. Fortunately a number of new algorithms have appeared during the last year or so to improve matters. The work of Dieter and Ahrens (for example, [1, 6]) is particularly noteworthy here; their acceptance-rejection methods for the gamma distribution and the beta distribution [6] have the attractive property of requiring nearly constant time regardless of the parameter(s) of the distribution.

4. IMPLEMENTATION

Now how should we go about implementing our chosen random-number generators? The range of issues here is quite broad -- from choosing the level of programming language to being careful, in an assembly-language uniform generator, not to throw away significant bits when converting the result to floating-point. Let's look at some of the questions from the top down.

1. What should the generator return? For most applications the convenient form of output is a vector of random numbers, and this means we will be producing subroutines instead of functions. In some cases a function might be better, but we would want to balance this against the overhead (both in programming effort and storage space) of adding the function form to the library.

2. How should calling sequences (parameter lists) be structured? To integrate the routines as a library, we would put the common parameters first, as in

RANDOM (X, N, [other parameters]).

3. How should we organize the way in which non-uniform generators use a basic uniform generator or generators? Here it is likely to be cleaner if each routine which requires a source of uniform random numbers actually incorporates its own. This would consume little space and eliminate considerable subroutine linkage, especially in the more complicated rejection algorithms. By removing "side-effect" interactions among different non-uniform generators it should make complex simulation programs easier to debug.

4. How should we handle starting values? In order to reproduce results the user must be able to set the starting value(s) (usually for the basic uniform generator(s)) and recover the current value at any point in the sequence. For the user who wants a "random" start we can provide a routine which uses the system clock or some other such source.

5. In what language should we program the generators? This question may receive more varied answers than the previous ones. Many generators, especially uniform ones, have in the past been coded in assembly language because the result runs

faster and because most higher-level languages don't have the primitives for the operations involved. Now, however, it seems preferable to use higher-level languages (such as FORTRAN or PL/I) as much as possible. For one thing, this is the only sensible way to approach portability from one line of computers to another, and having machine-independent generators will facilitate replication of simulation studies, something we have largely neglected. Another important consideration is the correctness of the implementation: assembly-language generators are likely to have more bugs, and those bugs will be harder to isolate. One IBM/360 assembler implementation of Marsaglia's rectangle-wedge-tail algorithm for the Gaussian distribution [4] provides a good example. Because the programmer misused one of the machine instructions, the generator produced an excessive number of deviates with large magnitude (like 5 and 6!). It's reasonable to admit that one can gain a good deal of speed in most random-number generators by coding them in assembly language, but the conclusion has to be that we should never start at that level. Program the generator in a higher-level language and debug it thoroughly so that there will be a well-understood version to compare the assembly-language one against.

6. What should we do about testing? The simple answer, of course, is "Be thorough". This is old advice, but many generators seem not to get a very extensive workout. For example, apparently the only test applied to the Gaussian generator [4] mentioned previously was a chi-squared test based on dividing the real line into 20 intervals of equal probability content. Since each tail lies entirely within one of these intervals, there was no check on the tail part of the algorithm. A simple probability plot would have exposed the problem almost immediately. This example suggests a natural strategy: the testing should be designed to cover each segment of a complicated algorithm (in addition to the performance of the whole). This is valuable when the implementation is in a higher-level language, and it is vital when assembly language is involved. Testing also should reveal something about the comparative speed of the algorithm because this is often a more complicated question than theoretical calculations (of such things as the average number of uniform deviates used in a rejection algorithm) can answer. For example, W. M. Gentleman told me recently that on a Honeywell 6000-series computer the Gaussian algorithm of Brent [3] runs about 35% slower than the 1964 algorithm of Marsaglia and Bray [15]. Information like this is machine-dependent but still quite useful.

5. DOCUMENTATION

Finally we come to documentation -- the most important step in making a generator or library accessible to users. Here the procedure is straightforward, but lapses are frequent enough to demand a brief discussion. There are two basic aspects: use of the generator and its "pedigree". Documentation describing use is what every programmer will read immediately, and it should start with a precise statement of what the generator is and what it produces. (It may be that this goes without saying, but an earlier (1 July 1973) edition of the IMSL Library 1 Manual [11] did not tell what congruential generator was implemented in the subroutine GGU1; it was necessary to read the assembly code [7]. Fortunately this is no longer true in the latest edition.) Other essential details for use are the calling sequence or parameter list, restrictions on parameters (for example, the starting value), what other generators are used, and the default initialization.

To establish a generator's "pedigree", supporting documentation should report the specific algorithm (with information on its efficiency), relevant theoretical properties (especially for uniform generators, including any embedded in a non-uniform one), the sources of any previous implementations on which the present one is based, and the results of testing. Together, these should give the user an adequately detailed picture of the generator.

6. SUMMARY

This paper has briefly endeavored to give an up-to-date consumer's view of random-number generators. Specific recommendations cover uniform generators, the composition of a basic library, and principles of implementation and documentation. While a number of actual examples indicate that currently available generators and libraries often fall short of the best that we know how to do, it is reassuring to note that most of the tools needed for substantial improvement are ready to hand. We should now expect (and perhaps demand) the gap between possibility and practice to close rapidly.

REFERENCES

[1] J. H. Ahrens and U. Dieter, "Computer Methods for Sampling from Gamma, Beta, Poisson, and Binomial Distributions," Computing 12 (1974), 223-246.

[2] W. A. Beyer, R. B. Roof, and Dorothy Williamson, "The Lattice Structure of Multiplicative Congruential Pseudo-Random Vectors," Mathematics of Computation 25 (April 1971), 345-363.

[3] Richard P. Brent, "Algorithm 488: A Gaussian Pseudo-Random Number Generator," Communications of the ACM 17, 12 (December 1974), 704-706.

[4] Lovick Edward Cannon III, "Pseudo Random Number Generators for Statistical Applications," Technical Report 69, Department of Statistics and Computer Sciences, University of Georgia, August 1971.

[5] R. R. Coveyou and R. D. MacPherson, "Fourier Analysis of Uniform Random Number Generators,"

Journal of the Association for Computing Machinery 14 (1967), 100-119.

[6] U. Dieter and J. H. Ahrens, "Acceptance-rejection Techniques for Sampling from the Gamma and Beta Distributions," Technical Report 83, Department of Statistics, Stanford University, May 29, 1974.

[7] David C. Hoaglin, "Some Remarks on the IMSL Random Number Generator GGU1," unpublished.

[8] David C. Hoaglin and David F. Andrews, "The Reporting of Computation-based Results in Statistics." In revision for The American Statistician.

[9] David C. Hoaglin and Gordon Sande, "A Study of Multipliers for Pseudo-Random Number Generators with Modulus $2^{31}-1$." Presented at Joint Statistical Meetings, St. Louis, Missouri, August 1974.

[10] IBM Corporation, System/360 Scientific Subroutine Package (360A-CM-03X) Version III, Programmer's Manual. H20-0205-3, 1968.

[11] International Mathematical and Statistical Libraries, Inc., The IMSL Library 1 Reference Manual, Edition 4, 1975. (FORTRAN IV, S/370-360).

[12] D. E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison-Wesley, 1969.

[13] G. Marsaglia, "Regularities in Congruential Random Number Generators," Numerische Mathematik 16 (1970), 8-10.

[14] G. Marsaglia, "The Structure of Linear Congruential Sequences," Applications of Number Theory to Numerical Analysis (S. K. Zaremba, editor), 249-285. Academic Press, 1973.

[15] G. Marsaglia and T. A. Bray, "A Convenient Method for Generating Normal Variables," SIAM Review 6, 3 (July 1964), 260-264.