Behavior Research Methods, Instruments, & Computers
1984, 16 (6), 548-550

A Pascal version of a pseudorandom number generator

C. DONALD HETH
University of Alberta, Edmonton, Alberta, Canada

This article describes a random number generator routine based on a Pascal algorithm suggested by Lewis and Payne (1973) and Payne (1973).

Stochastic models of psychological processes have become increasingly important in many areas. Frequently, these processes are analyzed by means of computer models that simulate the formal properties of the process (e.g., Cornell & Heth, 1983; Eckerman, 1980; Spetch & Wilkie, 1980). A vital component to such computer simulations of stochastic processes is the random number generator.

Several options are available to a microcomputer user who wishes to do simulation modeling. For example, most microcomputer languages contain some provisions for random number generation. Unfortunately, these supplied functions lack several important properties. First, there is little or no documentation of the algorithm used to produce the numbers. Consequently, the statistical properties of the generator are difficult to evaluate (but cf. Strube, 1983). Second, they can exhibit undesirable limitations, such as a short "cycle length" (the number of times the generator can be used before it begins to repeat the sequence of random numbers). Finally, portability from one machine to another is a serious problem. Because simulation models are meant as formal statements about a phenomenon, their communication value depends to a great extent on the ability of another person to understand and, if necessary, replicate the theory described by the model. A model that uses, for example, the Apple II random number generator cannot be run on an IBM personal computer. Indeed, many random number generators are designed to discard easily an old random number sequence, making replication of the model with the same random number input impossible. Because many techniques of digital simulation require a replicable random number sequence (Bratley, Fox, & Schrage, 1983), this characteristic of built-in generators is especially undesirable.

As an alternative to built-in generators, a mathematical function can be written to generate pseudorandom numbers. Kaner and Vokey (1984) suggested a machine language pseudorandom function for the Apple II microcomputer. However, speed and efficiency of machine language routines are gained at the cost of portability.

Another option is to write a random number generator in a higher level language. For example, a model can be expressed as a Pascal program and the generator included as a declared function. Pascal is a good example because there exists one implementation, USCD Pascal, that can be run on such diverse computers as the PDP-11, the IBM personal computer, the Apple II, and the Motorola 68000.

A PSEUDORANDOM ALGORITHM

This article describes a random number routine based on an algorithm suggested by Lewis and Payne (1973; Payne, 1973). The Lewis and Payne algorithm is suitable for a higher level language implementation, like Pascal, because it does not rely on many mathematical calculations. Furthermore, it does not require intermediate computations to preserve the full precision of the computer's word length. Finally, the cycle length is not limited by the word size, as is the case with many other algorithms. In the version to be described here, for example, the sequence will repeat only after $2^{47} - 1$ numbers.

Lewis and Payne's (1973) suggestion is a variant of a "Tausworthe type" generator. This type of generator constructs sequences of numbers by taking previous numbers modulo some polynomial function of 2. Lewis and Payne extended this idea to the following: Set up an initial table of random numbers of length n. Iteratively select an entry, i, from the table, and compute the result of entry (i) exclusive or (XOR) with entry (i+m), where m is some constant. Replace entry (i) with the new number and step to the next entry for the next random number. Lewis and Payne reported that with the right values of n, m, and initial "seed" random numbers, this algorithm produces excellent random numbers. They also present a FORTRAN coding of this algorithm.

Implementing this algorithm in UCSD Pascal, however, presents some problems. The entries in the table must be treated at one time as integers (to produce the random numbers) and at other times as logical variables (to permit the logical XOR operation). Pascal was designed as a strongly typed language specifically to discourage such

Preparation of this article was supported by Grant A0280 from the Natural Sciences and Engineering Research Council of Canada. Requests for reprints should be addressed to the author at the Department of Psychology, University of Alberta, Edmonton, Alberta, Canada T6G 2E9.

```
Program demonstration;

type
  randlink = ↑randentry;
  randentry = record
    next : randlink;
    case boolean of
      true : (randset : set of 0..15);
      false : (randinteger : integer);
    end;

var
  currentrand, offsetrand : randlink;
  i : integer;

procedure randinitialize;
  const
    n = 47;
    m = 5;
  var
    temp : randlink;
    index : integer;
    seed : array[1..n] of integer;

begin
  seed[1]:=6936;seed[2]:=11137;seed[3]:=175;
  seed[4]:=28333;seed[5]:=8228;seed[6]:=23343;
  seed[7]:=16201;seed[8]:=525;seed[9]:=32646;
  seed[10]:=12998;seed[11]:=14044;seed[12]:=22459;
  seed[13]:=8155;seed[14]:=14560;seed[15]:=5428;
  seed[16]:=3057;seed[17]:=13500;seed[18]:=7489;
  seed[19]:=23956;seed[20]:=1631;seed[21]:=18724;
  seed[22]:=12979;seed[23]:=7543;seed[24]:=26891;
  seed[25]:=5076;seed[26]:=18818;seed[27]:=17248;
  seed[28]:=26679;seed[29]:=8706;seed[30]:=9342;
  seed[31]:=29575;seed[32]:=31530;seed[33]:=23069;
  seed[34]:=26123;seed[35]:=21236;seed[36]:=18077;
  seed[37]:=20080;seed[38]:=12260;seed[39]:=26133;
  seed[40]:=18581;seed[41]:=3331;seed[42]:=26261;
  seed[43]:=18650;seed[44]:=8271;seed[45]:=29635;
  seed[46]:=11322;seed[47]:=2239;
  index := 1;
  new(currentrand);
  temp := currentrand;
  currentrand↑.randinteger := seed[index];
  index := 2;
  while index <= m do
    begin
      new(currentrand↑.next);
      currentrand↑.next↑.randinteger :=
        seed[index];
      currentrand := currentrand↑.next;
      index := index + 1;
    end;
  offsetrand := currentrand;
  while index <= n do
    begin
      new(currentrand↑.next);
      currentrand↑.next↑.randinteger :=
        seed[index];
      currentrand := currentrand↑.next;
      index := index + 1;
    end;
  currentrand↑.next := temp;
end;

function rand : integer;
  begin
    currentrand↑.randset := (currentrand↑.randset -
        offsetrand↑.randset)+
        (offsetrand↑.randset -
        currentrand↑.randset);
    rand := currentrand↑.randinteger;
    currentrand := currentrand↑.next;
    offsetrand := offsetrand↑.next;
  end;

begin
  randinitialize;
  for i := 1 to 100 do
    writeln(rand:6);
end.
```

Figure 1. Listing of a demonstration program that writes out 100 random integers.

ambiguities; furthermore, logical operations are restricted to single Boolean variables rather than to arrays of Boolean values.

Two Pascal constructs provide a way around these difficulties. First, the case variant declaration of record types permits one to reference a particular area of storage in different ways. Second, the UCSD implementation of Pascal represents sets as arrays of binary digits. Hence, if A and B are both declared as sets, then (A−B)+(B−A) produces the exclusive or of the two arrays (here, "+" refers to the union operation on sets, and "−" refers to the set difference). If the result is now referenced not as a set, but as an integer, the Lewis and Payne (1973) algorithm can be realized. Whether this particular technique would work with other versions of Pascal depends upon the manner in which the Pascal set construct is implemented.

Figure 1 presents an example of this technique. Here, a random number function that operates on a table of random numbers is declared. To decrease the computation involved in incrementing array indices, the table is implemented as a circular list rather than as an array. The table is pointed to by the global variables *currentrand* and *offsetrand*. These variables must be included in the beginning declarations of a program, and should not be altered anywhere else. If dynamic storage is to be used anywhere else in the program, the user must be careful not to disturb the space allocated to the table. Finally, the table is initialized with starting values. Those given in Figure 1 were computed according to guidelines suggested by Lehman (1977).

The program given in Figure 1 demonstrates the basic idea by printing out 100 random integers. To use it in an application, one would include the constant, type, and variable declarations, the declared procedure *randinitialize*, and the declared function *rand*. In the main program, *randinitialize* would be called at the start. Subsequently, any call to *rand* will produce a random integer in the range of 0 to 32,767.

EMPIRICAL TESTS

It is a maxim of simulation research that a random number generator should be tested before use. The specific tests may depend upon the application. However, to assist users of the algorithm presented here, tests of both the speed of the algorithm and the randomness of its results were conducted.

The computational speed of the algorithm was compared with that of the random number generator supplied with the Apple Pascal language library. Both generators were used to calculate a series of random numbers on an Apple II microcomputer. The mean time to generate 10,000 random numbers in a simple loop was 16.6 sec for the Apple Pascal library function. The random function of Figure 1 was slower, taking 44.8 sec. The increased time, of course, is the cost of portability, and must be considered in light of the requirements of a modeling study.

There are numerous tests of the random character of a pseudorandom sequence. Three were selected here. The first was a test to determine whether or not numbers chosen according to the sequence are equally likely. The random function of Figure 1 was used to select numbers uniformly from the interval 1 to 100 according to an algorithm suggested by Bratley et al. (1983). A sample of 100,000 such numbers was selected, and the frequency of each number was tabulated. The distribution did not differ reliably from a rectangular expectation [$\chi^2(99) = 89.79$].

The second test evaluated the sequential dependency of the random numbers at different lags. The result of the random function was divided by 32,768. The product of this fraction with another random fraction at a given lag is an index of the dependency between those numbers at that lag. The mean of the products from 100,000 numbers was computed for lags of 1 through 20. None of these differed significantly from the chance value of .25 according to a test suggested by Lehman (1977). All of the 20 z scores were below 1.70.

Finally, a sequence of 100,000 random numbers was generated, and the sequence was examined for runs of consecutively higher or lower numbers. The sequence exhibited a total of 66,664 consecutive runs. According to Bratley et al. (1983), 66,666 runs would be expected. The difference was not reliable.

REFERENCES

BRATLEY, P., FOX, B. L., & SCHRAGE, L. E. (1983). *A guide to simulation*. New York: Springer-Verlag.

CORNELL, E. H., & HETH, C. D. (1983). Spatial cognition: Gathering strategies used by preschool children. *Journal of Experimental Child Psychology*, **35**, 93-110.

ECKERMAN, D. A. (1980). Monte Carlo estimation of chance performance for the radial arm maze. *Bulletin of the Psychonomic Society*, **15**, 93-95.

KANER, H. C., & VOKEY, J. R. (1984, June). A better random number generator. *Micro*, pp. 26-35.

LEHMAN, R. S. (1977). *Computer simulation and modeling: An introduction*. Hillsdale, NJ: Erlbaum.

LEWIS, T. G., & PAYNE, W. H. (1973). Generalized feedback shift register pseudorandom number algorithm. *Journal of the Association for Computing Machinery*, **20**, 456-468.

PAYNE, W. H. (1973). pseudorandom numbers for mini- and microcomputers: A generalized feedback shift register algorithm. *Behavior Research Methods & Instrumentaion*, **5**, 93-98.

SPETCH, M. L., & WILKIE, D. M. (1980). A program that simulates random choice in radial arm mazes and similar choice situations. *Behavior Research Methods & Instrumentation*, **12**, 377-378.

STRUBE, M. J. (1983). Tests of randomness for pseudorandom number generators. *Behavior Research Methods & Instrumentation*, **15**, 536-537.

(Manuscript received September 14, 1984;
revision accepted for publication December 28, 1984.)